Redesign and Enhancement of the Katja System

Internal Report No. 354/06

Patrick Michel

October 30, 2006

Contents

List of Figures

1 Introduction

Katja is a tool generating order-sorted recursive data types as well as position types for Java, from specifications using an enhanced ML like notation. Katja's main features are its conciseness of specifications, the rich interface provided by the generated code and the Java atypical immutability of types.

After several stages of extending and maintaining the Katja project, it became apparent many changes had to be done. The original design of Katja [11] wasn't prepared for the introduction of several backends [4], the introduction of position sorts [7] and constant feature enhancements and bug fixes.

The Java backend in particular was growing more complex with each feature added, necessary changes resulting in tremendous efforts. In anticipation of the introduction of attributes to Katja, a cleanup had to be done and a proper design had to be found.

With all those difficulties in the implementations, Katja never reached a closed and stable feature set. While core features were working in principal, aspects like imports combined with the use of position sorts had to be used with caution and the support of a Katja developer only. So with the ongoing redesign of the Katja system, the feature set, including the interface of the generated code, was redesigned and stabilized as well.

By supplying this report Katja reaches release status for the first time. Section 2 describes the changes made to the structure of the Katja system and the global architecture. Section 3 explains the Java backend in more detail and focuses on the ideas and considerations which lead to the new code generator design. Section 4 summarizes the features of the Katja system apparent to the user and gives an account of the long process of designing the interface of Katja. This section should also serve as a documentation to the interested Katja user. The report finishes with future work on Katja.

2 Redesigning the Katja System

The Katja system basically is a compiler, taking Katja source files as input and creating several formats as output. The general architecture therefore resembles the one of a compiler, going through steps like

- scanning input files
- parsing token streams
- constructing abstract syntax trees

- semantic analysis
- generating output data
- writing files

On an abstract level all these steps are done in fixed order and can be separated quite well. Unfortunately the Katja system did not make these separations visible in any way, neither in code structure nor in the architecture. Code fragments, packages and other logical units had a high coupling while maintaining low cohesion. Again this was not the result of a bad design right from the start, but from the constant evolution of the system.

The following sections will analyze parts of the system in detail and document how things are done now.

2.1 Architecture and Control Flow

2.1.1 Logical Structure

The systems inherent separation into frontend and backends was already revealed in [4] and as the notion is both useful and necessary it will remain. However, I will elaborate a kind of *production line* view on the system. Like usual for a compiler, one Katja run is a logical sequence of different stages, which are each completely done and taken care of before the next.

Defining fixed stages for one Katja execution will have the following benefits:

- Each stage has an assigned tasks and each task is done in exactly one stage, rather than being distributed over the runtime of the system.
- On completion each stage constitutes a "toolset" to the next stage, guaranteeing certain properties.
- Stages communicate with well defined interfaces and can be developed without full knowledge of the others.
- Code realizing functionality is separated from data, which in turn is passed between stages.

The idea of such a production line is inspired by the well-understood scanner and parser technologies. Together they solve a greater problem by splitting it up into parts which they solve independently and conceptually in a fixed order. The realization, however, is tightly coupled and interactive, though maintaining loose coupling and high cohesion.

I need to achieve the same separation of concerns, which is essential to breaking up larger systems into manageable parts. The Katja system therefore splits up into several artifacts, which work together on a low coupled basis to realize a number of tasks in fixed order. This allows them to interact and benefit from guarantees other parts provide and ultimately enables them to be implemented separately.

Figure 1 shows the stages of the Katja system.

	Stage	Tasks done
start-up	Main class creation	- IOHandler, ErrorCollection and basic configuration are present - Katja can be executed in parallel, using different configurations
	Main class execution	- it is no longer important if Katja was executed from shell, a Katja Ant-task or from within another program
	parameter parsing	- command line is completely processed, configuration is done - most basic usage errors are found - backend is selected - next stages can use the configuration object only
frontend	file acquisition and transformation	- "include" statements and file handling in general are done - syntactic sugar is eliminated, path names are made canonical - next stages can use a set of abstract syntax trees only
	sort structure creation	- basic checks are done - all known sorts and their corresponding information is present - next stages can use the sort structure only
	semantic checks	- the sort structure is consistent, complete and free of errors - next stages can use all the sort information they need
	backend logic	the control is given to the backend, as no more general tasks need to be done

Figure 1: Main stages of the Katja system

It is vital to notice at this point, that a backend appears in several different roles. Concerning the logical structure of the system, the backend is only one final stage in which the logic of the backend is the main problem. The backend therefore only appears in the last stage of Figure 1, where it is the center of attention.

Another role of the backend, however, is the addition of the complete logic necessary to produce one product of a specific application domain. The backend therefore is and has to be supplied as one separated source artifact, e.g. one Java package. Regarding the control flow, this does *not* mean the backend is only executed last, but the backend has to aid several stages by supplying domain specific logic. The next section describes this in more detail.

Stages are in general separated into more detailed tasks, giving guarantees to other subtasks. Those guarantees most often belong to one of the two categories:

- **problem solved**: By finishing the task a specific problem was solved, like reading a file from the disk or treating syntactic sugar elements in an input language. Once the abstract syntax is normalized, for example, the latter problem is solved. Subsequent tasks need and must not include operations belonging to the solved problem, thereby achieving a separation of concerns.

- **toolset**: By finishing the task a new or updated set of tools or information, together with operations on this information, is supplied. Following tasks must not use low-level or deprecated toolsets and should profit from getting a cleaned up view in many cases.

Ultimately this leads to code artifacts, which have a well defined purpose and can work with a small set of interfaces. Tasks solved by code artifacts are guaranteed to be of manageable size and can work with convenient sets of tools. A complete Katja run is therefore divided into small, understandable parts, which run one after each other.

2.1.2 Physical Structure

For the implementation of such an architecture it is vital that stages are implemented in a set of language artifacts which is as small as possible. However, there are other concerns which influence the development of such artifacts and the code structure therefore can't resemble the listed set of stages in detail.

Katja is capable of having several different backends, which can be dynamically switched between different executions. By design the developer of such a backend and the developer of the frontend need not work together. The Katja system is therefore split in three conceptual blocks, which supervise a set of stages but cannot implement them in completion.

Figure 2 shows the conceptual blocks which make up the Katja system. Block `Main` is the entry point for all applications using Katja and supervises the start-up functionality and all associated stages. Its implementation consists mainly of technical details. However, this block already needs to run backend code, as it has, for example, to be able to parse all user supplied options and get information whether all given parameters are correct.

The `Katja` block is the heart of the Katja frontend, supervising the main frontend functionality and all its stages. It is no longer concerned with many

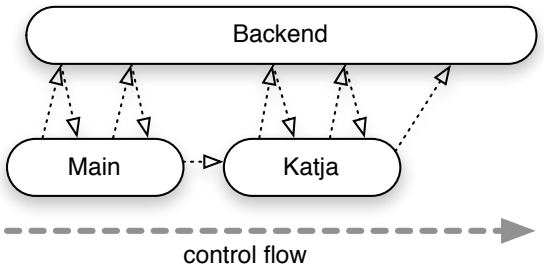

Figure 2: Conceptual Blocks of the Katja System

technical details and can work on a given configuration, input/output handler and error collection only. This block follows a straight forward protocol to solve the tasks of the different stages together with the selected backend.

The Katja execution is finished by the `Backend` block, which is no longer subject to any protocol and may finish the task of the backend. Note that at this stage of execution the backend is freed from all common concerns and has access to a concise representation of all specified data from the specification file. The backend is, however, involved in getting to this point, as many stages require backend specific interaction.

Each block is governed by one Java class, residing in a separate package.

- The class `Main`, realizing the start-up, is situated in the `katja` package, outside of the Katja package hierarchy and governs the `Main` block. The method `Main.run()` realizes the control flow and has to be called explicitly whenever Katja is to be started from within another Java program.

- The class `Katja`, realizing the frontend, is situated in the `katja.frontend` package and governs the `Katja` block. The method `Katja.execute()` realizes the control flow and represents one Katja execution with a given file, configuration, input/output handler and error collection.

- The interface `Backend` describes a backend specific block and is situated in the `katja.backend` package. It is implemented by each Katja backend, which should be situated in an equally named subpackage of `katja.backend`.

The reader should note that such an architecture might sound straightforward and simple at first, but is by no means the obvious and only choice when designing a system. The former Katja system, for example, used some kind of *recursive object structure* to handle imports of Katja specifications.

For each specification a `Katja` object was instantiated which shared some attributes with others while creating some new ones. So each imported specification was interpreted separately while some information was calculated together. This resulted in non-trivial system states, where the control flow was implicit and information was distributed in complex object structures.

This is but one example of possible design differences which resulted in tremendous efforts in implementing, understanding and enhancing Katja.

2.2 Sort Structures and Information Flow

For the implementation of system fragments we already achieved a separation of concerns, which in general leads to manageable code size and clearly defined tasks to be implemented. To keep implementations even more straightforward and maintain aspects like low coupling between stages and high cohesion within a fragment, we need a clear view on how information is obtained in different stages.

The most obvious starting point for information retrieval is the result of the parser step, which yields all the plain information given in a specification file as abstract syntax tree. This syntax strictly follows the construction of the concrete syntax, to avoid overloading the parsing step with domain specific knowledge and to keep it simple in general.

The transformation stage therefore works directly on the abstract syntax and eliminates all syntactic sugar elements, without changing the grammar. The result is again an abstract syntax tree, but with certain guarantees, like all sort declarations are on the top level of the specification and never nested in others.

With the introduction of position sorts, it became more and more apparent, that the abstract syntax was ill-suited to work with in following stages, like it was done in the former Katja system. Information about sorts was either directly taken from the syntax or from one of the various `Attribute` classes or from the `katja.helper` package, which implemented functionality to calculate non-trivial data specified in a specification file. There was no single *toolset* stages could work with and the mentioned functionality distributed over the system was neither coherent nor complete. My intention is to cut off the access of later stages to the abstract syntax and present a well designed toolset instead.

One important property of such a toolset is the representation of a sort, which is obviously needed for all functionality working on sorts and to talk about sorts in general. In the former Katja system these *runtime tokens* for sorts were their defining production or the `SortId` of the defining production.

One advantage was that most commonly needed information about the

sort was present in the token itself, like the number and names of selectors or the list element sort. With the introduction of syntactic sugar elements, positions and external sorts, however, it became apparent that the presence of this information was coincidental and was actually causing the dichotomy of information retrieval in Katja.

With positions alone the situation got worse, as there was no trivial runtime token for position sorts. In fact a simple root declaration in the abstract syntax causes the introduction of an arbitrary number of sorts and therefore can't be used as token.

These problems lead to the introduction of *sort structures*, which are collections of sort tokens, together with a variety of methods giving convenient access to all needed properties of sorts. The tokens are modeled and constructed in another abstract syntax, so the process of creating those tokens from a set of abstract syntax trees can be understood as transition from one syntax to another.

The basic sort structure is the `TermSortStructure` as it is unaware of positions and imports, it resembles the old `Attribute` functionality as close as possible. It is extended by the `SortStructure` to add position sorts as well as transparent imports. This structure is the starting point for the backend and is constructed in the "sort structure creation" stage.

The key features of a sort structure are:

- Each specified sort, whether created explicitly with a term production or implicitly with a root statement has one unique runtime token used to represent the sort in the Katja system.

- All properties of sorts, whether explicitly or implicitly defined by the specification, can be accessed in a convenient way.

- The sort structure gives an abstract and complete view on sets of specifications, without the need to handle syntactic sugar elements or implicit information retrieval.

- Access to the abstract syntax is neither necessary nor intended for later stages of a Katja execution. These stages can focus on their own task and need not help with the interpretation of Katja specifications, in fact it is considered harmful if they use other means to get specification information.

Each stage of execution is therefore part of the information flow and supporting it by either extending the quality of information for the next stage or presenting it in a more convenient and specialized manner.

A specification can also contain backend blocks, which wrap all backend specific parts of a specification. As backends may also interpret parts of the specification in different ways, it is recommended that they define their own sort structure, by extending `SortStructure`. The Java backend, for example, defines a specific `JavaSortStructure` to offer information like package names or Java super types of sorts to the later stages.

All sort and specification related information should be calculated in such a structure, so the `generate` method of the backend can focus solely on the creation of the backend specific product, like a Java package or an Isabelle theory.

2.3 Sort Descriptors

The `SortDescriptor`s, which are modeled with Katja, are used as runtime tokens in all sort structures. All sorts specified by a user at any time are modelled by a `SortDescriptor`, i.e. imported sorts are also modelled, as well as declared external sorts.

At the current state of the Katja System, a user can only use sorts in specifications, which he has either declared or imported. Common Katja sorts, like `KatjaList` cannot be used in specifications directly, as the existence of such sorts is backend specific.

So the core features of a sort in Katja are:

- The name of the sort.
- The file name and line number it was defined in.
- Whether it is defined in the root specification or was imported.

This is the only information available for all Katja sorts. In fact it would have been sufficient for a sort to only consist of a single name, to fulfill the task of being a runtime token, as no other information is needed for sort identity. Adding information like file name or line helps in differentiating conflicting sorts of specifications and makes error messages more expressive. A valid Katja specification, however, will have at most one sort for each possible name.

Given a sort descriptor, all other information has to be requested from the sort structure, which offers a variety of methods yielding basic as well as complex information. To offer a certain degree of static feedback to the developer, there are several subtypes of the class `SortDescriptor` defined:

```
SortDescriptor =
  TermSortDescriptor
```

```
| PosSortDescriptor ( String name, String filename,
                      Integer line, Boolean imported,
                      TermSortDescriptor baseSort,
                      PosStructDescriptor struct)

TermSortDescriptor =
  TupleTermSortDescriptor   ( String name, String filename,
                              Integer line, Boolean imported )
| ListTermSortDescriptor    ( String name, String filename,
                              Integer line, Boolean imported )
| VariantTermSortDescriptor ( String name, String filename,
                              Integer line, Boolean imported )
| ExternTermSortDescriptor  ( String name, String filename,
                              Integer line, Boolean imported )
```

Position sorts have two additional arguments, as they are derived from a base sort and belong to a position structure. There can be various position sort descriptors for one base sort, as well as for one position structure. Again the name of position sorts would suffice, but this information is considered essential for positions and is therefore integrated in the identity.

There are only two other sorts needed to give all artifacts of a specification an identity:

```
SelectorDescriptor  ( SortDescriptor parentSort, Integer count,
                      String name, SortDescriptor paramSort )

PosStructDescriptor ( TermSortDescriptor rootSort,
                      String suffix )
```

A selector descriptor completely describes one selector specified in the specification or generated by Katja. The latter is done for variants, if a selector can be lifted to it, and for tuple components which did not have a selector specified, so a default selector is created.

