Adding Position Structures to Katja

Internal Report No. 353/06

Patrick Michel

June 16, 2005

Contents

List of Figures

1 Introduction

This document introduces the extension of Katja to support position structures and explains the subtleties of their application as well as the design decisions made and problems solved with respect to their implementation.

The Katja system was first introduced by Jan Schäfer in the context of his project work [4] and is based on the MAX system [3] developed by Arnd Poetzsch-Heffter. The reader unfamiliar with the Katja system is adviced to read those documents to get an introduction to the system.

The following sections are structured as follows: Section 2 will explain the concept of positions and visualize the idea behind them. The necessary extensions to the Katja specification language to support positions are discussed in Section 3. The Java implementation the user works with will be described in Section 4, which explains the type hierarchy, user interface and application subtleties. Section 5 will go into much more detail and discuss design decisions which arose while implementing and migrating Katja to Java 5. Last but not least Section 6 describes future work on Katja.

Usage of Terms The Katja system is based on several ideas and concepts which can be viewed on from different angles. Each has advantages and disadvantages and is more or less adequate in different situations. However, we will only use one of the alternatives, as long as the others are not needed to better visualize the notion or concept of the term.

The main concept explained in this document is the concept of *positions*. In rare cases we will refer to positions as *occurrences* or *nodes*, which are both terms used in the past to describe the exact same idea.

The terms *type* and *sort* are difficult to seperate when talking about the Java implementation of Katja, since the objects referenced by these terms are strongly correlated. However, we understand *sorts* to be the main result of a Katja specification, opposed to *types* being the result of a class or interface definition in Java.

2 The Concept of Positions

2.1 Motivation

Katja's main usage is the representation of data by order-sorted recursive data structures. The main advantage of Katja in this context is the simplicity and conciseness of its specification language, which allows fast and convenient development. The specification file gives an account of the sorts needed and their relation, which is used by Katja to generate a Java package.

This package does not only define the complex types and relations specified, but adds a rich interface with many methods used frequently when working with terms. Term types are generated immutable, which is a great advantage compared to most hand-written implementations done in a rush. Immutability is one of the most appreciated features on term handling and processing, since it enforces reasonable designs and ensures safety of interfaces between modules.

There are many applications which already profit from using *generated* Java types instead of hand-written ones, like language analyzing tools. It is common procedure to write concise descriptions of lexical and grammatical properties of languages to generate parsers. Katja is the natural extension of this, as parsers can use the generated Java types of specifications to build up the abstract syntax tree. The acutal work can then be done using a convenient interface to the syntax tree from within Java instead of embedding code into the parser.

However, we are able to present much more features to the developer than are present in the Katja system so far. These are essential features, which are cumbersome to implement by hand, but make working with terms so much easier. Expressing relations between term data, substituting parts of terms by others and converting whole syntax trees are only few example operations which should be easy to express by the user and work well together with the present concepts like immutability.

All this has been successfully implemented in Katja and will be presented in this document.

2.2 Necessary Additions

There are several situations which arise again and again when working with terms:

i) Descending in the term structure visiting subterms, but remembering from where one came to continue with other subterms later.

ii) Calculating values using subterm data, but needing information from the *upper context* (that is other parts of the superterm).

So it is often important to keep knowledge of the superterm when descending. This results in great efforts to organize all the data which is needed for the current task and makes it increasingly difficult to keep track of the essential parts of work.

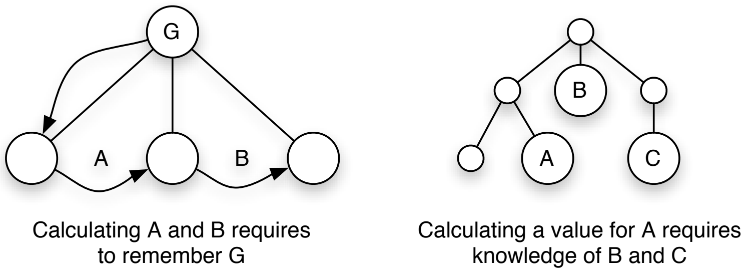

G
A
B
Calculating A and B requires to remember G
B
A
C
Calculating a value for A requires knowledge of B and C

Figure 1: Common Situations on Term Operations

Even in simple situations where it is sufficient to store a reference to the *parent* node when descending, one has to utilize a stack or use recursion, which is either inconvient or inefficient, respectively.

It is very important to realize what *upper context* really means. Everything which is only reachable from the current node by walking at least one time *upwards* in the term structure is the upper context.

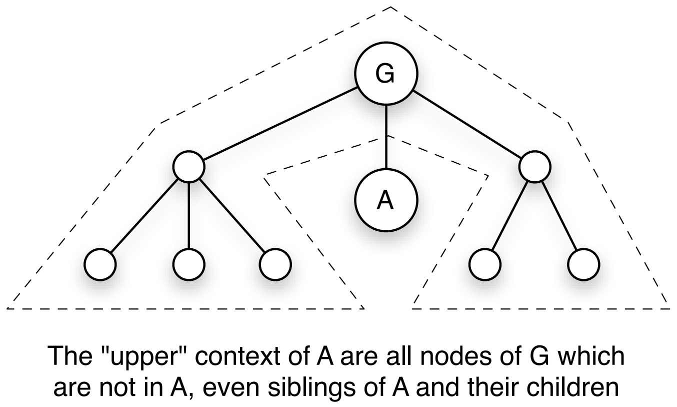

G
A
The "upper" context of A are all nodes of G which are not in A, even siblings of A and their children

Figure 2: Upper Context

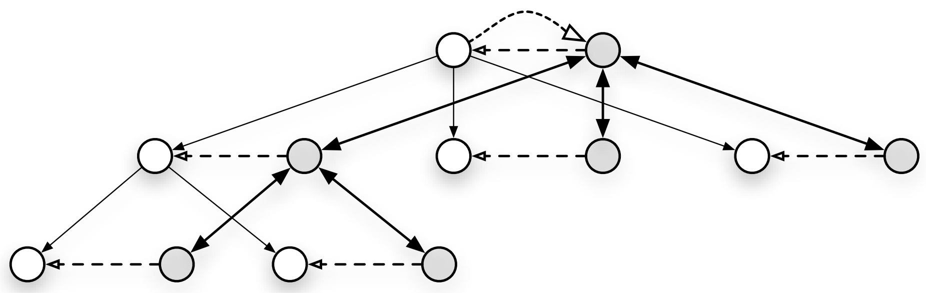
Therefore it is desirable to introduce the concept of term *positions*. These are terms, together with their position in a superterm. This concept is occasionaly referred to as *nodes*, since the notion of a node of a tree fits more perfectly to positions as to terms, since the latter don't know anything about there upper context nodes. As mentioned in the introduction, it is also valid to refer to a position as the *occurrence* of a term in another.

