Language Independent Structure Editing
using the ToolBus

Tobias Kuipers

Language Independent Structure Editing

using the ToolBus

Tobias Kuipers

Doctoraalscriptie Informatica
Universiteit van Amsterdam
Faculteit WINS
Sectie Programmatuur
Afstudeerdocent: Prof.dr. P. Klint
Begeleider: Dr. M.G.J. van den Brand
Examendatum: 20 december 1996

©1996, Tobias Kuipers

Colofon

This document was typeset using Leslie Lamport's LaTeX macro-package for Donald Knuth's TeX typesetting system. The bodytext is typeset in Computer Modern Roman 10pt, with keywords in Computer Modern Sans-Serif 10pt and Computer Modern Typewriter 10pt, all by Donald Knuth. The bibliography database and cross-references were maintained using BibTeX, by Oren Patashnik. Figures were created using `dotty` by Stephen C. North and Eleftherios Koutsofios, `pic`, by Brian W. Kernighan, and `xfig`. Input for `dotty` was generated from the specification by `get-import-graph` and `ig2dot` by Leon Moonen. Typeset specifications were obtained through ToLaTeX by Eelco Visser. The cover of this thesis was designed by Maartje Prins.

Acknowledgements

During the work on this thesis, I have received a lot of support from a lot of people. I am going to thank them all here, so bear with me...

First of all, I would like to thank Mark van den Brand for his seemingly unconditional support. He was always there to answer questions, and did not mind that the subject of this thesis changed quite a bit during the work.

Paul Klint provided me with a firm deadline, without which this thesis would probably have never been completed. Furthermore, he always asked precisely the right questions.

Eelco Visser almost never complained when I had ToLaTeX, or other questions. He gave me advise on a number of topics regarding this thesis, and other things.

Thanks go out to Leon Moonen as well. We shared a room for over a year. This may not have been the most productive set-up, but it was usually very inspiring, and I got to learn Perl.

I have had a number of very interesting discussions about the subject of this thesis (and other things), with other people from the Programming Research Group: Pieter Olivier and Chris Verhoef.

Discussions of a more absurd nature ("But how does the Heisenberg compensator work?") were provided by my very good friends and (former) fellow students: Bas Luttik, Ramin Monajemi, and Robert Belleman. These discussions took often place in one of the best pubs of Amsterdam (and anyway the one nearest to the department): Eik en Linde.

Then there are the friends outside computer science, who made sure I went out every once in a while. I am not going to mention every one of them, but rest assured, I thank you very much.

A special dedication goes out to my parents, who always had me make my own decisions. Even though they still do not have any idea what I am doing, they support me (I will visit you soon to explain this thesis...), no matter what. I thank my brother too, for his patience when we were out drinking, and I started monologues on the state-of-the-world in general.

None of this would have ever happened without Maartje, my love. Thanks.

Section 2.1.4 was written in part by Mark van den Brand. The picture in Figure 3.1 is a modification of a picture by Pieter Olivier. The example ToolBus script in Figure 3.4 was written by Eelco Visser, as was the syntax definition for the While language in Appendix B. Parts of Section 1.4.2 were inspired by Appendix A of [Moo96]. The grammar in Appendix C was taken from [BKOV].

Contents

Chapter 1

Introduction

A *programming environment* is a collection of software tools needed to program a computer. One of the components of such an environment is invariably an *editor*. The editor is needed to input the source text (or otherwise manipulate it).

At the Programming Research Group (University of Amsterdam) and the Centre for Mathematics and Computer Science (CWI), a system has been constructed to allow for the *generation* of these programming environments. This system (the ASF+SDF Meta-Environment) takes a language definition as its input and generates a number of tools to construct a programming environment for the defined language.

One of these tools is an editor, the Generic Syntax-directed Editor, or GSE [Koo94] for short.

The ASF+SDF Meta-Environment is a monolithic system. All its functionality is built into one large program. As a result it is rather inflexible. It is very hard to add new functionality to the system, or replace a component with a newer version. All this has led to the decision that the ASF+SDF Meta-Environment needed to be re-implemented. The new Meta-Environment should preferably be written using the old Meta-Environment. Since the Meta-Environment can be used as a system for the formal specification of (programming) languages, we would then have a formal specification of the new Meta-Environment.

The idea is to use this specification not as a starting point for the actual implementation, but to *generate* the implementation from the specification.

And, because one of the reasons to replace the old Meta-Environment was its inflexibility, the new system should be modular. This requires the separate specification (and implementation) of the various tools needed in the Meta-Environment. It also requires a mechanism to connect all these tools, to form a single application. This mechanism has been implemented in the ToolBus. The ToolBus is a software architecture that was specifically designed to allow for the interconnection of a diverse and heterogeneous set of software tools.