Position sort descriptors serve as runtime tokens for complete position structures, which are created by the `root` keyword.

2.4 Backends

Katja allows the definition of arbitrary many backends. The tasks to be solved by a backend are:

- The backend has to supply a name.

- The backend helps with the interpretation of command line parameters, by taking them away from a supplied list of arguments and returning a backend specific configuration containing the data. It also has to be able to print usage information for the user.
- The backend helps with the creation and checking of the sort structure, by supplying a specialized version. In addition to the checks supplied with the sort structure it helps checking the names of specified sorts by supplying a namespace.
- The backend gets the control flow to generate whatever product the backend wants to create.

So to implement a backend the user should supply at least:

- An implementation of the `Backend` interface.
- An extended `SortStructure`. If no extensions are needed in a specific backend, the class `SortStructure` can just be instantiated.
- An extended `Configuration` class, containing all backend specific configuration options. If no additional options are needed the backend can return an instance of `Configuration`.
- An implementation of the `Namespace` interface. The backend can return a trivial anonymous implementation of `Namespace`, if no constraints are needed.

Backends conceptually contain only static methods and should not have any internal state. All necessary data for each method call will be supplied by adequate parameters. Especially the created configuration and sort structure will be returned to the backend on each method invocation where it is needed.

A backend needs to be statically registered in the Katja frontend. The frontend therefore contains an enumeration of all backends, holding a singleton of the backend. All interaction of front- and backend is done through this instance, using the `Backend` interface. The enumeration supplies methods to identify a backend by name, which is supplied by the singleton.

2.5 Error Handling and Error Checking

Katja uses the exception mechanism to abort an execution and to report fatal errors. There is, however, a class of exceptions and errors which allows the execution to proceed, eventually reporting additional errors or finishing the execution.

Errors can arise in different situations for very different reasons and there are several kind of people involved. The users of Katja, the Katja developer and the backend developer can all be responsible for generating errors and have to handle errors generated elsewhere as well. This section covers how Katja and backend developers handle errors they detect, most important usage errors or errors in specifications.

The general error handling concept is as follows:

- Exceptions in the Katja system itself, caused by bugs, simply yield exceptions and don't use the error collection system.
- All usage errors or warnings are put into the error collection. If the execution should be aborted at this point, where the error was detected, an empty runtime exception is thrown. This results in the execution falling back to a given save point, where execution can either continue or the exception is translated into one defined in the interface.
- If the caller of a method (Katja or backend developer) does not want to proceed with the execution after an error occurred he has to check the error collection for errors.
- The main control flow aborts with a `katja.ExecutionFailed` exception or terminates the JVM if Katja was called from the command line.
- The backend control flow aborts with a `KatjaGenerationFailed` exception, which is therefore declared to be thrown in the various methods of the `Backend` interface.

Besides the usage errors, the start-up part of the frontend deals with, like wrong command-line arguments or syntax errors in the specification, there are those errors found in the semantics of a specification file. A specification can be checked for such errors at several stages.

If all checks are done in an early phase they are tedious to implement. Checks involving position sort names, for example, are much easier done after all position sort names have been calculated, instead of calculating them again for the checks only. On the other hand we can't check all conditions in the end, where all information is already present, since many calculations depend on the absence of some errors.

Errors to be checked in general are the following:

- Syntactic sugar is eliminated *without* error checking. Errors resulting from the use of syntactic sugar will be found on the normalized input.

(1) Duplicate or interfering sorts. This includes:

  - Duplicate term sorts, even if they are not in the same specification.
  - Interference of position sort names with term sort names.
  - Interference of position sort names with other sort names, due to the construction of the sort name by appending a suffix

  (11) Conflicts of sort names with specification names or between specification names.

  (12) Conflicts of position structure suffix names. The same suffix cannot be used in the same specification, due to the types introduced in Section 4.12 and it is very unlikely that using the same suffix for two position structures does not lead to conflicting sorts.

  Note that those conflicts arise between *all* sorts known to the Katja system, no matter if imported or not, since Katja does not support namespaces at the moment.

(2) Missing sorts, i.e. sorts which are referenced but not defined. This includes:

  - Sorts appearing in lists, tuples and variants.
  - Sorts appearing in root declarations.

(13) Duplicate selector names of sorts.

- Katja limits, including:

  (3) External, imported or position sort appearing in variants.

  ```
  external A
  B = A | CPos | D
  ```

  All cases of a variant need to be subtypes of the variant, which cannot be defined in all backends for sorts Katja does not create itself.

  (4) Recursive definition of variants.

  ```
  A = B | C
  B = A | D
  ```

  Such a definition immediately results in cyclic subtype relations.

  (5) Directly recursive lists.

```
A * A
```

This is no error in general, but can lead to problems in a backend. If a list is its own element sort it is also a subtype of its element sort and vice versa.

(6) Recursive dependency of terms, so no finite term can be constructed for some sorts.

```
A ( B, C )
B ( A, D )
```

Both sorts `A` and `B` need a term of the other sort to be created. Such problems can easily be avoided in practical applications by using either variants or lists in one of the tuple definitions, as lists can be empty and variants may have other cases, which are independent of the sort to be created.

(7) No higher order positions are allowed, i.e. position sorts mustn't appear in tuples or lists appearing in terms reachable from a root sort.

```
root A Pos
root B Occ
A ( D, E )
C ( DPos )
B * C
```

Though higher order positions have interesting applications, Katja does not support them at the moment.

(8) Katja identifier interference with backend identifiers. This includes:

- Interference with identifiers, keywords and literals of the backend language.
- Inability to define all identifiers in the backend, which are definable in Katja's lexic.

- Backend specific checks, which include:

(9) Definitions for all external sorts are present and conflict free. The Katja frontend does not consider multiple definitions of the same external in different specifications to be a conflict.

(10) Other backend specific checks.

These twelve checks are done in different stages of the execution, there a four categories in which they can be done. Checks are mostly done when they can be checked in a convenient and natural way and always before other stages depend on them. The categories, including the checks done in them, are presented in the order they are executed:

1. During the creation of the standard sort structure:

   (1) Duplicate term sorts are found the moment the duplicate is inserted into the sort structure. Note that duplicate externals do not yield an error directly, but are checked later for consistency.

   (1) Interfering position sorts, either with other positions sorts or term sorts, are found on their insertion to the structure as well.

   (2) Missing sorts *can* be detected on position structure creation.

   (7) Higher order positions are detected on construction of position structures.

   (11) Names of specifications are gathered and checked against all created sorts, as well as checked for conflicts.

   (12) Suffix names of position structures are checked against all others at the creation of the structure.

2. During the creation of the specialized sort structure, but called from the super constructor:

   (1)(9) Duplicate externals are checked for consistency in the specific sort structure, but in a method called from the super constructor. Backends can use this to check if all externals are defined.

3. After the creation of the sort structure, called from the frontend:

   (2) The sort structure is explicitly checked for missing sorts.

   (13) The sort structure is explicitly checked for duplicate selector names in sorts.

   (3)-(5) All those semantic issues cannot be detected by name-based construction and analysis; special checks are invoked on the sort structure.

   (8) The frontend checks all sort names against the `Namespace` provided by the backend.

4. After the creation of the sort structure, called from the backend:

(6) This is only checked in the Java backend for now, as methods making the check easy are defined there. The Isabelle backend has its own checks, as it has to find witnesses.

(10) All backend checks, which can wait for the constructor to be finished, should be made here. The Java backend, for example, has to check all sort names against the top level package they are generated to.

The creation of a sort structure is a difficult task and there are some implicit rules which mustn't be violated at the moment. Making these rules explicit isn't straightforward in Java and would involve changing the design decisions made for the sort structures.

First of all you can't call most of the methods of a sort structure from within the constructor, as most methods work only with the guarantee that all sorts are already known to the structure. The results of those methods are also cached in general, so calling them to early will yield both a wrong result and damage the cache. Methods used in the constructor are therefore documented, explaining why it's safe to call them. Also after the creation of a sort structure some checks have to be done, before calling of recursive methods is allowed.

As the specialized sort structures are subtypes of the normal sort structure, their super constructor is called first. As some checks need to be done in the super constructor, which can in turn be done only by backend specific code, this super constructor calls methods of the specialized sort structure, before attributes are initialized. Therefore the super constructor calls a special initialization method first, before creating the structure, in which specialized sort structures have to initialize all attributes they intend to use in the checks. As the backend block data will have to be parsed for those checks to be done, the sort structure will need some of its attributes to safe the results.

Backend specific checks, which are done after the creation of the sort structure are defined in the specialized sort structure as well, but are executed from the frontend and therefore mustn't be called from the backend itself.

All these rules can be summarized as some kind of *usage protocol*, which cannot be enforced in Java directly, as conditions change over time. It would, however, be possible to minimize the possible usage errors with a different design, but this was simply not done in the current Katja system.

2.6 Bootstrapping

Katja is a system utilizing itself in greater parts of the implementation. It was created by constant bootstrapping and continues to do so with each feature added. The significant changes to the complete Katja system made it necessary to clearly separate and understand the role of each artifact in the projects build process.

This had not been done to this degree so far, so I had to go back one step and unfold the bootstrap cycle. I copied the complete project to be allowed to freely make changes in certain parts of the project, without breaking it. This was necessary, for example, to adjust the `katja.common` package, without the need to adjust the generated classes immediately. I kept one version running and able to generate code, while refactoring the other.

This would not have been necessary with a build process which is really aware of the bootstrapping going on, shown in Figure 3.

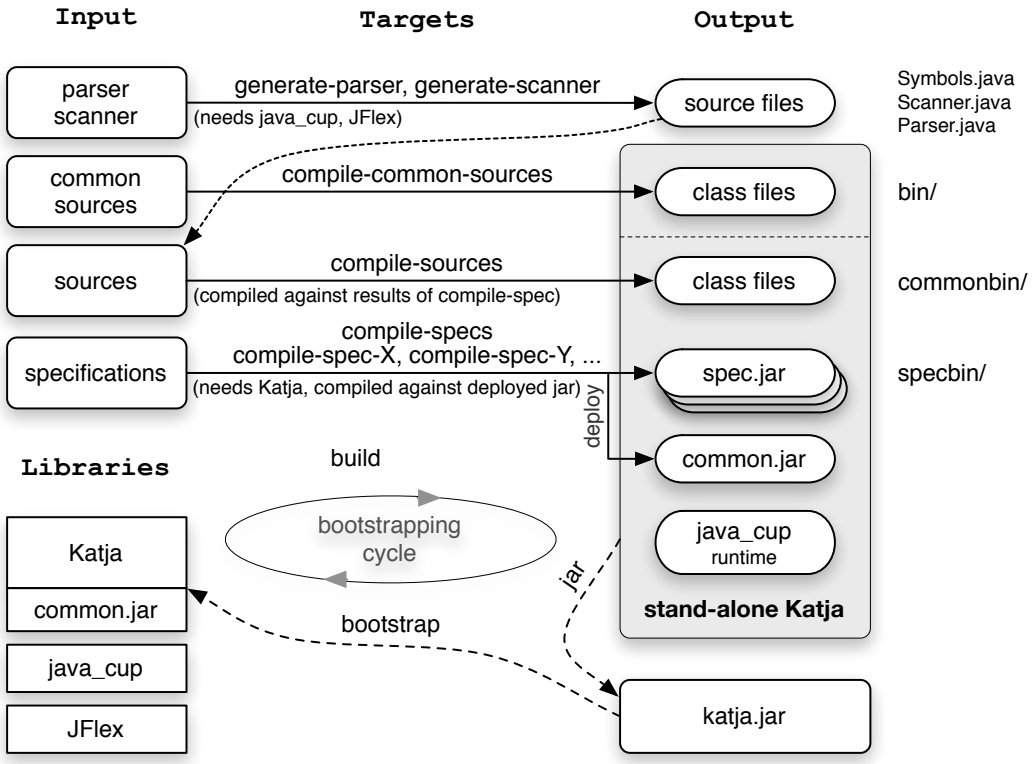

Figure 3: The Build Process of the Katja System

It has to be possible to adjust all different parts of the Katja system and integrate a new feature, while still being able to compile them all, without

breaking the system. Changing the common classes, for example, would result in breaking the compatibility to other code parts using them, if there is no stable version maintained. The specification libraries always have to run with the version of `katja.common` they were created for.

The old Katja system always compiled all parts under development together, so they had to be consistent. This was not achievable for a bootstrapped system, as the generators always need old libraries to work, but create code for the new. It is now possible, for example, to do the following when adding a feature to Katja:

1. Add support for the feature of the generated code in the sources and compile them.

2. Use the `jar` target, to create a version of Katja including the adjusted generators and compile a test specification.

3. Adjust the common classes, so they support the new features and work together with the test specification. It is still possible to compile all parts of the system and create a new `katja.jar`, as the modified version of `katja.common` is used only for the deployment of generated code, not to run Katja itself.

4. Iterate the process of adjusting the generators and compiling the test specification, i.e. compile the generated code against the deployed `katja.common` version, which is the adjusted one.

5. Bootstrap as soon as the generators are stable again and all necessary changes in the common classes are done. After another build of Katja, the developer has to adjust the system to work with the new specification libraries, as well as the new common classes. As soon as the system compiles again it should have reached a new fixpoint.

6. The system has done one bootstrapping cycle, without the need to defer, or temporarily take back, changes made in parts of the system and to build and test them separately.

It turned out to be natural to adjust the common classes first, whenever such a basic change was necessary and adjust the generators afterwards to reflect those changes. This is now easily doable with only one copy of Katja.

3 Redesigning the Java Backend

The Java backend started as a quite simple set of generator classes. As `SortId`s where the runtime-tokens for sorts, there was a generator for each kind of sort. Those generator classes worked directly on those tokens, using `Attribute` classes whenever they saw fit and calculating some information they needed by themselves.

As different sorts often share some code parts the generators where put into an inheritance hierarchy, shown in Figure 4. Term factories, visitors, folds, switches were all generated to standalone classes, but had nothing in common with Katja elements.