2.3 The Concept

When we are talking about a position we want to know three things about it:

- The subterm whose occurrence is described.
- The superterm which is the context of the position.
- The relation of the term to the superterm, given mainly as a path from the root of the superterm to the subterm.

With all this information present we are now able to reach every node of the superterm from any position in it. All positions of one superterm are considered a *position structure*, which can be thought of as the tree of the superterm with bidirectional edges, so walking upwards in the tree is possible.



The position structure can be considered an additional tree
with extend navigation properties above the term tree

Figure 3: Position Structures

The most obvious operation, which is now possible using positions instead of terms, is the `parent` operation, which retrieves the direct predecessor in the tree structure. Of course there are many more possible operations like `left` and `right sibling` or `root`, which are not marked in Figure 3.

To get back to a term, when dealing with one of it's positions, the `term` operation has to be used. Obtaining a position from a term is more difficult in general, since it is necessary to specifiy the three parameters mentioned above. To overcome this inconvenience and support easy usage and readability we only allow to create root positions, i.e. positions of terms in the context of themselves.

This allows to use a single parameter constructor which only takes one term, since the superterm is the term itself and the path is trivially the emtpy path. This operation is called `pos` and can only be invoked on special term sorts, which will be discussed later.

All other positions of the position structure created by a call of `pos` can be retrieved by navigating through the structure using the operations explained in Section 4.

3 Katja Specification Extensions

3.1 Changes

There are several changes concerning the Katja syntax and semantics, which are already made and documented in Jean-Marie Gaillourdets work on the Isabelle formalizations for Katja [1].

To support positions in Katja specifications it would not be necessary to change anything, since all information needed is present. The term sort hierarchy resulting from a specification can be transformed to a position sort hierarchy without too much trouble.

However, such a decision would have a great impact on the Java realization. The type of a position is required to reflect the corresponding term type, as well as the type of the root term of the structure. Without the latter one could not express situations in enough detail to ensure execution safety of operations.

Consider, for example, a method which calculates for a given identifier position the defining position of the identifier in a Java program. Such a method would take an identifier position and return a definition position. Given a position in a correct Java program, the method would work as required, but when given a position in the context of just a statement block, for instance, the method may not be able to retrieve the definition.

So it would be necessary to generate position types in enough detail to allow such differentations. The number of types necessary can only be bounded by $O(n^2)$, where n is the numer of sorts specified. This is far too much to generate and since only a small subset of all these types will be needed in practice we decided to extend the specification language.

The number of actual contexts used in position structures is very limited. Most developers start off with only one and eventually add a few more. Good examples for position structure contexts are for instance

- The `Program` sort used in the analysis of programming languages.
- The `Specification` sort used in more general specification languages.
- The `ProofTree` sort containing a proof for parts of a program.

- The `Expression` sort for highly recursive mathematical expressions used in various languages.

Typical positions in these contexts could be:

- An `IfThenElse` in a `Program`.
- A `HoareTriple` or `Formula` in a `ProofTree`.
- A `Binary` operation in an `Expression`.

To specify which sorts can be the context of a position structure, we introduce a new keyword called *root* to the specification language:

```
root␣<term sort>␣<suffix>
```

It is followed by a sort name together with a postfix and expresses that the `pos` operation can be invoked on this sort and returns a position of the sort "<term sort><suffix>", which is the name of the position sort corresponding to the term.

3.2 Position Sorts Defined by a Specification

The term specified in a *root* definition therefore becomes the root sort of position structures in which the user can navigate. The sorts to be created for this structures are all sorts necessary for the definition of the context sort. The names of these sorts are constructed using the term names together with the specified postfix.

The sets of position sorts generated for the following example specification with two roots are shown in Figure 4:

```
root Program AtProg
root Unary   AtUnary

Program    * Assignment
Assignment ( Identifier left, Expr right )
Expr       = Binary ( String operator, Expr left, Expr right )
           | Unary ( String operator, Expr exp )
           | Identifier ( String name )
           | Value ( Integer val )
```

In most specifications no position structure will contain a sort for each term sort, since not all sorts are reachable from a given root. This is especially true for helper sorts like lists which do not occur in the original input but are needed in processing.

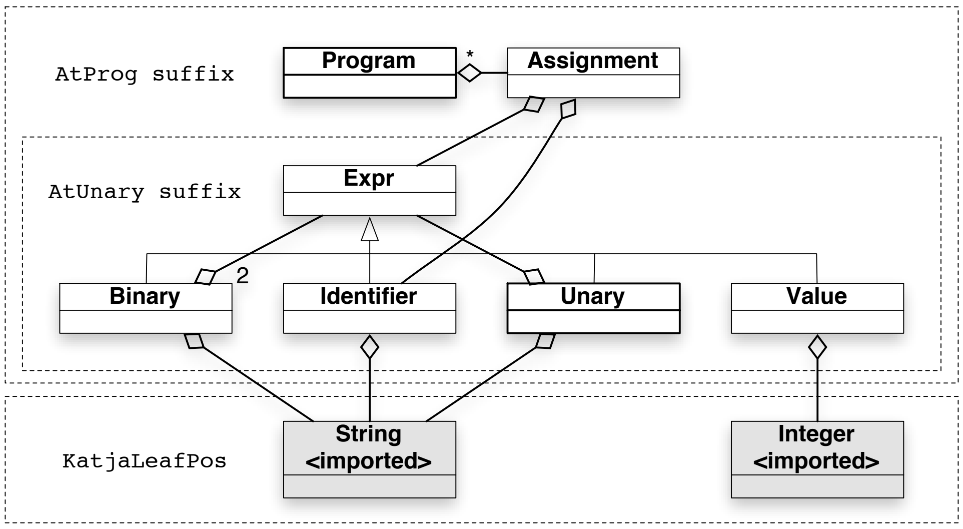


Figure 4: Position Sort Sets

With this in mind the return types of many operations can be defined much stronger. As terms have many parents in general, they often can have only one parent in a certain position structure. This allows us to strongly type the parent operation of types of this structure, so going upwards in the tree can often be done as easy as going downwards using selectors.