The work that is presented in this thesis is the specification of one of the tools that should make up the new Meta-Environment: the structure editor. Apart from that it is also an exercise in using the ToolBus for a programming-environment-like system. Finally, the specification as presented in this thesis generates the actual implementation. Part of the work in this thesis is in finding out whether generating an implementation is possible and whether the final implementation is efficient enough to facilitate large interactive systems.

We do not, however, attempt to give a formal specification for every aspect of the structure editor. The user-interface (or what we will call the text-editor for the remainder of this thesis) is not formally specified. It is written in a special-purpose language for implementing user interfaces, and connected with the specification of the structure editor using the ToolBus.

1.1 Goals of this Thesis

This thesis will attempt to answer the following questions:

- Is it possible to give a formal specification of a (language-independent) structure editor?
- If such a specification can be devised, can we generate executable code from this specification, and will this code run?
- Can we split the structure editor up in different components, and implement/specify these components separately?
- Can we connect all these components to the ToolBus and therefore have them communicate with each other?
- Is this combined system of tools efficient enough to allow us to use it interactively?

1.2 Overview of the thesis

In this chapter we will introduce a number of preliminaries. In Chapter 2 we present and discuss the formal specification of a structure editor. The other components needed, and the interaction between these components is discussed in Chapter 3. The final chapter discusses the results of the work so far, and gives pointers for the future direction of this project.

1.3 Basic Notions

A structure editor is an editor that manipulates structures, much like there are text editors to manipulate text, film editors to manipulate film, and sound editors to manipulate sound. Although it is quite clear what manipulating text means, manipulating structures is less well-defined. We will try to define a number of notions that recur throughout this thesis.

Parse tree Given a context-free syntax definition of language L, a parser for language L and a program written in language L. The result of parsing the program with the parser is a parse tree. A parse tree is a tree with the following properties (following the definition from [ASU86]):

1. The root of the tree is labeled by the start symbol.

2. Each leaf of the tree is labeled by a token or by ϵ (the empty string).
3. Each branching node is labeled by a nonterminal.
4. If A is the nonterminal labeling some branching node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then

$$X_1, X_2, \ldots, X_n \rightarrow A$$

is a production from the context-free syntax definition. Any X_j with $1 \leq j \leq n$ can be either a terminal or a nonterminal symbol. If there is a production $\epsilon \rightarrow A$ then a node labeled A may have a single child labeled ϵ.

Structure editor A structure editor is a program that allows a user to manipulate structures. In the case of this thesis we are exclusively interested in editing the structure of parse trees. However, in other fields other structure editors exist (in chemistry, for instance, structure editors to edit the structure of molecules exist). If we are editing a parse tree we would at least like to perform the following operations: retrieving a specific part of the structure, replacing parts of the tree by other trees, inserting (sub)trees in an existing tree, removing (sub)trees from an existing tree, and, of course, moving around in a tree.

Text editor A text editor is a program that lets a user manipulate text. Similar operations that we would like to have in a structure editor, are also desirable in a text editor; retrieving parts of a text, replacing parts of text, and so on. Examples of text editors are `vi` and `emacs`.

Hybrid text/structure editor A hybrid text/structure editor (or hybrid editor, for short) is the combination of a text and a structure editor. Any structure editor that has even the slightest text editing capabilities should be considered a hybrid editor. As a result of this, most structure editors are really hybrid editors. Hybrid editors can be any combination of a structure and a text editor.

Language independence A system is called language independent if it has no built-in knowledge of a specific language. It is called language parameterized (or generic) if the language is a parameter of the system and upon instantiation with a language definition a language specific system is obtained. A system is called language dependent if the knowledge of a language is hardwired in the system. (These definitions are taken from [BKV96]).

Examples of language independent systems are: the text editor `vi`, the UNIX tool `diff(1)`. The ASF+SDF meta-environment [Kli93] is an example of a generic system. A C-compiler such as `gcc` is an example of a language dependent system.

Language independent structure editor Combining the above notions, a language independent structure editor is a system that can edit the structure of any language, without having any built-in knowledge of such a language.

1.4 Specifying a Structure Editor

Before we present the specification of the structure editor, we will first introduce a number of formalisms we use for this specification.

1.4.1 Parse Trees in AsFix

As we defined in Section 1.3 a structure editor edits the structure of a parse tree. The structure editor we are specifying is a stand-alone tool. Its inputs are a parse tree, and an (edit) operation on this parse tree. In order to provide the structure editor with a parse tree as input, we need some formalism to encode parse trees. This formalism has to provide and hold all the information that is needed for (structure) editing.

Because our aim is a *hybrid* editor (an editor in which both text and structure can be edited) the parse tree formalism has to provide a very close and direct relationship between the parse tree and the source text. Specifically, we want to be able to retrieve the original source text from the parse tree. This means that we also want to be able to retrieve whitespace characters and comments. Saving comment in the parse tree is essential, as the structure editor we are about to specify does not discriminate between editing "useful" text (program code) and "useless" text (comments, layout).

Since the algorithms used in this specification rely heavily on so-called position information (the exact location of each character in the original source) the retrieved source text has to be *identical* to the original source text.