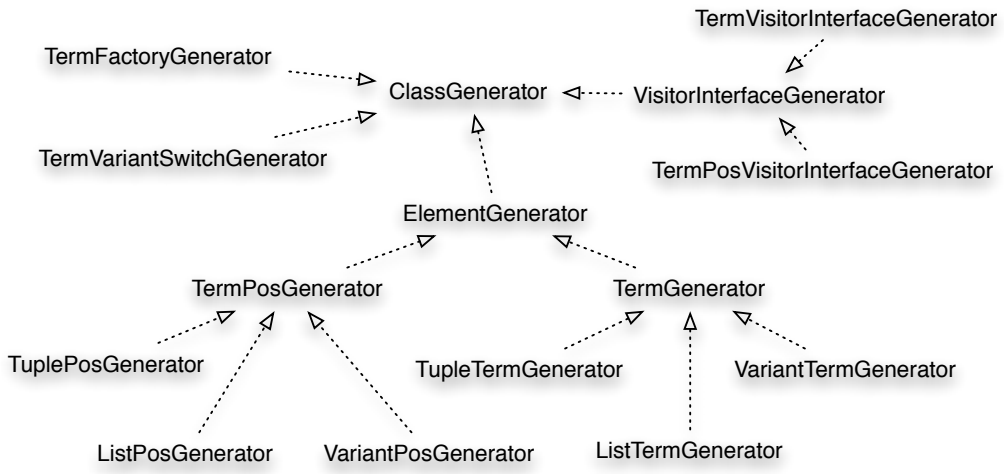

Figure 4: Excerpt of the old Generator Hierarchy

This resulted in the code creating one single class to be distributed over a large inheritance hierarchy, the criteria for its position in the hierarchy only being in which leaves it was needed. Code output was also distributed over the complete hierarchy and with the control flow alternating between code fragments in all classes it was a challenge to make changes. The idea to reuse code to create, for example, class headers or import statements resulted in a overly complex structure, with virtually no separation of concerns.

Changing aspects of the generated code, which had to be done differently for different sorts, meant adjusting code in virtually all classes. Finding code realizing a certain aspect was a challenge and all classes had to be searched. The decision to split the inheritance hierarchy into terms and positions first, followed by a split into the three different sort types, seemed straightforward at first. While adding more features to the Katja system, it became apparent

that often code could have been shared between terms and positions of the same type, but not between different types. In Java, however, the separation had to be done in exactly one order.

The layout of the generated code was distributed over the Java backend, changes in the generators often resulted in misplaced code fragments or tedious compiler errors. Though layout isn't a big issue for a code generator, it is in general considered helpful to be able to read its output anyways.

3.1 An Abstract Syntax for Java Code Generation

I decided to introduce a far more general approach to code generation. Again I want to achieve a best possible separation of concerns, while keeping developer efforts low. Therefore I introduce a third abstract syntax to Katja, which models Java source files up to a convenient, yet powerful level. Code generators will then create an abstract model of the code, rather then printing characters to a stream.

This approach has several benefits:

- Code generation and layout, besides the layout of algorithms, is completely independent of the generators and can be done in a general context.
- Code generators can start with an abstract, but technically complete source file and refine it step by step.
- Import statements can be created automatically.
- It is trivially possible to create interfaces from classes.
- It is possible to derive specialized classes from existing ones.
- It is possible to decorate the abstract model in the future, to add features like Javadoc or code annotations.
- Many properties of the output file can be statically checked and many errors in the output can be prevented altogether.

The intention of the Java model is not to completely describe all possible Java programs, but restrict the user, in a convenient way, to the features he really needs. Such features include, for instance, inner classes or final method parameters, but exclude static initialization blocks and static imports. The decision what features can be modeled was completely driven by need and what could be easily added.

The technique used, however, should make additions to the model possible, with only small adjustments to be done. In most cases it is even possible to use variants in the extension of the model, so only the unparser of the model has to be adjusted and code generators need not be touched at all.

3.1.1 File and Class Model

All sorts defined in the model specification are prefixed with `M`, to avoid conflicts with other names and keywords right from the start. Files and classes are defined as follows:

```
MFile  ( String name, String packageName, MClasses classes )
MClass ( MModifiers modifiers, MClassType type, MTypeDef name,
         MTypes extend, MTypes implement,
         MAttributes attributes, MMethods methods,
         MClasses nested )
MClassType = INTERFACE () | CLASS()
MModifier  = PROTECTED () | PRIVATE () | PUBLIC () | STATIC ()
           | ABSTRACT () | FINAL ()
```

The Katja view of a source file is just a name, a package name and a list of classes. Classes, however, need more explanation. A class starts with a list of modifiers, which are for simplicity neither constrained in order nor appearance anywhere in the model. As far as the model is concerned there is no real difference between a class and an interface, so this differentiation becomes a property of the model class.

The list of modifiers is normally followed by the name of the class, but as of Java 1.5 this name can be a generic type definition, which is allowed to have other types as bounds in wildcards and can be arbitrary complex. It is, in general, necessary to model this definition for the automatic import system to work. Section 3.1.4 will describe in detail why and when it is necessary to model types in detail. The following properties of a class are straightforward again and don't need further explanation.

3.1.2 Attribute Model

Attributes are modeled down to the optional initialization, which is then given as simple string. Again the list of modifiers is not constrained by the model.

```
MAttribute ( MModifiers modifiers, MAnyType type, String name,
             MCodeFragment initial )
MCodeFragment ( String part, MTypes needed )
```

Whenever the model stops going into detail there has to be a mechanism of specifying which types occur in the plain string given, so analyzers and the import generator know what do at this point.

The string part supplied in the code fragment does neither contain the *equals*-sign, nor the *semicolon*, but the expression needed to initialize the attribute only. At present state the model does not assume attributes to be initialized with an expression taking up several lines; many attributes will not have an initializer at all.

The current Katja system, however, uses the code fragment of static attributes to assign a complete anonymous class implementation in one line. Those are trivial anonymous class implementations, not of any interest to the possible reader of the source code, potentially appearing many times in one source file.

The special string `$attributetype$` is replaced by a string containing the attribute type, as often as it appears in the code fragment. This allows the developer to conveniently cast the expression to the attribute type or call a constructor of that type.

3.1.3 Method Model

Method signatures are completely modeled, method bodies are given as lists of strings together with a list of types occurring in the strings, which is the same mechanism used as in attribute initializers.

```
MMethod ( MModifiers modifiers, MTypeVariables generics,
          MReturnType returnType, String name,
          MParameters parameter, MTypes exceptions,
          MCode body )
```

The return type of a method can be absent, in case of a constructor definition, which is also modeled. This should not be confused with the `void` return type, which is modeled as normal Java type.

```
MReturnType = MNone()
            | MAnyType
```

The model explicitly allows final and variable argument parameters to be used, by extending the type system model explained in Section 3.1.4.

```
MParameter ( MParamtypeDecl decl, String name )
MParamtypeDecl = MParamtype
               | MFinal ( MParamtype type )
```

```
MParamtype = MVararg ( MAnyType type )
           | MAnyType
MCode ( Strings lines, MTypes needed )
```

Code lines supplied in the method body will be automatically ended with a newline character and moved to the methods nesting level. The developer should therefore only prefix lines with tabulators or spaces if they are adequate in the algorithm to be modeled itself.

The special string `$returntype$` is replaced by a string containing the return type of the method, as often as it appears in the method body code. This allows the developer to conveniently cast expressions to the return type or declare local variables of that type.

3.1.4 Type System Model

The model allows to precisely define all different types that occur in a Java program. This is done for several reasons:

- Import statements can be automatically generated, if needed. By adjusting the file unparser only, it is possible to optimize imports, generate all types full-qualified or use short names for all types, detecting conflicts automatically.
- The model helps the developer to decide which types can be used in which situation. An array type, for example, mustn't be declared to be thrown by a method.
- The model explicitly knows Katja sorts, and allows the developer to integrate them into the model.

The current policy is to generate all Katja sorts full-qualified and import all Java sorts needed in the implementation.

The basic Java types are those types defined either by Katja or Java, in both cases it is possible to specify arbitrarily nested classes.

```
MNonGenericType = MJavaType ( String name, Strings importNames )
                | MKatjaType ( SortDescriptor sortDesc )
                | MInnerType ( MNonGenericType outerClass,
                               String innerClass )
```

These are the only type system sorts in the model, which can or have to be unparsed full-qualified. A Java type is simply a string, together with a list of needed imports to satisfy the type. Java types should be used like this:

```
MJavaType("NE", "katja.common.NE")
MJavaType("ArrayList", "java.util.*")
MJavaType("void")
MJavaType("Integer")
```

It it never necessary to specify more than one import statement for a Java type, as long as the developer uses the type system model. At the present state there is, however, no reason to forbid JavaTypes like

```
MJavaType("List<Set<Term>>", "java.util.*", "spec.syntax.Term")
```

to be declared without using the model, but it is discouraged. One of the ideas of the type system model is to give the developer an abstract tool, to express what he needs. Generic facilities can then analyze the code and, for example, generate import statements following a given policy.

The next step is to allow generic type parameters, with wildcards and optional bounds.

```
MType = MNonGenericType
      | MGenericTypeApp ( MNonGenericType type,
                          MTypeOrWildcards types )
MTypeOrWildcard = MType
                | MWildcard ()
                | MWildcardLB ( MType bound )
                | MWildcardUB ( MType bound )
```

There are important differences between the application of a type constructor in Java and the definition of one, concerning the number of bounds of type variables and the use of wildcards. Therefore type definitions used as class or interface name are also modeled.

```
MTypeDef = MJavaType
         | MGenericTypeDef ( MJavaType type,
                             MTypeVariables typeVars )
MTypeVariable ( String name, MTypes bounds )
```

The next step is to allow array types, which cannot be used at several places.

```
MAnyType = MType
         | MArrayType ( MAnyType type )
```

In Section 3.1.3 we have already seen how these types can be made to variable argument types in case of method parameters.

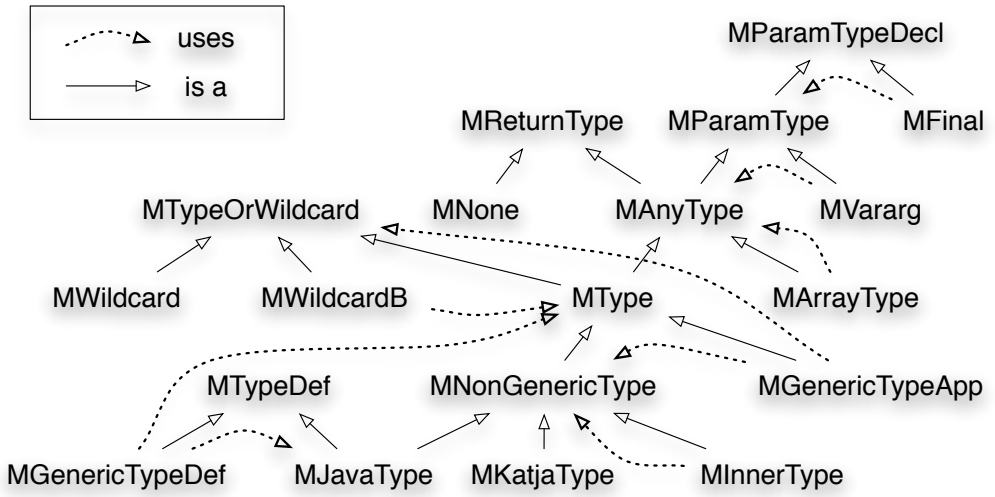

Figure 5: The Type System Model of the Java Backend

Figure 5 describes the complete type system model, showing subtype, as well as usage relations.

A simple map, containing strings as keys and sets of a Katja sort as values, would be defined as:

```
MGenericTypeApp(MJavaType("Map", "java.util.Map"),
    MJavaType("String"),
    MGenericTypeApp(MJavaType("Set", "java.util.Set"),
        MKatjaType(sort)))
```

The generic type definition, used as class name in the specification's tuple position class, is modeled like this:

```
MGenericTypeDef(MJavaType("Tuple"+suffix), MTypeVariable("T",
    MJavaType("KatjaTuple", "katja.common.*")))
```

3.1.5 Unparsing the Java Model

To actually get to a source file from a given `MFile`, a generic unparser is supplied by the Java backend. This unparser is realized as `MFileInFile` visitor, i.e. a simple visitor implementation on the model position sorts.

The unparser first scans the whole model for types to import and generates a list of needed import statements. It proceeds with unparsing the classes and transforms the model by filtering out unneeded attributes and

methods, e.g. private methods in interfaces. The unparse process then continues by visiting all parts of the model.

The Java model concept is therefore completely independent of Katja sort generators and can be used for other source files as well. It became apparent, however, that the class namespace of a generated Katja package should not be unnecessarily polluted by generating arbitrary many auxiliary classes.

The general concept of class generation for specification packages was therefore changed in several points:

- Only sort interfaces and one specification class are allowed at the top level. The name of the specification class is given in the specification file and is checked against all sort names.
- Sort implementation classes are static inner classes of the sort interfaces.
- Switch classes and interfaces are static inner classes of the corresponding variant sort interfaces.
- Visitor and fold classes and interfaces are static inner classes of the sorts they belong to. There is no such thing as a *specification visitor*.
- The specification class contains all methods of the former `TermFactory`.
- All other specification unique classes are static inner classes of the specification class. At the moment these are only the position sort classes of position structures (see Section 4.12).

3.2 Generation Aspects

With the Java model and the `MFile` unparser many concerns of code generation are already dealt with, but the main task of the Java backend remains unsolved so far. The former design of the generator classes, which is already described at the beginning of Section 3, had to be changed to something more reasonable. A better design should have the following features:

- Generator code should be shared for different types of sort generators.
- Most of the generator code dealing with one Katja feature should be located in the same source file.
- Unrelated generator code should be separated in different source files.

These features can again be summarized under the more abstract phrase "low coupling, high cohesion" and help with finding a valid separation of concerns.

I decided to introduce the notion of *generation aspects*, which represent everything that has to be done to achieve one given aspect of the generated code. Generator code is not split by the type of sort it generates, but by the type of feature it generates. The obvious benefit of such a design is that strongly related code is located in the same file and can easily be shared and understood.

The reader should note, that the introduction of the Java model was vital to make this design work, as the creation time of the model and the actual generation time of the code are no longer mixed up. It should also be noted, that there is no complete separation of generation aspects in non-dependent modules. There are very few generation aspects, which can be taken away from the system without harm, as the generated Java code is strongly related. The interaction points between aspects, however, are reduced to a minimum, like naming conventions of class names and methods, or the presence of a generated method at all.

The general architecture of the Java Backend is also the one of a production line, but involves a much simpler protocol. The control flow is governed by the `SpecificationGenerator`, which creates all sort classes and the specification class, by invoking the production line of aspects for each of those.

It is thereby guaranteed, that each generation aspect has seen

- every sort, together with it's interface describing the sort.
- every sort, together with it's class implementing the sort, if the sort has an implementation.
- the specification class, *after* all sorts of the specification have been shown to the aspect.

The generation aspect has the possibility to change or extend the given class model and has to return the modified version to the specification generator. For convenience reasons, the abstract `GenerationAspect` class has default implementations for each method, dealing with a specific type of sort, calling the more specific version. The developer has then the option to

- Override the generic `SortDescriptor` method. The default implementation can then be called as well, so specialized methods can be used too.
- Override either or both of `TermSortDescriptor` and `PosSortDescriptor` versions.

- Override any needed case of tuple, list and variant sort methods, for term and/or position case.