3.3 Possibilities and Restrictions

By introducing positions to Katja, many new types are added to a specification, which can be used in Java. But there are many cases in which one wants to consider positions as normal terms, to benefit from those Katja features:

- Storing positions in lists while processing a position structure or term.
- Defining tuples containing processing information, as well as (an) associated position(s).
- Using positions in a second specification as base sorts.

It is therefore possible to use position sorts, like imported sorts, in tuples and lists. This can be done in any specification where the sort was defined or imported.

However, there are limitations to this feature. It is not allowed to create a position structure in which a position sort (of another structure) appears, i.e. relative to a root sort which already needs position sorts in its definition.

This would result in the creation of *higher-order positions*, which describe the occurrence of a position (seen as subterm) in a superterm. This is not supported at the present state of the project, but will be considered in the future. We will discuss the usage of such a feature in Section 6.1.

The second problem is the variants, in which positions must not occur. Allowing this would make positions real `KatjaTerms`, which is also not intended at the present state. However, this concept is interesting and would open up many new possibilities. This will also be discussed in Section 6.1.

4 Java Realization

4.1 Type Hierarchy Changes

The term position implementation conforms to the same design desicions made for the original system. The hand-written type hierarchy needed for positition types exists in parallel to the term types and extends the generic `KatjaElement` class.

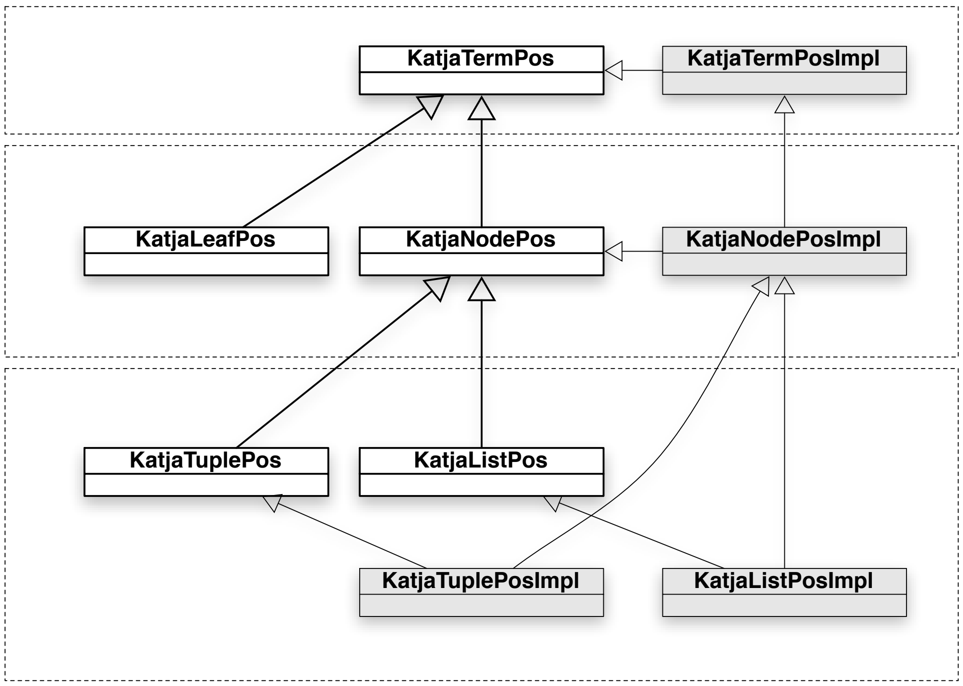


Figure 5: UML Diagram of the Position Type Hierarchy

The `KatjaTermPos` is the most generic type of the position hierarchy, implementing the `KatjaElement` interface. We then separate those types capable of having children from those who cannot, which are therefore called *leaves*.

`KatjaLeafPos` is a generic Java class which wraps all kind of not Katja generated sorts, like built-in or imported sorts. We do not generate position types for such sorts, but use this type constructor to get them.

The main difference to a `KatjaNodePos` is found in the return type of the `term` method, which yields the corresponding term of the position. For leaves this can only be `Object`, but for nodes this can be set to `KatjaTerm`, which is obviously a wide difference in the usable interface.

Note that this difference is most important for the `parent` operation which in any case yields a `KatjaNodePos`, so one can be sure the corresponding term of a parent position has the full `KatjaTerm` interface available.

The rest of the hierarchy is self-explaining when compared to the `KatjaTerm` hierarchy and is also splitted into types and implementations.

4.2 Position Interface

Positions have a rich interface, which offers most of the operations available on terms plus a great addition of advanced ones, for which positions where designed in the first place.