Besides containing the full source text, we also want the parse tree formalism to be generic. The syntactic constructs of a language need to be hidden. The editor only operates on the structure of the parse tree, so it does not have to have knowledge of the syntax of the language a program is written in. However, the formalism must be able to distinguish between certain general concepts that occur in any context-free language. For instance, it has to be able to distinguish between terminal and nonterminal symbols.

We should be able to encode a parse tree in the formalism easily and efficiently. Since this is an interactive system, any delay will be immediately felt by the user. The formalism therefore should not add more than a reasonable amount of overhead to the original parse tree.

We also would like the formalism to possess a mechanism to extend or annotate the parse tree with arbitrary information. We might want to store intermediate editing results, for instance when replacing part of a parse tree with a different parse tree. Another example could be to store position information, in such a way that we can immediately identify the part of the text that corresponds to a certain part of the parse tree.

Finally, the parse tree encoding formalism should be well-defined and be usable by other tools then our structure editor. It should, ideally possess some kind of interface that allows different tools to retrieve exactly the information they need from the parse tree, without having to traverse the parse tree, or having any knowledge of the concrete formalism syntax.

Considering all this, we decided to use AsFix ([BKOV]) as our generic parse tree representation. AsFix is a formalism for representing parse trees, closely related to the language definition formalism Asf+Sdf (see Section 1.4.2).

We will now show that the AsFix formalism satisfies all criteria mentioned above.

All layout and comment information can be preserved in AsFix. Moreover, the parser we use to create parse trees encoded in AsFix *does* preserve all comment and layout. We have written a tool that, given a parse tree in AsFix, returns exactly the source text (see Section 3.2.5).

The AsFix formalism is generic. It labels all nodes in the parse tree with a symbol. These symbols can be any one of: function application (appl), list constructor (list) for the nonterminal nodes in the tree, and any one of: literal (lit), whitespace/comment (w), lexical (lex), variable (var) or meta-variable (metavar) for the terminal symbols. If a node is a branching (nonterminal) node (a function application or a list construction), then the production rule for that nonterminal is also encoded in the node (with symbol prod). The production rules are encoded in the syntax definition formalism SDF [HHKR89]. Any language that can be defined in SDF (any context-free language) can have its parse trees represented in AsFix.

The AsFix formalism allows each node in a tree to be annotated with arbitrary information (which should be encoded in Annotated Term Format, see Section 2.1.1). We could, for instance, replace a subtree of the parse tree with another subtree, and annotate this new subtree with the old subtree. This way, when we decide we would rather have the old tree back, we can immediately put the old subtree back, without having to generate it again, or without needing any external storage. We could also use the annotation mechanism to annotate each node with the text location that corresponds with the beginning (or the end) of the subtree in the original source text.

AsFix has been defined using ASF+SDF in [BKOV]. The definition incorporates a function wf-AsFix and a predicate holds. For any term t the function holds(wf-AsFix, t) returns whether or not t is a valid (well-formed) AsFix term.

Apart from the well-formedness predicate, a number of other functions are defined on AsFix. Most notable are the so-called access functions. These functions allow us to retrieve a named part of a parse tree. For each function symbol that is defined in AsFix, one or more access functions are defined that retrieve the various subterms (or arguments) to this function symbol. For instance, the function symbol for list construction (list) has two access functions associated with it, because any function with symbol list has two (named) parts. These named parts are the production rule that produces this list (symbol: prod), and the elements this list consists of (symbol: elems). Now, if we have an AsFix term t representing a list, we could retrieve the elements of the list by saying t[elems], and the production rule by saying t[prod]. There is a similar mechanism to replace named subtrees. All the function symbols and their corresponding access function names are in the AsFix grammar, in Appendix C.

As stated before, we like to have a mechanism that allows for the easy distribution of parse trees to any number of tools, such as, for instance, a parser, a structure editor and a prettyprinter. As will be described later in this thesis, we use the ToolBus [BK96a, BK96b] (see Section 3.1) for the communication between these tools. The ToolBus uses a term format called ToolBus-terms as its internal data representation. As we will see later, this format bears a very close resemblance to AsFix, which allows us to easily switch from one representation to the other.

And finally we would like the encoding of an AsFix parse tree to be efficient.

As encoding a parse tree in AsFix only changes the labeling of the individual nodes, and not the structure of the tree, the encoding of a parse tree in AsFix is linear. To translate a parse tree from an internal representation as is used in the Asf+Sdf meta-environment to an AsFix representation, we only have to visit each node in the tree once. The translation is $O(n)$. If we use a parser that immediately creates an AsFix parse tree (as its internal representation) then no translation is needed. The parsers generated by the Sdf2 parser generator [Vis97] use AsFix as their native representation.