The specification generator always calls the generic sort descriptor method and a special method for the specification class. Figure 6 shows all aspects of the Java backend.

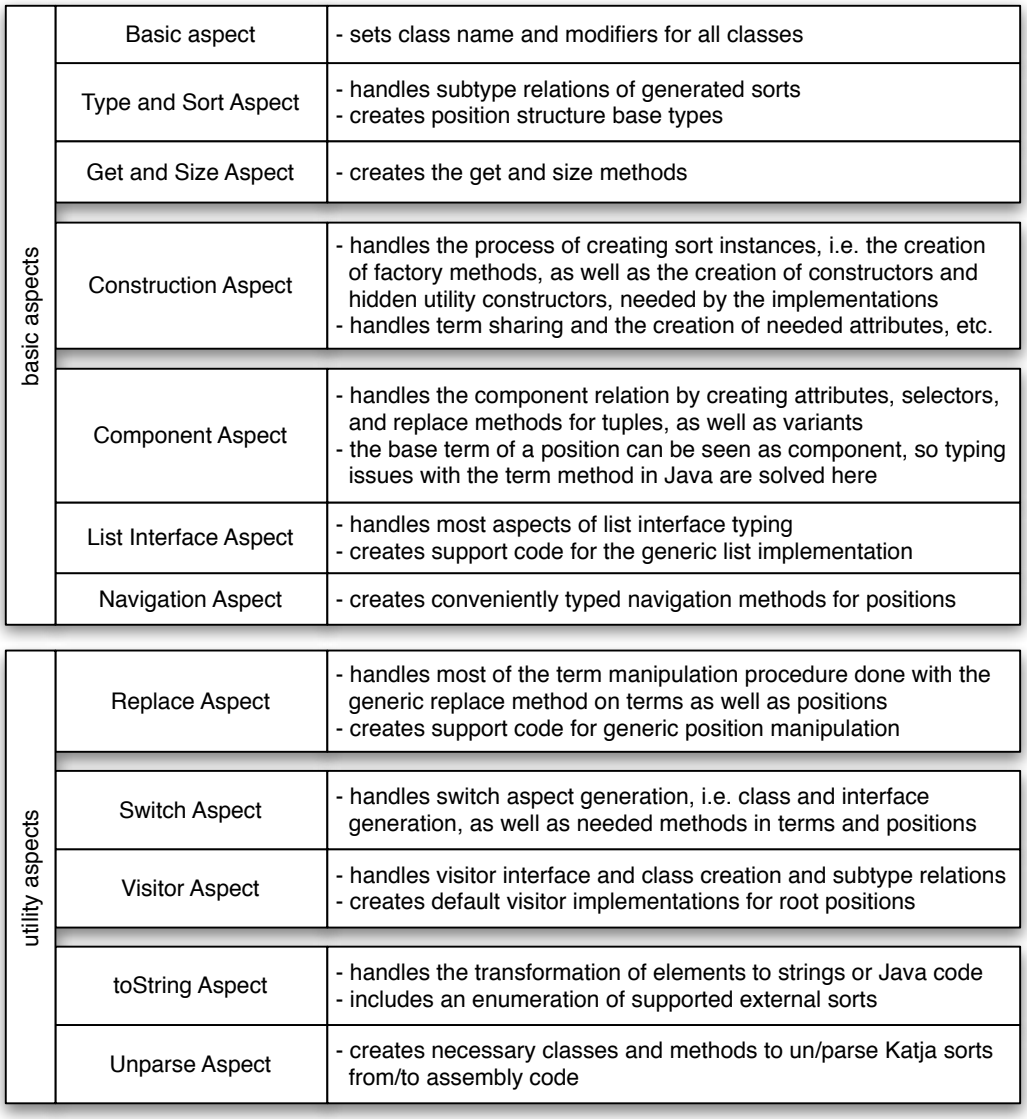

Figure 6: Generation Aspects of the Java Backend

3.3 Code Output

Creating hundreds of source files, as the result of a Katja run, did not turn out to be a wise decision for several reasons.

- Version control systems operate on file basis, not on directories. A build process involving Katja, however, considered one input file on the one hand and a complete package as output on the other hand. So whenever sorts disappear from a specification or are added to it, the number of files changed and the version control system had to recognize those changes.
- Dependencies in the build process were also difficult to realize, as a directory representing a package is not easy to check to be older or newer than another artifact, especially due to the fact that it is unknown in general, which or how many files belong the a given package.
- Cleaning up a target destination had to be done very carefully and was impossible whenever a generated package was mixed with other artifacts of the same build process.

I decided to generate jar files instead of directories, containing the generated sources, as well as the compiled class files. The output of translating one specification is therefore exactly one jar file, named after the specification name.

This solves all of the above problems and additionally frees the user of the need to compile the output himself. Katja uses the sun compiler to compile the generated code, which is not supplied with Katja at the present state, so it has to be present on a system running Katja.

As Katja experienced several issues with current compilers, this procedure would also make it possible for Katja to bring its own compiler to compile the generated code, known to work with the used feature set of Java and the language level.

To compile a generated specification package Katja needs the `katja.common.jar`, which is supplied by itself, as well as a link to each imported specifications jar file. The `katja.common.jar` can also be deployed into the destination directory, so the user has everything he needs to use the generated packages.

4 Redesigning the Generated Code

Many features were present in the initial version of Katja and many more were added later on. In many cases those features were not integrated in the

generated code, but added separately as generated classes. Other features were still experimental or awkwardly integrated.

The complete redesign of the Java backend, described in Section 3, offered the possibility to reevaluate all features separately and choose a concise and powerful set, which should integrate flawlessly with the generated code.

The following sections will investigate features or past design decisions and discusses their feasibility for the new Katja system.

4.1 Deprecated Methods

As a successor of the MAX system, Katja was designed to be fully compatible to max specifications and therefore had to offer many methods which could be used there. But Katja should also be usable for common Java programmers without difficulty. This resulted in some methods to exist with several names, like `get` and `subterm` or `size` and `numSubterms`.

It became apparent, that such aliases were not only confusing, but had several downsides:

- The `eq` method had the only advantage of being slightly better typed than `equals`, but was rarely, if ever, used by Java developers. The implementation had some problems in the beginning, as control flow passed several times from one version of this functionality to the other.
- Position sorts needed two additional methods, named `child` and `numChildren` to realize `get` and `size` functionality, as the *term* versions would not be appropriately named.
- Code avoiding `get` and `size`, by using the old names of the methods, was tedious to migrate from the usage of terms to positions. As this is a common procedure for developers starting to use Katja, a migration should be as easy as possible.
- The `KatjaElement` interface did not have any methods so far, but as all elements could share the generic selector `get` and the `size` method, I moved them up. This was, of course, not possible for the special term and position methods.

So I decided to no longer support specialized method aliases and keep the often shorter Java inspired versions. So `subterm` and `child` were removed in favor of `get`, `numSubterms` and `numChildren` became `size`, and `eq` was removed, so only `equals` remains for this purpose.

4.2 Constructors

Term construction was done by either the `instance` methods or by using the term factory, which was optional in Katja. Applications most often used the term factory, as it could be imported statically, which proved quite convenient for the developer. Constructors of implementation classes themselves could not be used. I think the design of term creation using factories was fine and considering the changes of Section 4.3, factories are now mandatory. The term factory is now integrated into the specification class, so there is no `TermFactory` anymore.

Following this design, positions can now be created using root position constructors, taking one root term as argument. The former possibility of creating root positions with the `pos` method is discontinued, as such a method cannot be supplied in general anymore. Position structures can be declared on imported root sorts, which prevents the generation of such a method. As it is also allowed to create more than one position structure with the same root sort, but different suffix, this decision seems logical and justified.

The general concept of term and position creation therefore became very simple:

- The constructors of all sorts defined by a specification are located in the specification class.
- There is no difference between terms and positions when it comes to their creation.
- As implementation class constructors are now only used within the factory situated in the same package, they can be defined as *package local* to prevent access by the user.

As usual, there are variable argument constructors for lists, allowing the creation of empty lists as well as lists with an arbitrary number of initial elements. All constructors are checked against null arguments, yielding exceptions as explained in 4.7.

4.2.1 Variable Argument Tuple constructors

One of the most convenient features, introduced to Katja lately, are the variable argument tuple constructors. Whenever there is a tuple definition, ending with a list sort, I generate two constructor versions:

```
A ( B b, C c, D d)
D * E
```

leads to

```
A(B b, C c, D d)
A(B b, C c, E... d)
```

In many cases such lists at the end of tuples are initially empty, or consist of only one or two elements. Many examples using this feature can be found in Section 3.1, where it was used many times to simplify the formulation of the Java model.

As far as Java is concerned, the introduction of additional constructors cannot lead to any problems. Even if the list sort is defined to be a list of itself, the two methods generated are separated by Java. In cases where exactly one element (or list) is given, Java selects the more specific method, which is the standard constructor.

To actually see the benefit of this feature, let's look at one of the examples, given at the end of Section 3.1.4. The very simple term

```
MGenericTypeDef(MJavaType("Tuple"+suffix), MTypeVariable("T",
    MJavaType("KatjaTuple", "katja.common.*")))
```

expands to

```
MGenericTypeDef(MJavaType("Tuple"+suffix, Strings()),
    MTypeVariables(MTypeVariable("T",
    MJavaType("KatjaTuple", Strings("katja.common.*")))));
```

The example also shows how optional parameters can be neglected altogether, when using the variable argument constructor.

4.3 Term Sharing

Term sharing offers some advantages, as well as some disadvantages. As far as runtime is concerned, there are two options:

1. Equality of terms and positions is calculated every single time `equals` is called.

2. Equality of terms is checked once at creation and is trivial afterwards. A new object is, however, compared to about $\log(n)$ terms, though many equality checks will break early.

Which alternative is better depends on the application, as it is vital to know, if many created terms share common parts or if most terms are completely different. To notice any difference at all, there have to be created lots

of terms in any case. As comparing two terms is not considered to be an unusual operation, the second alternative seems nice to have.

Overriding the `equals` method in the generated code also means overriding `hashCode`, both methods are far from trivial in the context of Katja. By doing the check only once in the term factory, I do not need to override hashcode and have to implement a comparator only.

Considering space complexity the options are:

1. Returning a term or position as often as a constructor is called by the user, but each object is garbage collected automatically when no more referenced.

2. Returning only one object to the user for each constructor call constructing the same term or position, but the object is never garbage collected, even if no more referenced by the user.

3. Returning only one object to the user for each constructor call constructing the same term or position, which is garbage collected when no more referenced by the user.

The second and third option will have a tremendous effect in situations where many large objects are created, sharing many parts. The third option goes even further, tuning situations in which many terms are created and then discarded as the system enters later phases.

The reader should keep in mind, that time and space complexity are interwind in general. Using more space and accessing all of it at the same time often results in much longer runtime. With those considerations the second option yields the largest benefit as far as execution time is concerned, whereas the third option is vital for continually running systems.

There are additional data structures needed to realize term sharing, which also need space. In the worst case, where no term can be shared at all, those data structures will consume more space than was saved, but those cases are not considered to be common and the additional space complexity is linear.

With the above considerations I see it justified to introduce term sharing as non-optional feature, as it has several implications to the implementation of Katja. The realization of term sharing, however, is driven by simplicity.

I use a `TreeMap` together with a special `Comparator` for the sharing. This solves the two main problems:

- **term identity** The user calls constructors of a certain *type*, together with a *list of arguments*. To know if such a term already exists, I need to address a data structure with multiple keys, for lists even arbitrarily

many. Such maps do not exist for Java and writing one will result in a complex and error prone data structure. So all this information has to be wrapped in an object, which is eventually discarded after the comparison to existing elements. Therefore Katja just creates the term requested and checks if it exists in the `TreeMap` using the special comparator.
Using numbers or other comparable data structures to create a unique identifier for each term is impossible, because there is an infinite number of possible terms. Using strings or the like will not solve the general problem of creating an additional object for comparison and it will not be significantly faster.

- **term retrieval** To actually find a matching term or exclude all existing ones without checking them all, we need to order them, so we can use binary search. This is also done by the comparator, as a side product of getting the result of two terms not being equal.
  Using hashs to utilize the constant complexity is a difficult task, as there is no trivial hash code associated with terms. Calculating a good one takes the same amount of time as a *complete, recursive* comparison, which is not needed when ordering different terms in a tree, as comparison stops on the component level of a term, the latest. If at least one object with the same hash code exists in the hash, object comparisons have to be done like in the case of the `TreeMap`.

This implementation results in term objects being created on each constructor call and all but the first for each term immediately being able to garbage collect. As position constructors always have only one parameter, positions can be shared more efficiently.

There is a simple map from possible root terms to the root position, so it is only needed to check if the map already has an entry present for a given term. All other positions in a given structure must be created by invoking selectors, which always produces exactly one new position.

This implementation realizes option two of the space complexity considerations, as references to terms and positions are never lost in the term factory. To realize this feature I need to use weak references in Java, like done in a weak hash map. The `WeakHashMap` class can obviously not be used, as it works on hash codes and the `equals` method, which are meant to be different on semantically equal terms by design.

Using the `WeakReference` class will not work with tree data structures present in the API. So to allow option three to work, I need to implement a special tree data structure, which is aware of keys to be deleted without its

knowledge. This is possible, of course, but will be a difficult and error prone task. It will be added to the Katja system when the need arises.

4.4 Sortints

The `sortint`s, together with `KatjaSort`s, where introduced to Katja to allow variant sorts to be distinguished using `switch` statements and to have runtime tokens for sorts to check against. The latter can be done using `class` objects, so there was no need to create special `KatjaSort` objects.

It became apparent, that sortints where leading to many problems and that the distinguishing cases of variants could be done with other techniques. There always was the inherent problem of assigning an integral number to generated sorts, which did not collide with separately generated sorts. This cannot be solved in general and already lead to numerous problems.

Using `switch` to distinguish variant sorts and therefore using integral values, also has the downside of not being statically save. It was easy to forget cases, especially after changing a specification. If a sort was removed from a variant but not removed from the specification, it wasn't detected as well, but resulted in a dead case.

To solve the problem in a more convenient and save way, switch classes were introduced to Katja, which are therefore also present in the new Katja system. Section 4.5 describes this feature. `Sortint`s and `KatjaSorts` are therefore completely removed from the Katja system and the method `is` has to be replaced by `instance of` accordingly.

4.5 Switches

The feature to distinguish sorts of a variant in Java code is realized by so called switch classes. They were realized using the generated sortints and used `switch` statements in their implementation. With sortints removed I had to implement switch classes in another, even faster way.

The idea behind switch classes is to solve the dynamic problem of determining a property of an object. This can efficiently be done using dynamic binding of method calls, like it is done in the visitor pattern. The `accept` method of the visitor pattern is named `Switch` for convenience, and it has to start with a capital letter, as `switch` is a keyword in Java.