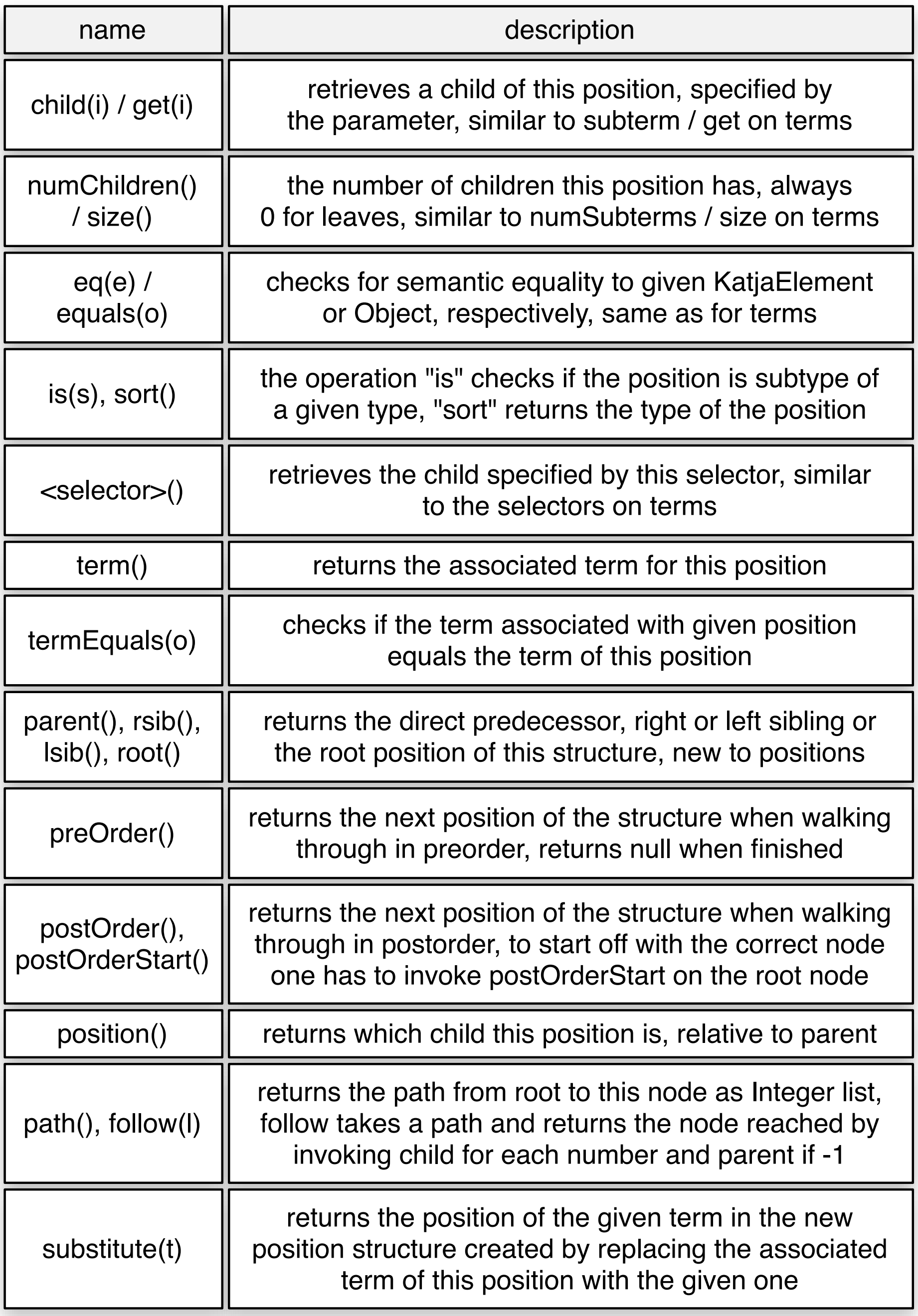
Figure 6 shows most operations available, though this cannot be a complete list since there are many more useful and commonly needed operations on positions, which will eventually be implemented in the future. Note that there are only few list operations available for list positions, namely `first` and `last`, since positions are immutable, too. The operations available on term lists just create new terms, without altering the ones they are invoked on. This is not transferable to positions since one would leave the current position structure in altering the associated term of a position.

Actually list positions can be considered as tuples, as they have a fixed size and fixed types in each component.

4.2.1 Iterators and Visitors

Visiting subterms or children of Katja elements is a task which occurs again and again when working with Katja. On positions this can be easily done using the post- and preorder methods, since positions have enough structural information present to calculate the next position without external help.

On terms this is an inherent problem which only allows visiting all subterms of a given superterm. In Katja this can now be done using the `iterate` operation, which yields an iterator that visits all subterms (down to the



name	description
child(i) / get(i)	retrieves a child of this position, specified by the parameter, similar to subterm / get on terms
numChildren() / size()	the number of children this position has, always 0 for leaves, similar to numSubterms / size on terms
eq(e) / equals(o)	checks for semantic equality to given KatjaElement or Object, respectively, same as for terms
is(s), sort()	the operation "is" checks if the position is subtype of a given type, "sort" returns the type of the position
<selector>()	retrieves the child specified by this selector, similar to the selectors on terms
term()	returns the associated term for this position
termEquals(o)	checks if the term associated with given position equals the term of this position
parent(), rsib(), lsib(), root()	returns the direct predecessor, right or left sibling or the root position of this structure, new to positions
preOrder()	returns the next position of the structure when walking through in preorder, returns null when finished
postOrder(), postOrderStart()	returns the next position of the structure when walking through in postorder, to start off with the correct node one has to invoke postOrderStart on the root node
position()	returns which child this position is, relative to parent
path(), follow(l)	returns the path from root to this node as Integer list, follow takes a path and returns the node reached by invoking child for each number and parent if -1
substitute(t)	returns the position of the given term in the new position structure created by replacing the associated term of this position with the given one

Figure 6: Position Operations

leaves) of the term it is invoked on. This iterator can be used in enhanced for loops as is described in Section 5.1.3.

A solution which works on both terms and positions is the application of the *visitor pattern*, for which Jan Schäfer added support in Katja by generating visitor classes.

Last but not least all lists and list positions implement the `Iterable` interface, which allows to use them in enhanced for loops, to iterate over all items of these lists.

4.2.2 The Substitute Operation

The Katja interface is held purely functional. When working on Katja terms and positions, it is impossible to actually change them, but copies will be made and returned by operations. As this works perfectly with terms, it is not that easy for positions.

Changing only one leaf in the corresponding term of a position does also alter the whole context of the position. For terms the user could simply get subterms of already constructed ones and put them together to new ones using constructors.

To combine both concepts of easy navigation using positions and easy construction of new terms, Katja offers the substitute operation, which allows altering the context without giving up immutability. It constructs a new term by replacing the term of the position in the superterm with the given term and returns his position in the new context.