1.4.2 Asf+Sdf

The structure editor will be specified using Asf+Sdf. As its name implies, Asf+Sdf consists of the combination of the formalisms asf (Algebraic Specification Formalism) and sdf (Syntax Definition Formalism). It is a formalism for the specification of the syntax and semantics of abstract data types and functions on these datatypes. We will give a short description of the formalism.

Modules

An Asf+Sdf specification consists of a number of modules. Each module may specify either syntactical or semantical rules, or both. A module can use other modules by importing them. The result of importing a module is that all visible names of the imported module become available in the importing module. A module can make its declarations invisible for other modules by hiding them.

Sorts

A sort is declared by listing its name in the `Sorts` section of a module. Sort names start with a capital letter. The four predefined sorts `LAYOUT`, `IGNORE`, `REJECT`, and `CHAR` cannot be redeclared.

For each sort in a Sdf specification, a number of derived sorts are automatically available. For sort S these are $S*$, for the list of zero or more elements of sort S and $S+$ for the list of one or more elements of sort S. Lists with separators between the elements can be constructed with the special signatures $\{S\ sep\}*$ and $\{S\ sep\}+$ for a list with zero or more elements, or one or more elements, respectively. *sep* can be any character string separating the elements.

Lexical Syntax

We can define the syntax of the lexicals of a language in a module after the keyword `lexical syntax`. Each lexical function definition consists of a regular expression followed by an arrow (`->`) and a result sort name. The syntax of the lexical tokens of a sort is defined by all lexical functions that have that sort as a result sort.

As an example consider the lexical syntax definition of the sort Natural (the natural numbers):

```
[0-9]+      -> Natural
```

`[0-9]` is a so-called *character class*. It means "any character in the range from `0` to `9`" (any digit). The `+` sign means "one or more of the preceding". So a

Natural is any sequence of (at least one) digits. Other primitives are * for "zero or more of the preceding", and ~ for "any character except the following". For instance

```
~[0-9]*     -> NotNatural
```

specifies any sequence of characters that does not include digits. This sequence can be empty.

The predefined sort LAYOUT is used to specify the characters that should be ignored. Typically this is used to specify whitespace characters, and comment.

```
[ \t\n]     -> LAYOUT
```

specifies that any whitespace, tab, or newline character is to be considered LAYOUT.

Context-free Syntax

The context-free syntax section of a module describes the abstract and concrete syntax of sentences in a language. A context-free syntax definition consists of a set of declarations for context-free functions. Each context-free function declaration consists of zero or more literals, sort names or lists, followed by an arrow and a result sort. All functions with the same result sort define the alternatives for that sort.

As an example:

```
Natural "+" Natural  -> Natural
```

defines the infix function + with two Naturals as input sorts, and one Natural as output sort.

Priorities

The context-free syntax defined in an ASF+SDF specification may be ambiguous; there are sentences that can have more than one associated parse tree. We can solve this by defining priority or associativity rules on functions. This can be done in three ways. The first way is to declare that certain functions have priority over other functions. This is done in the `priorities` section of a module.

The second way is to declare associativity on functions. If there are nested occurrences of binary operators, we can specify in which order they should be associated (if at all).

The third way is by defining associativity on groups of functions. We do that in the `priorities` section too.

Variables

Variables can be declared in the `variables` section of a module. Variables of sort S are declared by a regular expression (as in the lexical syntax declaration) that may not contain sort names, followed by an arrow, and followed by S. Using such a regular expression means that a variable declaration can define an unlimited number of variables. The sort of a variable may be a simple sort or a list.

module Booleans
imports Layout
exports
  sorts Bool
  context-free syntax

true	$\rightarrow$ Bool	
false	$\rightarrow$ Bool	
Bool "$\vee$" Bool	$\rightarrow$ Bool	{**left**}
Bool "$\wedge$" Bool	$\rightarrow$ Bool	{**left**}
"$\sim$" Bool	$\rightarrow$ Bool	
"(" Bool ")"	$\rightarrow$ Bool	{**bracket**}
if Bool then Bool else Bool fi	$\rightarrow$ Bool	

  variables
    Bool [*0-9'*]$*$ $\rightarrow$ Bool
  priorities
    Bool "$\vee$"Bool $\rightarrow$ Bool $<$ Bool "$\wedge$"Bool $\rightarrow$ Bool

equations

[or-1] true $\vee$ *Bool* $=$ true [or-2] false $\vee$ *Bool* $=$ *Bool*
[and-1]true $\wedge$ *Bool* $=$ *Bool*[and-2]false $\wedge$ *Bool* $=$ false
[not-1] $\sim$ false $=$ true [not-2] $\sim$ true $=$ false

[if-1] if true then $Bool_1$ else $Bool_2$ fi $= Bool_1$
[if-2] if false then $Bool_1$ else $Bool_2$ fi $= Bool_2$

Figure 1.1: The ASF+SDF specifications of the Boolean language

Equations

The semantics of the functions declared in the context-free syntax are defined using equations. Equations define equality between terms. The equality of two terms t_1 and t_2 is defined by the *unconditional* equation

$$[TagId] \;\; t_1 = t_2$$

where $TagId$ is a tag to identify this particular equation, and t_1 and t_2 must be terms of the same sort.