Katja therefore generates an interface for each variant sort, which consists of all needed case methods. The interface is an inner class of the variant interface and is called `Switch`, it will be used qualified with a variant sort name in most cases. The variant interface itself contains the `Switch` method, which guarantees the method to be present in all variant cases. Those cases

implement the `Switch` method by invoking their associated case method on the given switch class implementation.

An additional design goal was to allow switch class implementations of more general variants to be applicable to subvariants too. To completely realize this feature, all `Switch` interfaces have to be subtypes of their variant case sort's `Switch` interfaces. This is possible in Java, as interfaces can have arbitrary many subtypes and the resulting type hierarchy is the inverted variant sort hierarchy and therefore acyclic, as shown in Figure 7.

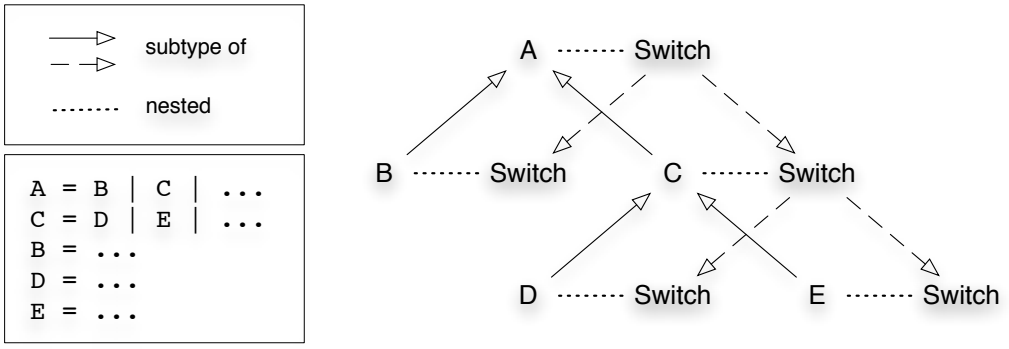

Figure 7: An Example Switch Class Hierarchy

The reader should note, that without the subtype relations between `Switch` classes it would have been possible to use more general switch class implementations on specific variants anyways, since those are subtypes of the generic variant and inherit the appropriate `Switch` method. The important feature is to allow switch class implementations to be *passed* through interfaces and in general through a complete software system, without adding needed super types by hand.

The usage of switch classes is a bit different to what it was before, as a switch class is now passed to a term or position, instead of the other way round. An example of the appropriate use of a switch class is:

```
result = term.Switch(new SomeSort.Switch<RType, IOException>() {
    RType CaseMySort(MySort term) throws IOException { ... }
    ...
});
```

As switches are most often implemented by anonymous classes, it is an advantage of this notation that the term reference, which is actually switched, is situated before the anonymous class, rather then disappearing at the end of the switch implementation.

Writing the name of a case sort in the method name and again as parameter type is redundant, but necessary at the moment. My first implementation defined all case methods using the name `Case`, as the type of the case was present in the parameters. The semantics of Java allows this and there is no theoretical problem with this at all. Present compilers, however, slowed down significantly, when they encountered a large number of overloaded methods, or refused to work at all. See Section 4.14 for more details.

Switch classes have two generic type parameters, as they should yield either a value or an exception as the result of a switch. If no return type is needed `Object` can be used, together with `return null` statements in each case. If no special exception is needed or only unchecked exceptions are thrown, the second type parameter should be set to either `RuntimeException` or the much shorter exception `NE`, defined in `katja.common`. The latter can't be instantiated and its name is the short version of "no exception". To manipulate data structures of the context from within the anonymous switch implementation, these structures need to be declared as `final`.

The presented solution statically detects missing cases of a variant and can also detect dead cases, if the `@override` annotation is used for all cases, together with a tool giving a warning whenever such an annotation is misplaced.

The `Switch` method of all variants are implemented in the list and tuple sorts, by calling their appropriate case method on the supplied switch class. There is one overloaded `Switch` method for each variant a sort is part of.

4.6 Visitors

Visitors are used to compute data for complete terms, the most obvious examples being term unparser or interpreters, which need to visit all nodes of a term and need to work on nodes with an exactly typed object reference. Without visitors a traversal of a term had to be done by hand, identifying and casting each node when passed.

On position sorts the traversal can be easily done using the `pre-` and `postOrder` methods, but the problem to identify and cast elements remains. Those methods are therefore used in cases were only few different nodes have to be found and others are of no interest.

Visitors were implemented using the visitor pattern, using dynamic method binding to distinguish variant components of lists and terms. There was exactly one visitor for a specification, containing a visit method for each sort of the specification. This resulted in many visit cases being empty for most of the visitors and in tremendous efforts to adjust all visitor implementations, whenever a sort was added to or removed from a specification.

Like it was done for the switch classes, it should also be possible to apply visitors to terms holding a subset of needed visit cases and to pass such visitors through interfaces. As each sort in a specification potentially needs it's own set of visit cases, the problem has a larger scale then the switch class problem.

Using the visitor pattern to implement visitors for arbitrary sorts would result in a large amount of `accept` methods in all sorts and is impossible in general, as sorts can be imported by other specifications, where they can appear in tuples or lists. Specification boundaries were in fact one of the major problems of the visitor implementation.

Taking a look at the general problem we need to solve, these are the important properties:

- There should be a visitor type for each sort, holding methods for all needed visit cases.
- These methods are used to get a statically typed reference to a node and holding the code associated for it, so the general task solved by the visitor is broken down to smaller problems.
- When visiting a tuple or list, all children are known statically, so appropriate methods can be called from the developer to visit them. Methods visiting elements of a variant sort, however, should not be implemented by the developer, but they should be supplied, calling the appropriate method dynamically.

So there is no need to apply the visitor pattern and utilize dynamic method binding, except for variant sort components. But to solve these we can use the switch classes, which were introduced for this exact reason.

My visitor implementation therefore works as follows:

- Each sort has a nested interface named `VisitorType`, containing all needed visit case sorts. This interface should be used to pass visitor implementations around in a system, as it is the precise definition of a visitor which is needed to visit an element of the current sort.
- Each sort has a nested abstract class named `Visitor`, implementing all variant sort visit methods, using anonymous switch class implementations. This class should be used as base class for each visitor implementation, as it takes care of the variant cases and implements the `VisitorType` interfaces needed to pass it through interfaces accepting only visitors of an applicable kind.

- A visitor can then trivially be used to visit all elements appearing in its `VisitorType` interface, as there is no accept mechanism prohibiting its use, but the user invokes the visitor on an element.

The usage of visitor implementations is therefore a bit different to what it was before, an example of the appropriate use of a visitor is:

```
visitor.visitMyTerm(myTerm);
```

and the implementation of a visit method could be:

```
void visitMyTerm(MyTerm term) {
    System.out.println("MyTerm (");
    visitComponentA(term.left());
    System.out.println(", ");
    visitComponentB(term.right());
    System.out.println(")");
}
```

Again there is the need to specify the sort of a visit case twice, in the name of the method and as parameter type. Section 4.14 describes the problem in detail. It might be possible in the future to switch back to a set of overloaded `visit` methods, which is considered much more convenient by many developers.

This design enables us to statically check for the absence of needed visit methods, but demand only those really needed for a given sort. Import boundaries of specifications impose no problem at all, as imported sorts can be treated like those from the same specification, generating all needed visit case sorts if necessary, or implementing the imported `VisitorType` interface.

The syntax used to visit components of a list or tuple is much more convenient than the `accept` syntax of the visitor pattern; the concept of dynamic method binding is not needed anymore.

4.6.1 Default Implementations

Generating a rich set of interfaces, to work with visitors in Katja, is already a huge feature. When implementing a visitor the developer can benefit from static guarantees, reminding him of neglected cases, for example. In some case, however, the developer wants to implement only half the needed cases of a sort, still too many to use the `pre-` and `postOrder` position methods instead.

For these cases Katja could supply default visitor implementations, visiting components from left to right. The developer can then implement only

those methods necessary and even call the super methods to continue the visit, which is most convenient for lists.

Sharing implementations in Java is not that easy, however, so it is in general necessary to supply code implementing each method of each abstract `Visitor` class. The complexity of such a code generation is completely out of proportion compared to other features of the Katja system.

I therefore decided to generate such default implementations for positions sorts only, which are the root of at least one position structure. Such a default visitor is in many cases the only one needed and it still can be used to implement visitors for other position sorts of that structure.

4.6.2 Visitor Subtype Relations

The needed subtype relations for the `VisitorType` interfaces are comparable to those shown for switch classes in Figure 7. The main differences, however, are the presence of cycles in the component relation and the much higher number of involved sorts, even including imported sorts from other specifications.

So it's in general impossible to follow the same approach as for switch classes; a more sophisticated type hierarchy is needed. Figure 8 shows a small sample specification and the relations of sorts.

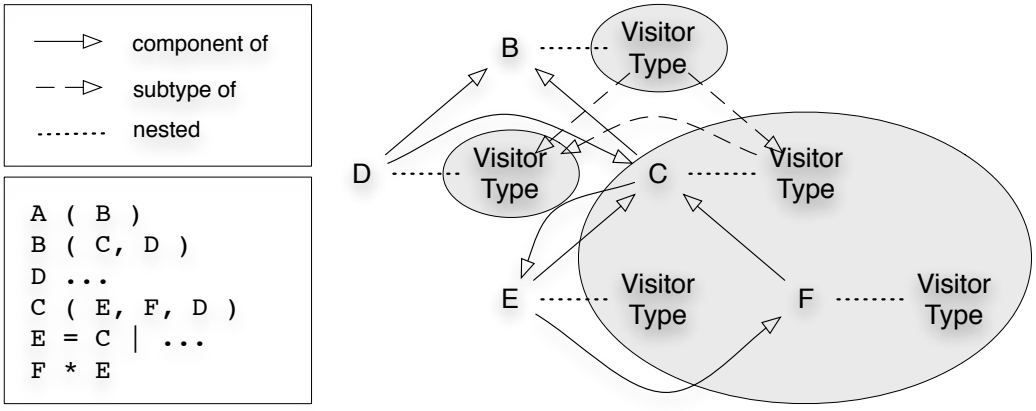

Figure 8: An Example Visitor Type Hierarchy

The "component of" relation is the key to the complexity of the problem and to the realization in Java. In the case of visitors we need to widen the definition of "component" a bit, so a component is a sort which is needed in a visitor defined to be able to visit all possible terms of a given sort, i.e.

- A sort is always a component of itself, in that sense.
- All cases of a variant are components of the variant sort.
- The lists element sort is a component of the list. The component sorts of a tuple are components of the tuple sort, obviously.
- The relation is transitive.

We can now define an equivalence relation of sorts, given a fixed root sort. Two sorts are said to be equivalent if and only if they are both components of each other, in the sense defined above. This relation is obviously reflexive and transitive, as well as symmetric by definition. Figure 8 marks the present equivalence classes of the `VisitorType` hierarchy with circles.

The consequence of the calculation of these equivalence classes are:

- All `VisitorType` interfaces in one equivalence class have the same needed visit cases, but can't all be subtypes of each other, as this would produce cyclic dependencies.
- It is sufficient as well as safe for the `VisitorType` interface of a sort to be subtype of all interfaces not in the same equivalence class, but in that of a direct component sort.
- A `Visitor` implementing the `VisitorType` interface of a sort can technically be used to visit terms of all other sorts in the corresponding equivalence class, as all needed visit cases are present, and no subtyping is needed to apply the visitor.
- The abstract `Visitor` classes can then be subtypes of all `VisitorType` interfaces of the same equivalence class as the own `VisitorType` interface. This makes the type hierarchy perfect and allows arbitrary, sufficient visitor implementations to be passed through interfaces.

The last point, however, cannot be realized at the moment, as compilers slow down too much. It is therefore not possible at the moment to pass a visitor implementation through an interface excepting a `VisitorType` of the same equivalence class, without adding this `VisitorType` to the list of implemented interfaces manually. Passing visitors of other equivalence classes, which pose a superset of the needed cases, can be done without explicit interaction and should be the by far more common case.

Implementation sharing of the abstract `Visitor` classes cannot be done in an efficient way in general. The visitor type hierarchy is for the most part

an inverted component hierarchy, so to share all implementations in the best way possible we would need multiple inheritance.

It is possible, however, to share implementations along one path of the type hierarchy, which can be optimized by selecting the `Visitor` with the most implemented variants from all available components. This feature had to be turned off as well, as it lead to some of the compiler problems described in Section 4.14.

4.6.3 Exceptions in Visitors and Switches

All visitor classes and interfaces have one generic parameter, defining the kinds of exceptions that can be thrown in the `visit` methods. As it was done with the switch classes, this exception type can be `RuntimeException` or `NE`, if there is no exception needed for the visitor.

Most unparser, for example, will write their output to some remote or potentially unstable location and therefore have to deal with various exceptions in every `visit` method. Without the generic parameter this would result in try-catch blocks being necessary in each `visit` method and to wrap exceptions in `RuntimeException`s or `Error`s.

Neglecting the generic parameter and defining `visit` methods to always throw checked exceptions, would result in the need to surround each visitor call with a try-catch block, no matter if some checked exception is thrown at all.

I considered generating additional, exception free, versions of switches and visitors, for which it would not be necessary to use the `NE` exception type. As there are no type aliases in Java, it would be necessary to define those new types as subtypes of the more generic ones.

This, however, causes several problems, related to the absence of multiple inheritance in Java. I would either loose the relation of generic and non-generic visitors or could not use inheritance at all, so the additional hierarchy of visitors would duplicate the source code needed to implement the visitors.

Both options are unacceptable, so I judge the necessity to always specify the generic types to be a minor inconvenience. Most of the time developers will use a wildcarded import statement for the `katja.common` package, so the `NE` type can be used unqualified.

4.7 Exceptions and Errors

When talking about exception and error handling in Katja, you need to separate four different areas:

1. Program errors in the Katja front- or backend.

2. Usage errors detected in the front- or backend.

3. Errors in the generated code or in the common library.

4. Usage errors detected by generated or common code.

Category one and two are taken care of in Section 2.5, all other categories represent errors discovered at runtime of projects using Katja, even Katja itself. The intended use of Katja should only yield unchecked exceptions, as the idea of an "unchecked exception" matches best in that case: Category three is supposed to never occur and truly denotes *unexpected exceptional* behavior.

The last category of errors, created by faulty use of the generated interface, is an unexpected behavior too, though the mechanism can be explicitly used by the developer. Nevertheless all exceptions thrown are runtime exceptions or errors, as I don't see any categories of errors to be always anticipated by the user and therefore be annotated as checked exception in the interfaces.