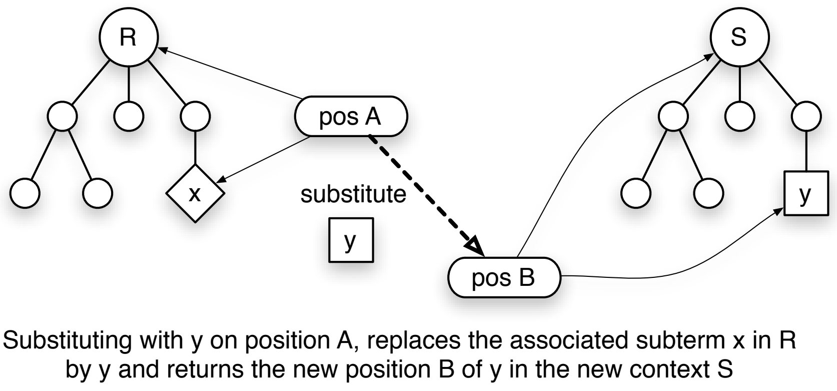

R
pos A
x
substitute
y
pos B
S
y
Substituting with y on position A, replaces the associated subterm x in R
by y and returns the new position B of y in the new context S

Figure 7: Substitution of Positions

There are several applications for such an operation:

- Transformations of abstract syntax trees in more general or enriched ones.

- Top-down construction of terms, which is most teddious without using substitute.
- Normalizing parts of abstract syntax trees using the same syntax.

Taking the example specification of Section 3.2, we could replace all positions of $0 - x$ by the unary expression $-x$ using this Java program:

```
public void replaceBinaryMinus(ProgramAtProg prog) {
    for(KatjaTermAtProg current = prog.postOrderStart();
        current != null; current = current.postOrder()) {

        if(!current.is(BinaryAtProg.sort) continue;
        BinaryAtProg binary = (BinaryAtProg) current;

        if(!binary.left().is(ValueAtProg.sort) ||
           !binary.operator().term().equals("-")) continue;
        ValueAtProg value = (ValueAtProg) binary.left();

        if(value.term().val() != 0) continue;

        current = current.substitute(Unary("-", binary.right()));
    }
}
```

Note that most lines of such a method handle the recognition of a certain pattern (binary minus, with 0 operand in this case) and can be avoided in future revisions of Katja, by using pattern matching facilities. See Section 6.3 for more details.

4.3 Advanced Usage

To overcome certain limitations of positions at this point, some strategies where successfully applied in applications. Most notable is a workaround for *higher-order positions*, which are positions of other positions. Note that we do not require these base positions to be *term* positions, i.e. they can already be of higher level.

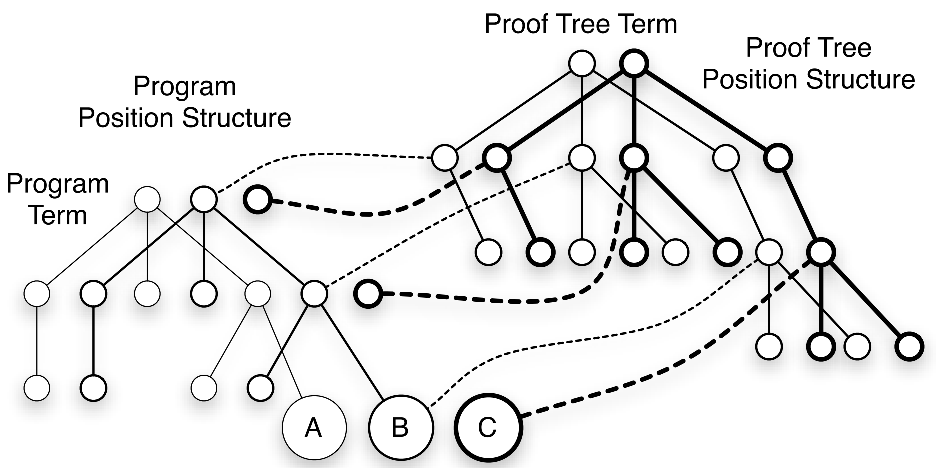
Consider an application which has to store proofs of program properties, like the Jive system [2]. These proofs are trees where nodes represent proofs for parts of the program, containing Hoare triples, consisting of two formulas and a program part. The latter is most naturally modeled as position in the abstract syntax tree of the input program.

When creating a position structure on a proof tree, we therefore would get positions of program parts which are themselves positions on program terms. These program term positions are also sub*terms* of the proof tree and therefore associated with positions in it.

If such a program fragment would, for instance, be an `Assignment`, it's position could be of the sort `AssignmentAtProgram`. The higher-order position needed would then be suffixed again, for example `AssignmentAtProgramIn-ProofTree`.

The reader should realize that there will also be positions created for all the subterm positions of an `Assignment`, so the proof position structure completely descends into the program fragment. The part of the program position structure starting from the `Assignment` node downwards will therefore be completely mirrored in the proof position structure.

These higher-order position sorts are extremely useful when working with a proof. For a given Hoare triple position, the user can browse the program fragment the triple is associated with, without loosing neither the program context nor the proofs one. So you can, for instance, find out about the local parameters of a method in which the assignment is situated, as well as the role this Hoare triple plays in the greater context. All from having a single reference to the `Assignment` in the context of a `Program` *and* the `ProofTree`.



A is a term in Program, B the position of A in Program,
but also a term of Proof Tree, C the position of B in Proof Tree

Figure 8: Higher-order Position Structure

Figure 8 shows which part of this functionality can be mimiced with the current Katja system. It is possible to link positions from the program position structure (B) to the proof tree and use positions of this tree (C). The important limitation on this point however is that these program positions are leaves for the proof tree. It is therefore not yet possible to navigate into subterms of the linked program fragment by using the proof position structure, but one has to use the transition to the program position structure to do so (see the parent of B as example).

This is done by the `term` operation, since the progam positions are considered terms of the proof. You thereby loose the proof context, which has to be saved by the user. However, this procedure allows to store the program fragments context while using proof positions, so one does not need to keep track of the whole program.

5 Implementation Details

5.1 Java 5

The introduction of Java 5 had a great impact on Katja, for both the realization of Katja and the generation of Java packages. Especially the latter opens up much more possibilities for the user as far as functionality and usability are concerned.

In the following we will go through the various new features introduced to the Java programming language and discuss their impact on Katja, both for term and position generation.

Note that the *metadata annotation* and the *typesafe enum* features are left out, since they did not have any notable impact on the core of Katja for now.

5.1.1 Generics

Generics allow the Java programmer to implement some kind of class scheme, that is a general class, which can be instantiated to work with all kinds of base types. Most common examples for the use of generics are container classes, like lists or maps, which have the same functionality no matter what is stores in them.