A *conditional* equation consists of a number of conditions and a conclusion:

$$[TagId] \;\; \frac{c_1, c_2, \ldots, c_n}{t_1 = t_2}$$

$c_1, c_2, \ldots, c_n$ are the conditions. They are positive (equality) or negative (inequality) equations. If these conditions are met, then t_1 is equal to t_2. There are two other ways to write conditional equations, but as we do not use them in this thesis, we will not explain them here.

As an example, consider the specification in Figure 1.1. After the keyword **equations** are the equations for the functions and ($\wedge$), or ($\vee$), not($\sim$) and the conditional function (if...then...else...fi).

Writing and Compiling a Specification

The ASF+SDF meta-environment [Kli93] facilitates the specification of a language or a datatype. Any context-free language specified in the ASF+SDF meta-environment immediately has a parser [Rek92], a syntax-directed editor [Koo94] and a pretty printer [BV96] generated for it from the SDF part of the specification. The ASF part of a specification is made executable by orienting the equations from left to right and interpreting them as conditional rewrite rules in a term rewriting system.

Pretty printing the specification itself can be done using ToLaTeX[Vis95]. It generates typeset specification as we will use extensively throughout this thesis (For example in Figure 1.1).

The parse trees of any specification can be encoded in AsFix, and dumped to a file. If we have an AsFix representation we can then compile the specification into a stand-alone term rewriting system for the language we just specified. The AsFix files are compiled using `ac(1)` (for AsFix compiler). Through a number of intermediate languages they are compiled into Abstract Reduction Machine code (Arm-code)[1]. This Arm-code is then interpreted by the Arm-interpreter [KW93, KW95], producing a stand-alone term rewriting system.

[1]Created by P.A. Olivier and M.G.J. van den Brand at the Programming Research Group of the University of Amsterdam

Chapter 2

Structure Editing

In this chapter we will present the reader with a formal specification of a structure editor. We will define the various datatypes needed for such a specification first. In the second part of this chapter we will specify the algorithms used for editing the structure of parse trees.

2.1 Datatypes for Structure Editing

The structure editor is implemented as an ASF+SDF specification.

The specification of the structure editor revolves around the datatype Editor. An Editor is built from several subtypes (see Figure 2.1). We will first discuss these subtypes, and in doing so explain the Editor type.

2.1.1 Annotated Term Format

Since a structure editor operates on parse trees, our Editor datatype should incorporate some encoding of a parse tree. As discussed in Section 1.4.1, we use AsFix as our parse tree representation. AsFix is in fact an instantiation of a generalized term format called Annotated Term Format, or ATF for short [BKV96]. The module ATerms (see Section A.1) defines the sort ATerm; the basic term for the ATF. The structure of ATerms is defined in this module too. ATerm is a parameterized datatype, only after it is parameterized with specific function symbols does it have a concrete syntax. Most of the operations defined on ATerms are also parameterized. For instance, the access function defined on ATerms:

```
ATerm "[" AFun "]"       ->  ATerm
```

(The function that retrieves the part named AFun from ATerm) has no equations associated with it. Only in the module defining AsFix parse trees (see [BKOV]) the equations for the different access functions are specified.

We use the sort ATerm throughout this specification, assuming that it is instantiated with the function symbols as defined in AsFix. To enforce this assumption, we should call on the well-formedness predicate each and every time an ATerm is used. Apart from making the specification unreadable, this would also slow down the rewriting process significantly. It is, however, a very

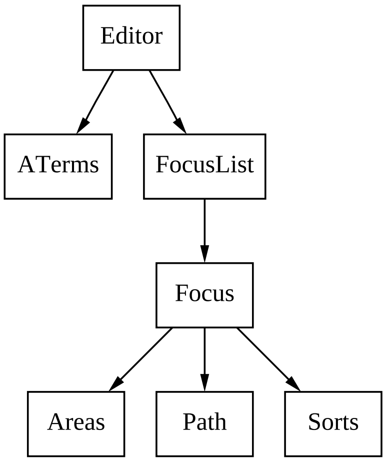

Figure 2.1: The datatype Editor, and the different other datatypes that it incorporates.

good check to call upon the well-formedness checker to make sure that the ATerm coming in (the parse tree as created by the parser), and the ATerm going out (this same parse tree after the requested operations are performed on it by the structure editor) of this specification are indeed AsFix terms. However, once we have established that the parse trees as provided by the parser are valid AsFix terms, and that our specification of the structure editor does not corrupt them, we can remove the well-formedness check.

2.1.2 Multiple Focuses

An important element of any structure editor is the Focus. The Focus is usually described as "the (sub-)term of interest". It is a pointer to a term on which the next editing operation will be performed. In a hybrid editor such as ours, the focus is usually the only place where text can be manipulated at a character level. Only the term in the focus can be syntactically incorrect at any given time. Let us consider the following program, written in C. The example is taken from [KR88].