Programming errors can result in various runtime exceptions or errors to be thrown, including those explicitly thrown by Katja developers, which are either `IllegalStateException`s or `AssertionError`.

Detected usage errors of the generated interface will always throw a runtime exception taken from the Java standard API. Those include, but are not limited to:

- `IllegalArgumentException`
- `IndexOutOfBoundsException`
- `UnsupportedOperationException`

Note that unintended usage of methods need not always result in an exception, but can result in a `null` value returned, for example. This is made by design and will be discussed in the following sections, whenever the interface of sorts is introduced.

One design decision was not to introduce special Katja exceptions to throw, as there would have been no way to make them both subtype of their API counterpart and a generic `KatjaException`. Having such a generic exception would have enabled the user to

- Catch Katja exceptions explicitly separated from exceptions thrown by other parts of the project.
- Catch all Katja exceptions in one `catch` statement.

- Catch all exceptions thrown by Katja with exceptions known from the Java API, so Katja integrates smoothly into the project.

All options at once are not possible as all subtypes of exceptions, and even `Throwable`, have to be classes, so an exception can only be subtype of exactly one type of exception. In the end it had to be decided whether exceptions are in any way related to the standard API exceptions *or* to a generic Katja exception. I chose the former possibility, which also makes the introduction of special Katja exceptions obsolete.

4.8 List Interface and Sets

While working with Katja generated lists, it became apparent that many more methods should be offered in the interface for convenience reasons. Users tended to use small code fragments over and over for a specific task or had to define their own enhancement of the interface.

Such methods include obvious things like reverting a list or selecting a sublist, but also includes more sophisticated methods. Especially the list position sort interface offered no methods at all, as position were considered to be fixed. With the addition of manipulation functions to the general position interface this view had to be revised.

Katja offered no notion of sets, which resulted in either the usage of Java sets or the aforementioned creation of small code fragments used over and over. To add a term only once to a list you had to write:

```
if(!list.contains(term)) result = list.add(term)
```

with `list` as well as `term` being arbitrary complex expressions, both even occurring twice in the code fragment. The new Katja system therefore offers a set of operations to work with Katja lists as sets. Figure 9 shows the operations defined on lists and defines groups to which they belong. Many operations will be known to the user from the old Katja system, though some facets of their behavior might have changed.

4.8.1 Set Operations

To work with Katja lists as sets, Katja offers the method `toSet` as primary tool. The result of this operation is a potentially different Katja list having the set property, i.e. each element of the starting list appears exactly once in the result list.

There is a huge degree of freedom which list representation of the set can be returned, in theory. First implementations of this method used Java's

	Operation		Intention and Result

list creation manipulation	`add`
	`appBack`
	`appFront`
		result is a new list	- Adds the element to the list, no matter if the element already occurs in the list or how often.
			- `Add` and `appBack` append at the end, `appFront` at the beginning.

	`addAll` `conc`		- Adds all elements of the given list to the end of the the list, while maintaining the order of both list. - `Conc` is an alias for `addAll` (as well as `appBack` for add).
	`remove` `removeAll`		- `Remove` removes the leftmost occurrence of an element from the list, an exception is thrown if there is none. - `RemoveAll` removes all occurrences of all elements from the list.
	`reverse`		- Reverses the order of elements in the list. - The operation is seen as a manipulating function, as no elements are selected, but it is the effect that is intended by the user.
element retrieval selection	`front` `back`		- Selects all elements of the list, but the last or first, respectively, maintaining the order of the remaining elements. - The operations are undefined on empty lists and throw exceptions.
	`sublist`		- Returns a subsequence of elements in order, potentially wrapping over the last element to the start of the list, so the second index can be smaller than the first, indexing is also allowed like with `get`.
	`first` `last`	element	- They select only the first or last element of the list and are therefore complementary to `back` and `front`. - Both are undefined for the empty list.
	`get`		- Selects one element of the list, specified by index, starting at 0. - So `get(size())` is out of bounds, negative numbers index from the end of the list, up to `-size()`, which is the 0 element.
set operations on lists manipulation	`toSet`	set represented as list	- The set version of the list, which is constructed by deleting duplicate items from the end of the list. The operation thereby preserves the order of elements the best possible.
	`setAdd` `setUnion`		- The operations add one or more elements to the list exactly once, appending them to the end of the list representation if new. - A list is transformed to a set first, if necessary.
	`setRemove` `setWithout`		- The operations remove one or more elements from the set. - It is not considered to be an error if a non-existing element is tried to be removed with `setRemove`.
	`setInter-` `section`		- Returns the set representation of the list, with all elements of the given list removed. - The order is preserved the best possible, taken from the list.
	`setEquals`	boolean	- As there are only operations working with lists as sets, there are no real sets, but they are all represented as lists. To compare two lists with respect to their set property we need `setEquals`.

	`isEmpty`
	`contains`
	`containsAll`
			- Checks if a list is empty or contains one or more elements.
			- `ContainsAll` is only true if the list contains all elements, obviously.

	`toArray`	array	- Returns the Java array representation of the list of elements represented by the Katja list. - Can be used as connector of Katja lists to the Java list API.

Figure 9: The Operations of the Katja List Interface

hash API, so the result changed from execution to execution. Besides being a correct behavior for a `toSet` method, this was extremely inconvenient.

Code generators produced unstable output, swapping the order of attributes or methods and such, resulting in the version control system `svn` to find changes in source files were nothing was changed. Writing tests for programs using this feature of Katja would have been tedious too, as the result wasn't fixed.

The motivation of *set* methods was given by a small example in which an element should only be added once to a list. This exact operation, i.e. appending an element to the end of a list, if it did not appear in the list before, would be unavailable to the developer, if `toSet` was implemented with no additional guarantees.

That's why I decided to guarantee the retrieval of exactly one possible representative of a set, which is the original list with all duplicates removed from the end of the list, i.e. elements in the list remain ordered by first appearance from the beginning. This decision seems even more valid considering the fact that there are no sets in Katja, but they are backed up on lists, so operations should discard the fewest number of properties possible of a list, while guaranteeing the set property.

All other set operations, as seen in Figure 9, are implemented with regard to this design, so they try to maintain the order of the list the operation is invoked on. The special method `setEquals` is needed to compare two lists representing sets, as they are not really sets in Katja.

The `toSet` operation seems to be expensive at first and this is certainly true for the conversion of a long list. In Katja, however, each list maintains a reference to its set version, which is uninitialized at first and can end up being `this`. Subsequent calls of set operations on a list are therefore much faster and in many cases Katja knows the result of an operation to maintain the set property. In these cases the newly constructed list does not need to calculate its set version at any time. Consequent usage of the set interface only will therefore not result in huge drawbacks in execution time.

4.8.2 Operations on List Positions

When considering the introduction of methods to the list position interface, their semantics have to be clear. There are several possible design decisions:

- Operations returning a new list could return a new list position, representing the new list in the context of the old one. Those operations would then be convenience methods for the use of `replace` and modify the underlying context of a position.

- Operations returning a new list could return a list of element positions, representing all positions of elements that were selected by the operation. Those operations would thereby leave the list position interface, as they return a term list containing positions.
- Operations returning a new list can also be left out, like it was done before.
- Operations returning something else impose no problem in general, but it often has to be decided what the parameters of such operations are; positions or terms?

I decided that methods like `add`, `conc` or `remove` are clearly *manipulating* operations, as are the new *set* operations on lists, which are often used to construct sets from others. These two groups of operations will therefore manipulate the underlying root term of a position and return a position from a new structure.

Such methods are extremely convenient to work with, as they allow very concise algorithms to be defined to manipulate positions. Consider a small specification defining a tuple with two list components:

```
root A Pos
A ( B left, B right )
B * Integer
```

When having a `B` list as the left component in an `A` term and there is an Integer to be added to the list, it had to be done like this:

```
A newA = oldA.replaceLeft(oldA.left().add(17))
```

Using positions allowed to reference the list directly, but did not shorten the operation at all:

```
BPos newB = oldB.replace(oldB.term().add(17))
```

With the added `add` method, from the group of manipulating methods, it can now be done in a much more convenient way:

```
BPos newB = oldB.add(17)
```

The ease of use of the term interface is thereby combined with the powerful navigation interface of positions, ultimately allowing positions to be used in any situation, without being forced to drop back to terms in certain situations.

Another group of operations, however, are not considered to be of a manipulating kind. Calling `front` on a list is first and foremost a selection being made, to get all elements of the list besides the last. The operation `front` is furthermore some kind of dual to the operation `last`, which is obviously not manipulating any list.

Selection on positions is a problem though, as there are no list position for sublists already in the structure. The result of such operations can therefore not be a list position (if one does not consider selection methods to be manipulating). I decided to add the `toList` method to the list position interface, which returns all element positions of the list position as term list. This list is of the general `KatjaList` type and provided by an anonymous implementation.

With such an operation all selection operations can be done on the term list view of a list position and no such operations are needed on positions in general. There is, however, the option to introduce such methods for convenience reasons and as `first` and `last` are already provided I decided to implement the dual methods `front` and `back`.

They simply forward the call to the result of the `toList` method and return the result, thereby leaving the list position interface, so the user can then work with all selection methods in the term list interface. The `sublist` operation is not provided for now, but can of course be used after calling the `toList` method on the list position interface.

The user can now split list positions in recursive function calls, using the `first` and `back` or `last` and `front` pairs of methods, without creating new positions or invalidating the theory of position structures. But he can also use positions to navigate over terms and manipulate them in a convenient way, using manipulation methods.

Manipulating methods always take term sorts as argument, as the operations aim on the manipulation of the base terms, while selection methods do not take elements at all. In the case of the `contains` and `containsAll` methods I decided to stay on the position level, so `contains` will simply check if the given position is a child of `this`, as will `containsAll` do for all elements given. Expecting a list position as an argument for `containsAll` does not make any sense at all, as the method would then be the same as `equals`.

As it was done with the method `termEquals` on general positions, I introduced `termContains` and `termContainsAll` as convenience methods, to check if the base terms of given positions are contained in the base term of the list.

4.8.3 Exceptional Behavior

The list interface on terms as well as positions will never return the `null` reference. This was done by the old Katja system, as the MAX system used a `nil` value to indicate undefined behavior of operations.

Whenever a method is used with wrong parameters or in an unapplicable situation, Katja will throw an exception instead. Some operations, which are used "wrongly" cause they have no effect on the current list, will simply be the identity in that case.

Invoking `get`, `replace` or `sublist` with too large or too small index values will result in an `IndexOutOfBoundsException`. Calling the `remove` method on lists with an element not present in the list will result in an `IllegalArgumentException`, which is not the case for `setRemove`, as set operations are used with different intentions. Calling `front`, `first`, `back` or `last` on the empty list is not defined and will result in an `UnsupportedOperationException`.

Many operations, like `setAdd`, `setRemove` or `removeAll` may very well return the same list they are invoked on, without calling it an error.

The `null` value is also not accepted by any operation, as all operations construct new result lists and Katja does not allow `null` to be passed to constructors in general. There is no such thing as `nil`, as Katja uses the Java exception mechanism for undefined operations on lists. If an exception is thrown from the list interface, it is always a usage error by the user, which is truly considered *exceptional*, as it could have been prevented with only small effort. Note that the position interface follows another concept, as described in Section 4.10

4.9 List Subtyping

The immutability of Katja terms offers some interesting possibilities, one being the option to generate list sorts as subtypes of each other. As there is no contravariance needed for immutable types, a list sort could be subtype of another, whenever its element sort is a subtype of the element sort of the other.

This is realized in Java for all array types, though Java arrays are not immutable. Java therefore needs runtime checks on write access to arrays, to catch this class of errors.

The generated Katja list interfaces could extend each other, resulting in the implementation classes to offer many methods in various versions. At first glance this sounds doable and would be a nice feature to have. Methods returning a list sort could be overridden in subtypes to return a specialized

list sort and many lists could be passed through interfaces accepting more general lists.

```
A = ...
B = A | ...
AList * A
BList * B
```

The interface of `AList`, for example, would then have to offer an `add(B e)` method, which returns a `BList`, to be fully compatible with its super type `BList`. As long as there is only one list type for each sort this is no problem at all.

Whenever the user defines two list types of the same element sort, these list sorts would need to extend each other, which is not possible in Java, as there are no type aliases. Subtypes of both list sorts could extend both of them, but could only return one of the list types in their interfaces, which would be sufficient only if they are equal.

Possible solutions would be:

- There can only be one list sort declared for each sort, which is checked and enforced by Katja. This, however, would cause major inconveniences for the user, as he would have to live with the name given to it by other developers in other specifications. Changes in specifications, like the addition of a list sort, would also result in incompatibilities with other specifications importing it. So this option is clearly out of question, as the price would be too high.

- There can be exactly one list sort for each sort, which is defined and generated automatically with a given naming scheme. This option is much more appealing, as list sorts themselves are often needed as helper sorts only and have to be added later on. Many specifications hold complete sections consisting of list definitions only, so they are not cluttering up the view of the core of the specification. An automated naming scheme, however, would cause troubles with naming conflicts and developers would no longer be able to make list names more expressive, like calling a list of *name-value pairs* an *environment*. Additionally, many list types would be generated in vain, as they are never needed by the user, though this could be avoided by requiring the user to mark sorts he wishes to have a list sort of.

It is also necessary to consider import boundaries of specifications, to supply such a feature. List sorts already defined in other specifications cannot

be subtypes of newly defined lists. But as subtype relations between lists follow those of variants, this is not a major problem and can't happen with the latter idea of avoiding multiple list sorts, for example.

I consider the price for implementing such a feature as too high to change Katja in this way. Passing a list through an interface accepting a more general type only, can be accomplished by creating a list of that type and adding all elements, as long as there is a convenient method to do so. Methods, of the Katja list interface, taking lists as argument are therefore adjusted to also accept lists of the elements subtypes. The interface in general was revised to be most flexible with regard to parameter and return types.

4.10 Position Navigation Interface

One of the major advantages of positions is the rich navigational interface, allowing to return to the upper context of a term in particular. These method are essentially the same as before, with only few added:

- The methods `parent`, `rsib`, `lsib`, `preOrder`, `postOrder` and `postOrderStart` are used to navigate one step in a specific direction.
- The methods `path`, `pathFrom`, `position` and `follow` allow to get, create and apply paths between positions.
- The method `term` drops the context, whereas `root` jumps to the position of the root term.

Most of these methods behave different than the list interface methods, as calling these methods in situations where the result may be undefined is not considered *exceptional*. Calling `rsib` or `postOrder`, for example, is done repeatedly, till there are no further positions to go to. Checking for a save execution of those methods would be tedious and creating methods to do it would unnecessarily slow down the execution of code using Katja.

All methods, except `follow`, therefore return `null` in *exceptional* cases, which has to be checked by the user. When following paths, however, we require the user to have an idea of what he is doing, like updating a position of a structure, after a replace operation has been done, so an exception is thrown in erroneous cases to indicate a semantic error.