Generics are therefore also called *type constructors*, since they are no real types, but are used to create new types, by applying them to base types. They describe a whole family of types without writing them all down, which opens up possibilities for the creation of term types by Katja, as well as for dealing with infinite many types.

The first question one has to answer is: "Why do we keep generating types for Katja sorts at all?" The question is valid, since one could just write term type schemes, like `KatjaList<T>` or `KatjaTuple`$<T_1, \ldots, T_n>$, which are than applied to other types.

However this is not a real option as there are several downsides:

- There are no variable argument type constructors to handle tuples of arbitrary size, while keeping strongly typed component types.
- There is no possibility to introduce selectors to tuples, which are essential to the use of Katja sorts in Java.
- There is no possibility to generate type specific code into the types, like upcoming attribution information (see Section 6.2).
- Constructed types are purely static types, only used by the compiler, and do not differ at runtime, what limits possibilities on working with terms and positions.

Besides these ideas, there are several applications of generics in Katja. The `KatjaLeafPos` constructor, for example, is used to create position types of imported sorts, for which no position types are generated. It therefore takes an arbitrary type, only limited to `Object`, and gives an subtype of `KatjaTermPos`.

Another application arises when thinking of higher-order positions, which are described in Section 6.1.

5.1.2 Covariant Return Types

Most important for Katja was the introduction of covariant return type support to Java. When implementing a subtype of a given class, this feature allows to use a subtype of the orginial return type of a method, when overriding it.

This is of great importance to the Katja type hierarchy. Without Java 5, the `KatjaTerm` type did *not* have a subterm method, what made working with terms most tedious, as one had to cast a term down to it's actual type to utilize such methods.

This situation was a result of the design descision to always get the best possible typing information from a given method. Once declared, the subterm method would always have the same return type, which has to be `Object`, in all subtypes. This was not acceptable, since lists for instance always return a reference of their item type. So methods were defined as deep as possible in the type hierarchy, to be able to use the most specific return type, what results in the method not being available when referencing an object with a supertype.

With Java 5 we are now able to introduce methods in the most general type possible and use the most specific return type at all levels of the type hierarchy. Take the `term` method on `KatjaTermPos` as an example, which returns `Object` in the original definition, as there can be positions above all kind of types. Of course this method should already be introduced in that type, since it enables us to get the term of any given position without knowing it's exact type.

`KatjaLeafPos` can refine the return type to it's generic parameter `L`, since the type `KatjaLeafPos<L>` is known to always be situated above a term of type `L`. `KatjaNodePos`, however, specializes the return type to `KatjaTerm`, since all `KatjaNodes` are designed to be positions of `KatjaTerms`. `KatjaTuplePos` always returns a `KatjaTuple`, whereas `KatjaListPos` always returns a general `KatjaList<?>`.

There are also limitations to the use of this new feature. As generic types created with the same type constructor but different parameters are uncomparable, methods which return generic types can not be specialized in some situations.

In multiple inheritance situations one cannot resolve typing conflicts arising from uncomparable return types using a new covariant one. This situation is best explained with the `term` method again, when looking at a variant hierarchy:

```
root D Pos

A = C | X
B = C | Y
C = Z
D ( A, B, C )
...

// results in the Java classes (simplified):

interface APos {
    A term() {}
}

interface BPos {
    B term() {}
}

interface CPos extends APos, BPos {
    C term() {}
}
```

So the type `CPos` would inherit the term operation with uncomparable return types `A` and `B`, but would specialize both by using `C`, which in principal resolves the conflict, but is not allowed in Java. However, this situation would be perfectly valid in the Katja context.

So we are unable to correctly type the term operation on variants at the moment and it is necessary to cast down the object of a variant reference to it's real type to get more information on it's term type than `Object`.

5.1.3 Enhanced For Loop

Enhanced for loops allow to walk through any `Iterable` object using a new *for* syntax. Therefore Katja supports putting any list term or list position into an enhanced for loop to get all items. Using the example specification from Section 3.2 and a given `Program` *prog*, an application looks like this:

```
for(Assignment a : prog) {
    ...
    for(KatjaTerm term : a.iterate()) {
        ...
    }
    ...
}
```

One can also use the result of the `iterate` method in enhanced for loops, which iterates through *all* subterms (not only direct subterms) of a term. This is also shown in the program fragment above.

5.1.4 Static Imports

Static imports allow to use static members of a namespace without prefixing them with the namespace everytime. This feature allows the Katja user to build up terms in a Java program in a very natural and concise way. Katja offers the `TermFactory` which essentially wraps all term constructors, so constructors can be used without the need to write `new`:

```
static import TermFactory.*;
...
Program prog = Program(
    Assignment(
        Identifier("a"),
        Value(5)
    ),
    Assignment(
        Identifier("b"),
        Binary("+", Identifier("a"), Value(7))
    )
);
```

However, this feature does not only allow convenient term construction but enables easy and transparent use of factories, which is important when it comes to internal realization and performance of Katja.

Without public constructors we are able to do maximum term sharing, by using the same object again and again for syntactically equal terms. This does not only save memory, but allows to compare terms using their reference instead of using hashes or recursive comparison of subterms.

5.1.5 Varargs

Varargs allow the specification of methods of variable parameter count. The last parameter of a method can therefore be declared to be of variable length, which results in this parameter being of the array type of the original parameter type, with respect to the implementation.

This allows, for instance, to implement the `println` method known from the `C` programming language:

```
public int println(String text, Object... parameter) {
    ...
    String next = parameter[i].toString();
    ...
}
```

Katja uses this feature to allow list constructors to take an arbitrary number of items, which again greatly improves readablitiy of larger terms.

5.1.6 Autoboxing/Unboxing

Autoboxing implicitely converts a primitive type (like `int`) to it's wrapper class (like `Integer`), in cases where the context needs the latter, but the former is given. Unboxing describes the inverse behavior.

Katja used to have primitive built-in types, which specifications could use as base sorts. As it was also possible to import any Java type to a specification, users were unsure which to use. By introducing the boxing features to Katja, the Java types got much more convenient, since one does not have to use a constructor to get a literal of these types and no selector to get their value.