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

The underlined part of the program constitutes the current focus. A user now decides to edit the program, so that it exits with a certain exit code after printing.

If a user decides to edit outside a focus, one of the following three actions need to be taken. With each action is the state of the program and focus(es) after this action.

1. The structure editor has one focus. Every time an operation is requested outside the current focus, the text in the current focus is first parsed. Then a new focus is created at the requested location.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
    exit(1);
}
```

2. The structure editor has one focus, only now, when an operation is requested outside the focus, the focus is extended to hold the new text insertion point, as well as the old focus.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
    exit(1);
}
```

3. The structure editor can have more than one focus. If an operation is requested outside the current focus, then the current focus becomes a secondary focus. A new current or primary focus is then created on the requested location. If a (secondary) focus existed on the requested location then that focus is promoted to the current focus. There can only be one current or primary focus at any one time.

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
    exit(1);
}
```

The structure editor we are specifying is one with multiple focuses. It will follow the scenario as described in the third item of the list above.

Allowing Multiple Focuses

The idea of allowing multiple focuses at once is an experimental one. The advantages of having more than one focus are

- Less parsing is needed. In a single focus structure editor, whenever a user decides to edit outside the current focus, the editor simply increases the focus to hold both the old focus and the new insertion point. This can lead to situations where the whole (program) text needs to be reparsed, although only two nodes of the parse tree have been edited.

- Parsing can be delayed. When compared to the first strategy as described above, we do not have to parse the focus each time we want a new focus. We just create a new focus and save the old one as a secondary focus.

There are however, some serious drawbacks on having multiple focuses. It still remains to be seen whether or not having more focuses is really more efficient than having one. The drawbacks are

- Too little parsing is performed. If we start parsing individual parse tree nodes, the chances of introducing a parse error in this single node are rather big. As an example, we have the following expression as a parse tree (from the While language, see Appendix B): `1 + 1`. This is a instantiation of the production `Exp + Exp -> Exp`. If a user decides to edit the first "`1`" and changes it into "`10`", then our focus strategy works really well. The focus holds the text "`10`" and expects that to be a term of sort `Exp`. However if a user decides for some reason or other, to put brackets around this expression, then we have a problem. The first bracket will be put in the focus that holds the first `Exp`. It will now hold the text "`(10`" and expect it to be a `Exp`. This will result in a parse error. The second bracket is part of the focus containing the whitespace and layout characters following the last `Exp`. As we don't allow focuses to contain just whitespace, this focus will be raised one node in the parse tree, to hold the second `Exp` of the term. The text in this focus will be "`1)`", which, again, results in a parse error.

  Had we simply increased the focus when editing the second bracket, then the focus would have contained the text `(10 + 1)` and expected that to be of sort `Exp`. This would have been a correct assumption.

- It is expensive to use more than one focus. We need more memory to store the extra focuses. As we can see from the module below every focus occupies on average the same amount of memory; differences occur in the length of the path of a focus, and the length of the sort name of the focus. The length of the focus itself (the number of characters it corresponds to in the source text) has no influence on the length of the datastructure focus as stored in memory.

  Calculating the datastructure for each new focus introduced is also more expensive than simply enlarging the existing focus. For each new focus we create, we have to check whether or not there is an existing focus that coincides with this new focus (has a root node that is a child of the root node of the new focus).

We will present a number of strategies to solve these problems, but will first specify the syntax for Focus.

The Module FocusSyntax

module FocusSyntax
imports Path-construction$^{2.2.6}$ AreaSyntax$^{2.1.6}$ Sorts$^{2.1.5}$
exports
  sorts Focus FocusList

The Focus Datatype

A focus is a 3-tuple $\langle \Pi, \Sigma, A \rangle$, with Π a Path, Σ a Sort and A an Area as defined below.

context-free syntax
“<” Path “,” Sort “,” Area “>” → Focus

(See for the definitions of Path, Sort and Area the sections 2.1.4, 2.1.5 and 2.1.6, respectively).

The FocusList Datatype

A FocusList is a list of focuses, separated by semicolons. We want to mark the *current* focus, i.e., the focus that contains the cursor. For now we assume that the first focus in the list of focuses is the current one, thus eliminating the need for a special (marking) symbol.

context-free syntax
“[” {Focus “;”}∗ “]” → FocusList

Variables for the Focus Datatypes

The variables below are used throughout this specification.

variables
“ϕ” [*0-9′*]∗ → Focus
“ϕ” “∗” [*0-9′*]∗ → FocusList
fl[*0-9′*]∗ → {Focus “;”}∗

Access Functions for the Focus Datatypes

We define three access functions on Focus. Note that the syntax for these access functions (Sortname[namedpart]) is taken from the Annotated Term Format (Section 1.4.1). This notation is used throughout this thesis. These access functions each return us one element of the 3-tuple focus. This way, the actual syntax of a focus is transparent to the rest of the implementation.

context-free syntax
Focus “[” “path” “]” → Path
Focus “[” “sort” “]” → Sort
Focus “[” “area” “]” → Area
FocusList “[” “head” “]” → Focus
FocusList “[” “tail” “]” → FocusList

Apart from the access functions there are also functions that *replace* a named part of a term. These too will be used throughout the specification.

context-free syntax
Focus “[” “area” “:=” Area “]” → Focus
Focus “[” “sort” “:=” Sort “]” → Focus
Focus “[” “path” “:=” Path “]” → Focus

And there is a *creation* function for the Focus, to hide its actual syntax.

context-free syntax
“focus” “(” Path “,” Sort “,” Area “)” → Focus

The equations for the access functions that have been defined follow here:

equations

[fp]	$\langle\pi, \sigma, \alpha\rangle$[path]	$= \pi$
[fs]	$\langle\pi, \sigma, \alpha\rangle$[sort]	$= \sigma$
[fa]	$\langle\pi, \sigma, \alpha\rangle$[area]	$= \alpha$
[rf]	$\langle\pi, \sigma, \alpha\rangle$[area $:= \alpha'$]	$= \langle\pi, \sigma, \alpha'\rangle$
[rs]	$\langle\pi, \sigma, \alpha\rangle$[sort $:= \sigma'$]	$= \langle\pi, \sigma', \alpha\rangle$
[rp]	$\langle\pi, \sigma, \alpha\rangle$[path $:= \pi'$]	$= \langle\pi', \sigma, \alpha\rangle$
[sf]	focus(π, σ, α)	$= \langle\pi, \sigma, \alpha\rangle$
[flh-n]	$[\phi;$ *fl*][head]	$= \phi$

The head of an empty list is undefined.

[flt-n]	$[\phi;$ *fl*][tail]	= [*fl*]
[flt-1]	[][tail]	= []

2.1.3 Parsing Multiple Focuses

As a result of the multiple focus strategy described in the previous section, we also have an "incremental parsing" strategy. We would ideally parse each substring in the source text that corresponds to a focus as a separate program. However, as stated in the previous section, this is sometimes impossible. Apart from the problem of the brackets, other problems can occur. For instance at the end of a list construct. Suppose that we have a language that has a list construct that can contain Naturals, separated by semicolons and started and ended by the keywords `begin` and `end`. In SDF that would look like this:

```
"start" { Nat ";" }+ "end"   ->  List
```

Now suppose we have a term `start 1;2 end` of sort `List`. We now add the characters `;3` to the end of the list. The focus now contains the text string `2;3`. The sort of the focus is `Nat`. This will result in a parse error.

As you might have noticed, this particular problem is not specific to multi-focus structure editing. The same thing would have happened with a single focus structure editor.

There are a number of strategies we can use to try and minimize the number of "false parse errors". False parse errors are errors as described above, they occur when the whole source text is syntactically correct, but, because the focus is positioned erroneously, the source text of a single focus is syntactically incorrect.

To correct for the parse error that seems to occur when brackets are introduced, we can use this strategy: When there is more than one focus, and a parse error occurs, we can unify all the different focuses (by creating a new focus whose root node is a parent of all existing focuses). We then parse the source text of the new focus.

This does not guarantee that we will not get a false parse error again. Of course, the first parse error could have been the result of a genuine syntactic error, in which case any strategy is useless.

If there is only one focus, and a parse error occurs, it is possible that we have an occurrence of the `List` example, above. In this case, we will move the focus up one node (by using move-focus, see Section 2.2.2). This eliminates most

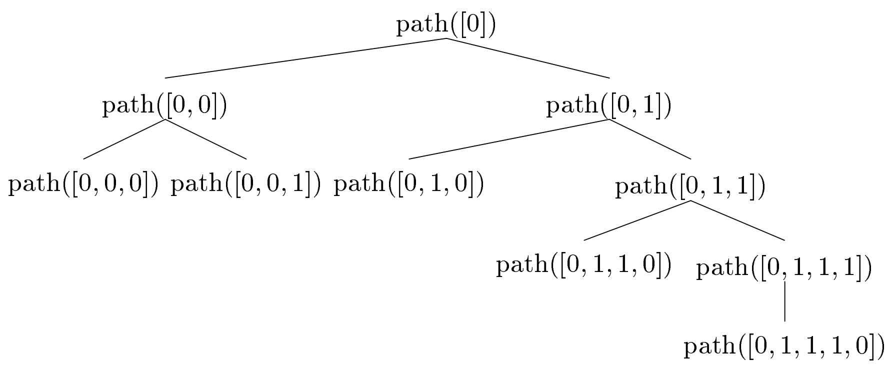

Figure 2.2: Examples of paths in a tree

errors that occur in lists. Again, there is no way to predict whether or not the parse error was the result of a *real* error or the result of a misplaced focus. If parsing is still unsuccessful after moving the focus up, we can repeat the process and move up another node (until we reach the root of the parse tree). This is the approach taken in [Bra92].

As these strategies do not guarantee success, we mainly propose to use them as a compromise. In some cases the use of these strategies relieves the user from the task of having to set or reset the focus himself. However, if there is a real parse error, it will now take longer to detect it, because each term will be parsed twice in case of an error. There is also the added cost of the structure editing operations that are needed (moving the focus, grouping all focuses into one).

Yet another approach would be to do nothing if a parse error occurs. We could signal the user that something went wrong, and leave it to him to decide whether there was a misplaced focus (in which case he can enlarge it himself) or a real syntactic error. This is the approach that is taken by GSE [Koo94].

We hope to create a system where we can run experiments with these (and possibly other) techniques for resolving false parse errors. The type of strategy that we choose to follow depends mainly on the way a user inputs a source text. Therefore the system should be configureable in such a way that a user can turn on or off specific features she likes or does not like. Through this configuration, a user should also be able to enforce the structure editor to use only one focus at a time. This way, we might find what strategy gives the best results.