4.11 Type Variables

The `katja.common` type hierarchy was always subject to change and many decisions had to be made. In [7], for example, I introduced the position

sort hierarchy and especially the differentiation between a leaf and a node position. It turned out there are even more separations to be made, in conjunction with the redesign of the number and kind of type variables position sorts have.

Most position sorts used to have `R` as only type variable, representing the root position sort, only list and leaf positions had an additional parameter `L` for their respective element sort. What about all the other sorts involved, like the term sort or those of the parent and siblings? Should there be type variables in all positions representing them?

I judge a type variable adequate for a sort, whenever two properties are fulfilled:

- The type represented by the variable has to be *essential* for the type to be defined. The element type of a list, for example, is essential in the definition of the list.

- The type represented by the variable has always to be *statically known* and has also to be *statically important* in many cases. Concepts like subtyping and dynamic binding are still available too and lots of interesting systems show dynamic variability as a key concept, so introducing static type variables in general does not make any sense.

Both properties hold for the type variable `R`, representing the root sort. It is essential to know if the context of an identifier is a complete program or a method body only and this information is always present statically, as positions can only be created from root terms or from other positions, in which case the root position type does not change.

The right sibling type of a position, however, might be essential for some position sorts, in which cases it might even be statically known, but this is no property holding for all position sorts and can't be true for interesting specifications at all.

The variable `L` again is essential to lists and leaves and is always statically known as well as important for the user working with lists and leaves.

It turns out at least one property does not hold for most variables possible, which were not present in the old Katja system, but they do hold both for the term type of a position. This type is obviously essential to a position, as it is the reason the position sort exists at all. It is also quite important for the user to know what the result of the `term` method is in a Java program.

This was done by generating better typed `term` methods in types, whenever they could be improved, ultimately being the most specific term type in list and tuple implementations. Why not make `T` a basic position sort type variable, so the term method is defined once and always returns the correct

type? There is only one reason preventing the flawless introduction of `T` to all position sorts and this one involves the variant sorts.

Generated Katja sorts should not have any type variables, as they are not necessary and are cumbersome to use. All sorts defined in a specification can be used from within Java by specifying the same single word. It is, however, not possible in Java to generate variants without type variable, when introducing `T` to `KatjaTermPos`.

Figure 10 shows three different possibilities of dealing with the inherited type variable `T` in variants, the type variable `R` is not shown to avoid the figure being unnecessarily complex. The first idea is to simply insert the term type of the variant position, which is the variant term sort, as it is done with all other generated sorts of the specification.

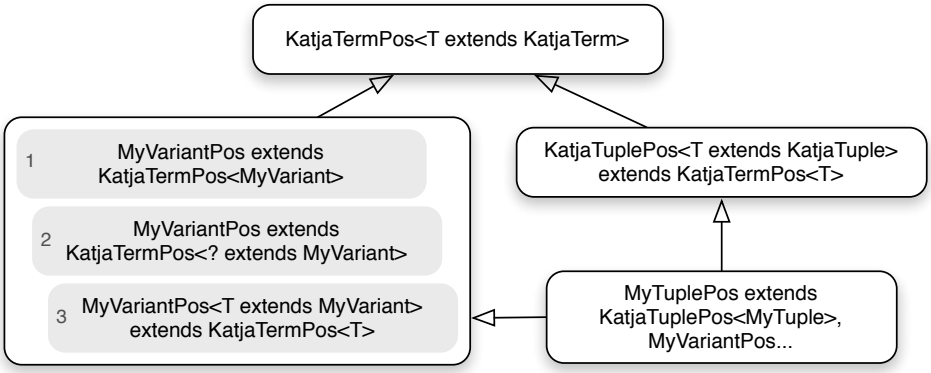

Figure 10: The Term Type Variable in Variant Positions

But variants are not leaf sorts, their term sort is not exactly known. The Java type system does not allow to extend the same interface with two different types inserted for a variable, as it would happen with `MyTuplePos`, extending `KatjaTermPos` twice with `MyTuple` as well as `MyVariant` inserted.

This is prohibited for a good reason, though it does not apply in the context of Katja, as elements are immutable. What we really want to express is that the term type of a variant position is some unknown subtype of the variant term type. So the second option in Figure 10 would solve the problem and would leave variants parameterless as well. Unfortunately wildcards are also not allowed in specified super types in Java.

What is left is to introduce a nonsensical type variable to all generated variant position sorts, which is generally used with a wildcard by the user, stating that the specific sort of the variant is not known statically. This leads us to the simple conclusion, that a type variable `T`, specifying the most

specific type of a position's term sort, should not be included in generated variant position sorts.

This, however, is only possible if either `KatjaTermPos` does not have this type variable at all or the generated variants are not subtype of `KatjaTermPos`. A concrete Katja term will actually never consist of any single term, whose most specific sort is a variant sort, there are no such things as variant term constructors. A term consists of lists, tuples and externals; variants are only introduced as super types, which is one of the main differences between Katja's type system and those of other systems or languages, e.g. ML.

It therefore seems reasonable to exclude variant sorts from the `KatjaTerm` hierarchy. We have to clearly separate Katja sorts, which are defined in a specification, from concrete Katja terms, which can actually be constructed.

The type variable `T` discussed above is then an essential and statically known property of all term positions but *not* of all position sorts. Figure 11 shows the complete `katja.common` type hierarchy of the current Katja system.

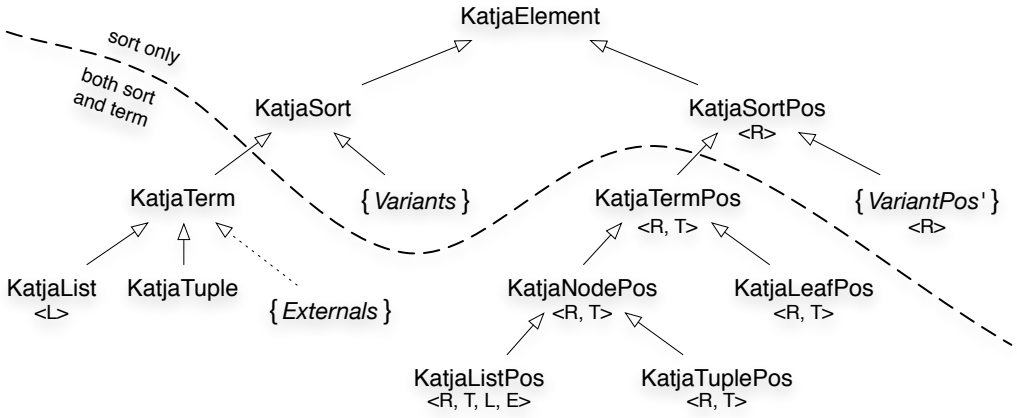

Figure 11: The `katja.common` Type Hierarchy

As variants and externals do not have any clearly distinguished super type in the hierarchy, the sets of generated types are denoted in the figure, with the super types they implement. Externals, however, implement KatjaTerm only on a conceptual level, as their position types all implement `KatjaLeafPos`, making them term positions.

I split the `KatjaTerm` and `KatjaTermPos` types into two new types each, which represent all Katja sorts in general and all concrete term sorts. The name of the latter was chosen to be `KatjaTerm` again, instead of `KatjaTermSort`, as the conceptual focus of its subtypes in Java lies on the creation

of concrete term types, rather than their sort types. The implementation hierarchy thereby starts with `KatjaTermPosImpl` as before.

The developer has now more possibilities when dealing with general positions. There is not only the notion of an arbitrary position in a fixed context, but also of a position of a fixed term in an arbitrary context. He can furthermore write code dealing with arbitrary positions in generic methods, allowing the user to pass static information through method calls.

4.12 Types and Sorts

The sorts of a position sort structure are closely related, as they are defined together and they are often used together. Many operations on positions return an arbitrary position *of the same structure*, but there is, however, no such sort to formulate this conveniently.

On terms such sorts are not needed, as there are few operations leaving the set of generated types and there are no closed sets of strongly related types. I therefore introduced sorts for Java, representing subsets of a position structures sorts.

Figure 12 shows the integration of those sorts for an example `root` statement. The sorts are generated into the specification class the root statement was defined in, so they are uniquely named using the suffix, but are also included in the specification class namespace.

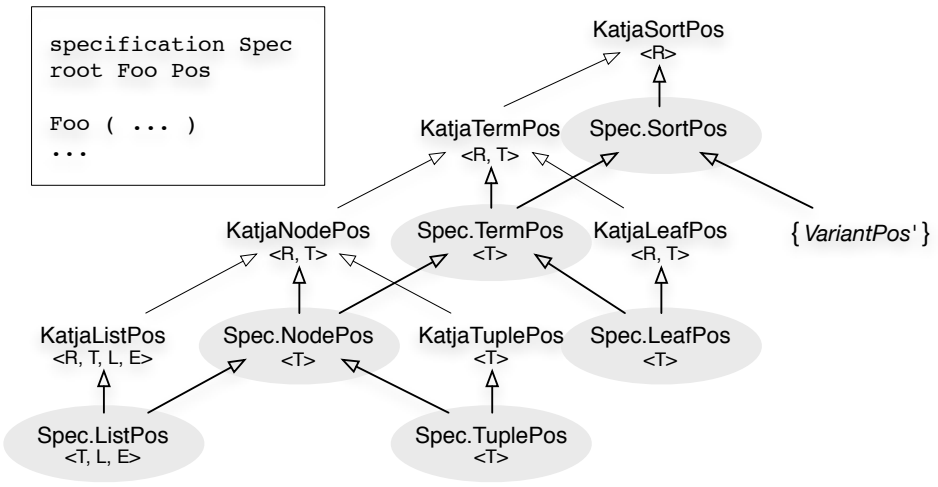

Figure 12: Position Structure Sorts and their Hierarchy

All position navigation methods will always stay within the same position structure and therefore return the new types. The user can use the

parameterless `Sort<Suffix>` type as loop variable and in other references.

Unfortunately Java does not allow to define more precise return types for most other methods along the hierarchy, due to one of the deficiencies described in Section 4.14. Methods like the special version of `replace`, which can theoretically be refined on each level of the hierarchy, cannot be refined without hassle.

Methods can be changed as follows:

- The `root` method was correctly typed in all cases anyways.
- The basic navigation methods `parent`, `rsib`, `lsib` can be improved to return any sort of this structure, i.e. `Sort<Suffix>`.
- The navigation methods `preOrder`, `posOrder`, `postOrderStart` as well as `follow` can be improved to return any sort of this structure, i.e. `Sort<Suffix>`.
- The methods `get` and `replace` cannot be improved, as their return type would have to be refined not only on the top level of the new hierarchy. Future versions of Katja might be able to show such improvements by introducing even more interface types in the hierarchy, which are not seen by the user.

4.13 Unparsing and Output

Katja terms could always be converted to `String` to put them in messages and logs, etc. The implementation was simple, cascading to subterms and their implementation, down to externals which were supposed to have a meaningful output.

But while using Katja in many contexts, different needs arose and some deficiencies of the implementation were uncovered:

1. With the introduction of term factories and static imports in Java 1.5, the `toString` output nearly resembled valid Java code to construct the term. As the only external types used are often primitive types of Java, there are only small adjustments necessary to actually make it valid. With the introduction of positions the approach isn't as straightforward as before, but I kept the general idea of such a functionality.
2. The presence of the previous feature is certainly nice to have, but not suited to solve the general persistence problem of terms and positions. Even with compilers present in the classpath there are language limitations to be considered, if the output of such a method is to be parsed

and executed from within Java. There has to be some simple plain text representation of Katja terms and positions as well as a lightweight parser turning this representation back into an object.

3. The naive recursive approach to such methods is far too slow for practical applications and large elements, as too many temporary strings have to be created and discarded in Java. A much more efficient implementation using buffers had to be found.

All these points have been taken care of in the new Katja system.

4.13.1 Efficient `toString`

The efficiency problem listed as concern three is solved by offering the more general methods

- `public Appendable toString(Appendable builder) throws IOException`
- `public Appendable toJavaCode(Appendable builder) throws IOException`
- `public Appendable toAssembler(Appendable builder) throws IOException`

which append their content to a given buffer, but are also used recursively. So only one buffer is used to create the complete output and the methods return this buffer for convenience reasons, as the user can use one cascade of `append` calls to add several parts of content.

Those methods have to throw `IOException`s, as such a buffer can also be a file or another potentially unstable target. The user can therefore use the convenience methods

- `public String toString()`
- `public String toJavaCode()`
- `public String toAssembler()`

which simply call the general ones after creating a `StringBuilder`, thereby statically guaranteeing that no exception is thrown.

The process of converting a term or position to a `String` of any kind is much more efficient than before, even if the user still calls the latter versions to create some large output.

4.13.2 `toJavaCode`

The only difference between `toString` and `toJavaCode` is the handling of external types. Where the former is compatible with all externals possible and aims on readable output, the latter does not know how to create Java code for most externals.

The reader should note, that an external is a Java type Katja does not recognize as generated by itself, but it may as well be a backend-imported Katja type from another specification. Such differentiations have to be taken into account when designing an operation like `toJavaCode`.

I decided to include several Java Types hard coded in Katja, which are easily extensible by Katja developers. At the moment these types include `String`, `Boolean`, `Double`, `Float`, `Integer`, `Long`, `Short` and `Byte`. New types can be added in the `ToStringAspect` source file, as members of an enumeration.

The developer specifies the full qualified Java name of the type as well as rules of what has to be printed surrounding the `toString` output of the element to turn it into valid Java code. In case of the listed types these include, for example, quotation marks for `String` or the appended letter `f` to mark literals as `Float`.

If an external is not listed in the enumeration, Katja generates code, which checks an object at runtime for assignment compatibility to `KatjaElement`. If the object implements this interface the `toJavaCode` method is invoked, an exception is thrown otherwise, as Katja has no idea how to generate Java code for such objects.

Katja could check for a special interface like `ConvertibleToJavaCode` instead of `KatjaElement`, but I don't consider it harmful to require the latter. As the user can't use this Katja feature at all, when dealing with final or precompiled types, it is not considered to be applicable in too complex situations. As most specifications are backed up on simple Java types or imported Katja types, the approach taken seems to be valid.

4.13.3 The (Un-)Parser Framework

A user should be able to transform Katja terms and positions to a simple, text based format, to be able to exchange them between different systems and to make them persistent. The format should then be easily parsable, without the need of complex parsing technologies, as it is not intended to be exceptionally readable or writable from hand.

One idea would be to use XML as target language, which is certainly valid, but might even be too much effort for this application. I decided to use a

small stack machine based approach, which gives some amount of freedom as well as the capability to optimize to the developer. A small interpreter reads commands of the language and assembles terms and positions using a stack.