Using the example specification of Section 3.2 we can show the difference on the `Value` type:

```
// without autoboxing/unboxing
Value a = Value(new Integer(5));
int b = a.val().intValue();
```

```
// using the builtin type KatjaInt
Value a = Value(new KatjaInt(5));
int b = a.val().intValue();

// using Integer with autoboxing/unboxing
value a = Value(5);
int b = a.val();
```

5.2 Type Information

Great efforts are made to always express the best type information one can get when it comes to parameters and return types. However there are Java limitations as well as theroetical problems which prevent expressing the best possible information.

There are far too many places of interest to discuss them all in this context and some are already mentioned in previous sections, so we will only look at some additional ones of greater interest.

For some time, Katja only generated precise return types for operations, when they were appropriate in all situations which could be present in a specification. So it is, for example, in general not possible to type `subterm`, `parent`, `rsib` or `lsib` any better then with `Object`, `KatjaNodePos` and `KatjaTermPos` respectively.

However, this is not true for all sorts in any specification. For the example specification in Section 3.2, the `rsib` as well as the `lsib` methods invoked on an `AssignmentAtProg` always return an `AssignmentAtProg`.

Such situation specific type information will be available to the user in future revisions of Katja.

5.3 Time and Space Complexity

The authors of Katja recommend using positions whenever possible, since they are much more powerful and open up many possibilities. Though migration to positions when starting off with terms is not that hard, one almost always ends up in situations where position operations are extremely useful.

To justify this advice we have to look at the downside of creating additional objects at runtime and using complex operations on positions. Many operations use a path directly from or to the root position and their complexity is therefore bounded by the height of the structure, which can be asumed to be logarithmic in many applications. However, we will denote the number of nodes in a structure with n and the height of a structre with m.

First of all we consider the position structure of one given term. From the theoretical point of view it consists of as many positions as the term has nodes, since any part of the term has a position within the superterm. This results in positions taking about the same dimensions of space as the original term ($O(n)$).

But this is not completely true for Katja, since one does not necessarily need all positions from one structure. Therefore Katja is as lazy as possible, when it comes to the creation of position objects. Since one can only create objects in roots of term trees by using the `pos` method, we leave the downwards references of the position tree uninitialized.

This is quite a natural approach, since the downwards navigation information is already present in the term structure and only needs to be copied to the position structure for optimization reasons (see Figure 3 for an illustration). So position objects are created on demand and will never be created for parts which are not needed by the application.

All basic navigation operations on positions are therefore of constant time and space complexity ($O(1)$), which is the creation of one object at the first invocation only.

This is especially important for term transformations using positions. If one explores a term using position operations and creates a new term with some parts replaced by processed or normalized data, one cannot continue exploration using the same position structure, since it does not match the new context. This is no Katja limitation but a theoretical one.

One has to create a new position structure for the term instead and can be sure that only those positions that were really needed for the previous term got created and that they will be freed together with the superterm, as soon as one does not reference it anymore.

In this context, the `pre-`, `postOrder` and `substitute` operations are of special interest to us. Substitute handles the complex operation of creating a new position structure above a modified term and additionally returns the same positions with respect to the upper context, compared to the one we invoked the operation on.

This is done in an on demand way too, i.e. only the positions on the path from the new root to the replaced term are created from the new position structure. This leaves us with linear time and space complexity ($O(m)$), with respect to the height of the structure, depending in detail on the style of application. The current state of the Katja project follows this approach.

However, this will most probably be reworked in a future revision to be even more efficient. The general idea is the same as for lazy evaluation of positions on terms described above, where the downward references were present in form of the term trees, so it wasn't necessary to initialize those in advance.

Exactly the same can be done with the upward references in the new created position structure, since the upper context of it is the same as for the position structure `substitute` was invoked on. One can therefore create those nodes on the path to the root the moment they are needed by looking to the old position structure.

This makes substitute a constant time and space operation with respect to positions. As the manipulated underlying term of the structure can be created on demand as well, the complexity transfers flawlessly to the time and space considerations of the term part. This enables extensive use of positions to transform existing terms.

The reader should note, that this optimization does not guarantee constant complexity for all position operations in general. The invocation of `root`, for example, is constant in general, but will force Katja to create all positions from the point of a substitution to the root, which we just discussed to be lazy evaluated above. So it is recommended to use basic navigation operations when rapidly changing position structures, but any operation when working with a fixed one.

The `pre-` and `postOrder` methods nicely play together with substitute, since one can always invoke those operations on the position of the *actual* position structure, without breaking correct order. So it is, for instance, possible to normalize an abstract syntax tree by walking through it in post order and substitute terms on the fly with new ones.

Those higher level navigation operations need constant to logarithmic time and therefore logarithmic space on new position structures in the worst case. So the time and space complexity of a total conversion of an abstract syntax tree is directly bounded by the number of nodes in the original term tree times the complexity of the traversal order and therefore $O(nm)$. This complexity even holds for the current implementation, since `substitute` is linear in m too.

6 Future Work

6.1 Higher-Order Positions

In the previous sections we came across some unusual, but interesting ideas, which we will investigate now in more detail.

In Section 4.3 we talked about higher-order positions, which arise naturally in common applications. Though it is not a simple thing to use for unexperienced users, higher-order positions are a concise way to deal with and describe complex data.

To introduce them to Katja several questions have to be answered:

- What is their semantical model?
- Which changes have to be made to specifications?
- Are all *levels* of higher-order position sorts free to use (within specifications and/or in Java source code), or do they have to be specified?
- Is there a need to refer to sorts of a precise level or is it enough for specifications to deal with normal and higher sorts?
- How can they be used from within Java?

Most questions can not be answered for themselves, but they point to bigger design decisions.

A first approach was already mentioned in Section 4.3. The idea is to consider positions as terms and therefore let them implement their own term type interface.

This allows recursive type constructors to be used to create arbitrary high position levels. The first order positions use the terms as basetypes, all others use positions types created in this way. This idea carries to Java 5 flawlessly. It would be possible to create one position constructor for each type, that takes the base and context as parameters.

This idea solves most implementation problems and questions as well as the theoretical problem to supply infinite many types for arbitrary position levels. However, several problems remain.