A generic unparser is supplied for all Katja elements, which is used in the `toAssembler` method, but the user can implement his own domain specific unparser using visitors or position navigation methods. An easy way to improve the assembler code produced is to share common subtrees of a term.

The default unparser does not optimize terms at all, but optimizes positions. It constructs all root terms needed on the stack and creates positions by loading them on top of the stack and navigating through them. Terms holding many positions of the same structure are therefore efficiently represented.

An assembler file consists of an arbitrary number of lines, starting with a command, followed by arguments, separated with whitespaces. A command is a simple word beginning with `#`. The following paragraphs cover the known commands, but the reader should note that the assembler language defined here is ongoing work and might be subject to change. It's introduction was motivated by practical problems and different situations might need slight adjustments to work efficiently or to work at all.

Basic elements The commands in this section put a basic element on top of the stack and represent Java base types.

The `#string` command puts a given string on top of the stack. The trimmed argument line has to be enclosed in quotation marks, which are not included in the string. Quotation marks within the string are not modified and are not harmful to the string.
As arguments cannot span over more than one line, newline characters have to replaced in strings, as well as the backslash character for that reason. Users wanting to write their own unparser can use the following line in Java, Katja transforms the string back when reading it:

```
"#string \""+myString.replaceAll("\\\\", "\\\\\\\\").
                      replaceAll("\\n", "\\\\n")+"\""
```

The commands `#int`, `#byte`, `#short`, `#long`, `#float`, `#double` and `#boolean` push their corresponding Java base type on the stack, by parsing it from the trimmed argument using the Java API.

Those are all external types supported at the moment, but it might, however, be possible to embed serialized Java objects into an assembler file to increase the number. This wasn't necessary so far and will be evaluated when the need arises.

`#apply` This command takes the name of a constructor, as well as an arity as argument. The interpreter then applies this constructor, if known, to this exact number of arguments from the top of the stack. Those elements are removed from the stack and replaced by the new element. The top stack element is taken as first parameter, the next as second and so on.

The constructor name is unqualified, as there are no namespaces in Katja. The interpreter either knows one constructor with this name or none, in which case an exception is thrown. The arity is especially important for list constructors, as they can be invoked with arbitrary many arguments. For tuples, however, the number is required too, as it simplifies the interpreter.

When writing an unparser for terms, the developer has to unparse children of an element from last to first, so they appear in the correct order on the stack.

`#load` and `#mark` To optimize element creation, it is possible to copy elements on the stack by loading them on top of the stack. The `#load` command takes an integer number and pushes the specified element on top of the stack, positive values addressing the stack from bottom, starting with 0, negative values addressing the stack from the top, starting with -1. The former is considered to be *absolute* addressing, the latter *relative* addressing.

The `#load` command is used to efficiently construct many positions of the same structure. The common root term is constructed at the bottom of the stack and loaded to create different positions in that term.

To flawlessly integrate several assembler programs, i.e. to call subroutines creating elements, which do not know how large the stack is and which should not manipulate the stack, the method `#mark` is provided. It marks the current height of the stack, having the following consequences:

- All subsequent calls of `#load`, using absolute addressing, are evaluated relative to the mark, rather than the bottom of the stack.
- No items on the stack, which are below the mark, can be used till `#end` is called, which removes the mark, but does not stop the interpreter.

The subroutine virtually has its own stack to create terms and can call other subroutines if necessary. All routines can put shared elements at their bottom of the stack and address them using absolute values.

`#end` If there are any marks left on the stack, a call of `#end` has several consequences:

- The mark is removed from the stack, making another one visible, which can also be the default mark at 0.

- If there is no element left above the mark, an exception is thrown.
- All elements above the mark are removed, except the top element of the stack, which is considered to be the result.

If there was no mark left, except the default mark, the interpreter ends, returning the top element of the stack as result. If it is not a Katja element, an exception is thrown.

`#follow` This command takes a list of integers as argument, which are separated by whitespaces only. Those values are taken as arguments for subsequent calls of the method `get` to the top element, so it allows to navigate in positions and select subterms of terms.

This is used by the default unparser to efficiently construct positions, as the root position is loaded from the stack and the path to the position is applied to it. Another application can be the decomposition of an element created by a subroutine.

So one overall concept of the assembly language is that each command takes exactly one line and each line is variable in the use of whitespaces, which makes the parser extremely simple and fast. It is text based and consists of only few commands, which enables the user to formulate all elements possible, but to optimize the representation as well.

The interpreter for assembly programs is supplied in a generic from, with the base class of all specifications, called `KatjaSpecification`. Only the names of constructors allowed in the `#apply` command vary from specification to specification, so this part of the interpreter is generated into each specification file.

It is an implementation of the nested interface `KatjaSpecification.ElementCreator`, called `Creator` and supplies the interpreter with the necessary functionality to select a static constructor for a given name and apply it to a given number of elements.

The special method `fromAssembler` in the specification class is used to recover an element from its assembler representation. The method takes an arbitrary `Reader` and may therefore yield an `IOException`. The class `StringReader` can be used to parse strings from within Java.

An assembler file can be interpreted by any specification class knowing all involved sorts.

4.14 Compiler and Language Deficiencies

While providing the user with a most convenient and powerful interface, I stumbled across several deficiencies of compilers as well as the language Java itself. Features often had to be removed or implement slightly different, to keep Katja going and to be able to compile Katja output at all.

As far as compilers are concerned, there aren't too many options. The Java compiler provided by sun [8], as well as the stand alone eclipse compiler [3] are the only working options, as long as Java 1.5 is the supported language level. The Jikes project [2] and also the gcj [1] project are not supporting Java 1.5 and it seems only the latter may add it in the future.

I tested the eclipse compiler several times and noticed some differences to the sun compiler:

- The eclipse compiler is much faster than the sun compiler, whenever the latter slows down to finish in minutes, if ever, the same input does not affect the execution time of the former at all, finishing in seconds.

- Unfortunately it also yields phantom errors on advanced input, like complicated type hierarchies, involving many nested classes, etc. Eclipse, for example, had problems with simple features like multiple classes in the same input file, most likely cause they were not used by many people. As Katja touches the limits of Java in many ways, it tends to uncover more problems of that kind easily. To make things even worse such behavior only appears in large scenarios, so small examples showing the compiler failure are not easily produced. I did not yet take the time to work those things out with the eclipse community.

- The eclipse compiler has no problems at all with handling situations in which the sun compiler behaves badly, like resolving overloaded method calls.

Altogether I decided to use the sun compiler, which has some downsides, but does not yield phantom errors in most cases. The combination of the Java 1.5 language level and the sun compiler had the following limitations at the time of the redesign of Katja:

- Large numbers of overloaded methods, like `Case(...)` or `visit(...)`, slow down the compiler considerably. Whenever the number of methods approaches values like 100 or 150 the compiler does not seem to proceed at all and consumes more memory than assigned to by default. Invocations with input files showing such numbers were often canceled after half an hour, longer tests did not show any advance at all.

Upon renaming those methods, by appending the sort name of the case or visit method, the compiler terminated in seconds again, clearly stating it has enormous troubles with large numbers of overloaded methods. The consequences for the user are the inconvenience to use redundant method names in visitors and switches.

- The sun compiler shows an evasive bug when dealing with large amounts of input files, containing methods which are declared to throw exceptions of some type variable:

```
static interface VisitorType<E extends Throwable> {
    public void visitMySort(MySort term) throws E;
}
```

Such methods are declared, inherited, implemented and overridden in many types, defining and instantiating type variables correctly bounded by `Throwable`. Implementations even call other interface methods throwing exceptions of a generic type, which resolve correctly to the declared type. In rare situations the compiler claims a method will throw an exception which is neither caught nor declared.
A known workaround is to immediately call the compiler again, so he uses class files already produced to minimize the files needed to translate, resulting in the error being gone as the input size drops below a given limit. This result, however, could not be verified in the case of Katja, as the compiler did not properly resume to translate files with more than one defined type and ended up missing types defined in there.
With several test specifications this bug occurred only once in the development process, as I added implementation sharing along one path in the visitor hierarchy. This feature is therefore turned off at the moment, with no visible result for the user of the system.

- Java 1.5 introduced the feature to use covariant method return types in subtypes, which was essential to Katja. It is, however, not allowed to define an interface, extending multiple others with conflicting methods, which resolves the conflict by defining a method with covariant return type to all extended interfaces.

```
interface A { A term(); }
interface B { B term(); } // incomparable to A
interface C extends A, B { // conflicting term method
    C term(); // this definition solves the conflict
```

```
}
```

This is allowed in the case of a class extending and implementing several other types. This ambivalence between class and interface behavior was filed as a bug or request for enhancement under the ID 6352388 in the Sun Developer Network [9]. Sun supplied this ID after nine month of waiting and the entry sees no comments or progress so far.
Cause of this behavior it is in general not possible to define a method in a type hierarchy, which always returns its own type, or a type of a mirrored hierarchy. This would be necessary at several places in Katja:

- The `term` method cannot be typed appropriately in variant sorts.
- The `replace` and `get` methods in the special position structure sorts cannot be typed appropriately.

A workaround would be to introduce additional interfaces, which "prepare" the fusion of two inherited methods, like shown in Figure 13. Extending `A` as well as `CHelp` in case b is allowed, as the conflict is resolved by inherited methods only.

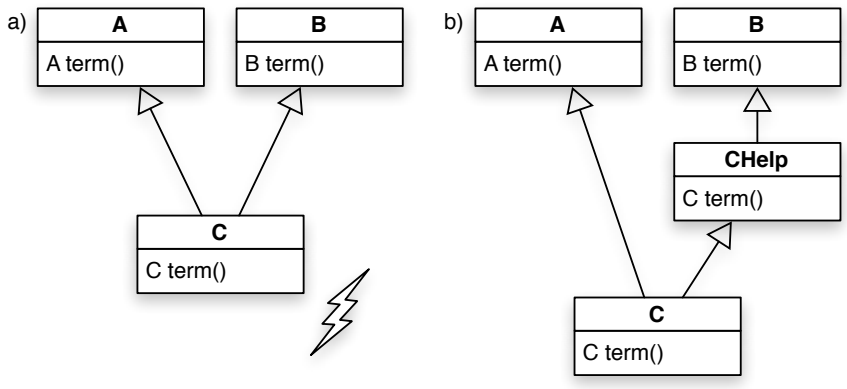

Figure 13: Return type conflicts in Java interface hierarchies

Surprisingly enough, the sun Java compiler does not allow this construction too, though this one is backed up by the language specification. The eclipse compiler even compiles case a, so we are left with several possible interpretations of the language specification.

- The sun compiler also slows down when dealing with large, complex type hierarchies, which also leads to increased memory usage. Such complex hierarchies are, for example, the result of design decisions

made in Section 4.6.2, which are therefore not implemented, having the downsides mentioned there.

5 Future Work

The presented Katja system has reached a much more stable state than ever before. Only experienced users could use all features of Katja so far and had to stick to certain rules to avoid being troubled by Katja. The core features worked well, but the complete set was never really closed.

Katja has now reached a first release status, where every user should be able to use the features offered, without restriction or unnoticed downsides. The import feature, for example, was always a cause of trouble and the integration of position sorts did not ease the problem. The redesigned system does not show the confusing behavior of the old system and imports can be used whenever the developer wants to.

There are still limitations to the use of imports, which can't be solved without introducing namespaces to Katja, but those will result in warnings or error messages to the user, instead of strange or unstable behavior.

The following sections will cover the next steps in the extension of Katja. Higher order positions are no longer listed in the future work, as their application is too situational to justify the impact on the theory and implementation of Katja. It also became apparent, that usability is hard to guarantee when introducing such a feature, especially when using generic types too excessively.

5.1 Namespaces

Considering features like position structures, imports or the upcoming attributes, it becomes apparent that Katja needs a notion of namespaces, to avoid a vast amount of possible conflicts. Katja wants to allow most possible constellations which can be thought of at the moment, without restricting the allowed set of specifications which can be imported or the position structures to be declared.

For now there is only one default namespace and nearly all sorts need to have different names. The only exception are external term sorts, but then again they need to have the same "meaning" in all contexts. It's quite easy to create conflicts when declaring a new root sort or importing several specifications.

Such conflicts can be solved, whenever the developer can change most of the imported specifications, but this is not always possible or realistic. Even

if the developer has access to the specification, he might need to adjust huge programs or libraries using this specification as well.

Introducing namespaces to Katja, however, is not as trivial as it may seem. There are several backends in Katja, which all have to be able to handle namespaces as well, so their concepts have to be included in considerations and they have to be unified.

Sorts are only simple names at the moment and the Java backend followed this design principle with all generated sorts. Adding namespaces to Katja will have a major impact on specification files and how sorts are referenced. If not all sorts should have to be fully qualified, there is need for another concept of imports in Katja.

The introduction of a namespace concept to Katja has therefore to be considered very carefully and a solution has to be consistent throughout the complete system.

5.2 Attributes and Pattern Matching

One of the very next steps is the introduction of an attribute specification and evaluation system. Users getting introduced to Katja often anticipate such a system, as it would integrate flawlessly to the current system and improve its usability even further. Katja was designed to be an attribution system right from the start, but did not make it there so far.

The choice of an object oriented, imperative programming language as host language for Katja offers many challenges as well as possible designs, whenever a feature is to be introduced. These will be considered in future work and integrated into the system.

With the presented system, however, the technical aspects of adding such a feature to Katja should turn out more convenient, as it can be added faster, separated from others and can benefit from the infrastructure created in the redesign.

A pattern matching facility will be one of the major aspects of an attribution system, as it simplifies expressions in the most common situations, which are the biggest downsides of Katja and Java at the moment. Testing terms and positions for structural properties and binding identifiers to parts of them should be done by Katja, so the user only needs to give a concise specification.

References

[1] GCJ, http://gcc.gnu.org/java/.

[2] Jikes, http://jikes.sourceforge.net/.

[3] The Eclipse Foundation. Eclipse, http://www.eclipse.org/.

[4] Jean-Marie Gaillourdet. Generation of term position algebras for Isabelle/HOL from order-sorted specifications using a modular generation framework. Internal Report, mail to: jmg@informatik.uni-kl.de, March 2005.

[5] Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785, pages 63–77, 2000.

[6] Patrick Michel. Katja: Generating immutable java classes for term and position types, December 2004.

[7] Patrick Michel. Adding position structures to Katja, June 2005.

[8] Sun Microsystems. Java platform, http://java.sun.com/javase/.

[9] Sun Microsystems. Sun developer network, http://bugs.sun.com/.

[10] Arnd Poetzsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica*, 34:737–772, 1997.

[11] Jan Schäfer. Katja: Generating order-sorted data types in Java. Internal Report, mail to: j_schaef@informatik.uni-kl.de, January 2004.