What does *equality* mean on positions? It is now possible to pass positions to methods, where terms are specified as parameters, since they are now terms too. The equality implementation should take care for such implementations and check for term equality. All operations inherited from `KatjaTerm` should work on positions as if they were invoked on their base term.

This conflicts with the idea that positions of different levels should be separable, as they have different degrees of additional information. So this idea allows to use positions as terms, which is perfectly valid in higher-order applications as seen in the example Section 4.3, but has inherent theoretical problems in the treatment of positions.

Another idea is to keep positions being subtypes of `KatjaTerm` but not subtype of the term type. Instead an interface is defined, combining both the term and the position type, which implements `KatjaTerm` and is used as parameter in the position type constructor (see Figure 9).

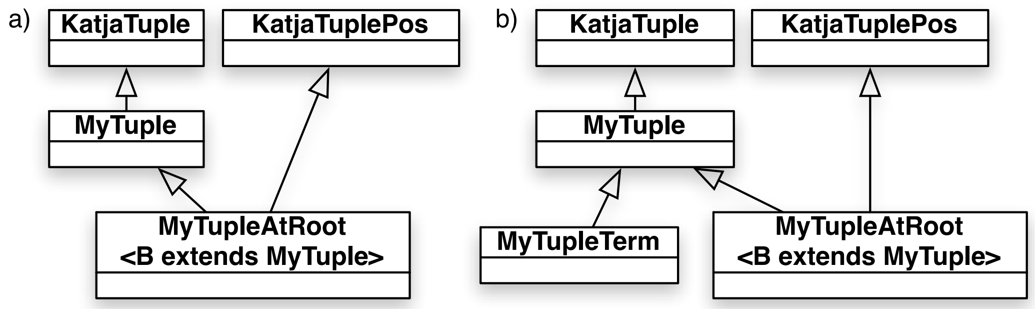


Figure 9: Implementation Ideas for Higher-Order Positions

This allows to create unlimited order positions and makes positions be terms, but does not have the downside of passing positions to methods which expect terms. As a matter of fact none of the position or term types are comparable, as long as one does not use wildcards within a type constructor, so it is possible to express the exact type needed in a method.

At a first glance this approach solves most problems and has many advantages, with respect to several views on the system, like implementation, usability and theoretical background. However, it will be necessary to evaluate the approach in more detail in the future, before it can be added to Katja.

6.2 Attribution

Katja was designed to be an attribution system, as is reflected in its name, which stands for "**K**aiserslautern **At**tribution System in **Ja**va". At present state it does not provide special facilities to store or compute values of terms.

However, positions did a great step towards term attribution. Positions give syntactically equal terms an identity by differentiating them with respect to their occurrence. All these identities can be assigned values by the use of positions.

A design pattern was successfully applied within Katja itself, to compute and store attributes for an abstract syntax tree. The computation is done by static methods, which are collected in one class per root used as namespace.

If the calculation is time consuming or demands many calls of other attributes, the result is stored in tables, which are also declared static in the same class. The immutability of the parameters and the functional behaviour of the attributes themselves justify this procedure.

So there are three additional parts in the context of attribution, which can be supported by future extensions of Katja:

1. The calculation of attributes.
2. The storage of complex attributes to increase performance.
3. A convenient interface to integrate attributes into Java.

For the first part, the user has to write Java code for now, but Section 6.3 describes another approach which will be implemented in the future.

The storage, mentioned second, can be acomplished either by generating tables for attributes named in the specification or generating class attributes to decentralize the data to the places they are associated with. This point will be transparent to the user and has to be evaluated in more detail, with respect to time and space complexity.

The third part, however, contributes to the usability of attributes for the user and therefore gets the main focus and demands consideration of all possible design decisions. It is, for example, possible to generate methods into position types, that correspond to attributes. The idea allows convenient use of attributes but does not consider, for instance, the addition of attributes by specifications which import the one where the position types were created in the first place.

So it will be necessary to find reasonable ways of implementation *and* usage, which are compliant with design decsions made for specification features.

6.3 Pattern Matching

A major part of the work with abstract syntax trees is the recognition of structural patterns within terms.

The flexibilty and expressiveness gained by variants, which is needed in adequate specifications, results in the creation of arbitrary terms at runtime, which differ in two dimensions:

- Structurally by the use of variant sorts in tuples or lists.
- By value in leaves.

As the example code in Section 4.2.2 shows, the recognition of patterns within Java is as cumbersome as simple and should be expressed in a more concise manner. The definition of attributes will also rely mainly on pattern recognition, so it is valid to introduce some kind of pattern matching facility to Katja.

A first approach is the definition of a small functional language, which can be used to define attributes and functions within the specification file.

Katja will then translate these to Java methods, which can be accessed from the position interface.

Such a language would allow pattern matching similar to the one implemented in the MAX system [3]. As Katja is already an appplication of itself, some parts can be generated by bootstrapping. With the introduction of a small functional language, we will be able to generate the Katja system from a concise specification file to at least 95%.

However, it will also be of interest to introduce pattern matching to the Java interface generated by Katja. This can, for instance, be done by allowing the user to construct pattern objects, which represent structural as well as concrete parts of a pattern, in the same way "normal" terms a constructed. By comparing the pattern object with a term or position in question, all open bindings in the pattern object will be set to the matched parts.

For the example code in Section 4.2.2, it would therefore be sufficient to define the pattern once and compare it to all positions of a program. The new code fragment might look like this:

```
public void replaceBinaryMinus(ProgramAtProg prog) {
    ExpressionPattern right = AnyExpr();
    BinaryPattern pattern = Binary("-", Value(0), right);

    for(KatjaTermAtProg current = prog.postOrderStart();
        current != null; current = current.postOrder()) {

        If(pattern.matches(current))
            current = current.substitute(Unary("-", right.term()));
    }
}
```

The `AnyExpr` constructor is here supposed to deliver a match object, which is equal to all possible `Expression`s and keeps a reference to the matching `Expression` when compared. The user therefore creates bindings to parts of the pattern which can be used after a successfull match.

The combination of the Java patterns and a functional language for specification files will be sufficient to express most applications completely in a reasonable, abstract and convenient way.

References

[1] 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.

[2] 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 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2000.

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

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