2.1.4 Paths in a Tree

A Focus points to a specific subtree of a parse tree. To address different subtrees we need to have a mechanism to identify these specific subtrees. Every node (vertex) in the tree has a unique Path associated with it. The root vertex has 0 as its Path, while the children of the root have $(0, 1), (0, 2), \ldots, (0, n)$ as a Path, respectively. Paths to subtrees are encoded as an ATerm. The advantage of encoding Paths as ATerms is that, when encoding Paths in trees encoded as an ATerm, the Path can be used directly as an annotation on a specific subterm. Recall that ATerms can be annotated with an ATerm (see Section A.1). Furthermore, since ATerms and ToolBus terms are equivalent(see Section 3.1), this notation allows Paths to be exported from the specification to any tool that might be interested.

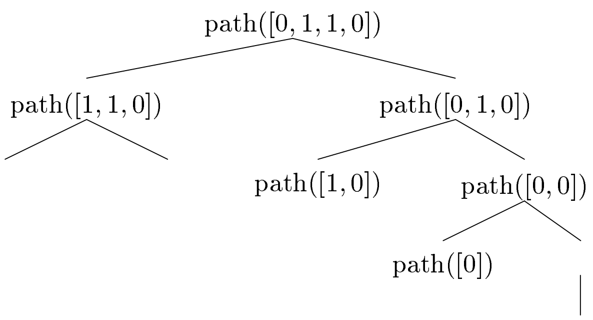

Figure 2.3: When moving through a tree, to a certain node, we deconstruct the Path pointing to that node. Here, we have go to path([0, 1, 1, 0]). From the root, we move to the first child by removing the first element of the Path. We now follow path([1, 1, 0]) in the same manner, until the Path we are following is path([0]). We have arrived at the requested location. Note that the tree in this figure is identical to the tree in Figure 2.2. In Figure 2.2 the Paths are given relative to the root, whereas in this figure thePaths are given relative to the target location.

Paths are now encoded as a term with function symbol path, followed by a list of children, such that the above Path will now be path($[n_1, n_2, \ldots, n_i]$). See Figure 2.2 for an example of Paths in a tree.

The Paths Module

module Paths
imports AsFix-Utilities$^{A.2}$ Count
exports
  sorts Path
  context-free syntax
    "path" → AFun

A Path is defined as a function with function symbol "path" and a list of ATerms as its argument. According to the following syntax definition, a Path is any ATerm. However, in the ASF specification we assume that a Path is a *specific* ATerm. The actual definition therefore follows below, from the equations. Again, the actual syntax of a Path should not be used. We provide a number of access functions to hide the syntax from the implementation.

  context-free syntax
    ATerm → Path

Variables for Paths

  variables
    "π"[*0-9′*]∗ → Path

Path Creation

We define a single Path creation function:

context-free syntax

"root" $\rightarrow$ Path

This is a Path to the root (the top node) of a tree.

equations

[r] root = path([0])

Predicates on Paths

context-free syntax

is-leaf-path(Path) $\rightarrow$ Bool
empty(Path) $\rightarrow$ Bool
head-zero(Path) $\rightarrow$ Bool
is-child-path(Path, Path) $\rightarrow$ Bool

All navigation functions work by stripping of parts of the tree. Consequently, if you start somewhere in a tree, the last node you will encounter is a leaf node.

[is-l-t] is-leaf-path(path([0])) = true

By default, a path is not to a leaf node.

[is-l-f] is-leaf-path(π) = false **otherwise**

Only the empty Path is an empty Path.

[pe-t] empty(path([ ])) = true
[pe-f] empty(π) = false **otherwise**

If a Path starts with zero, the function head-zero will return true. When moving through a list of siblings in a tree, we decrease the head of the Path until it is zero. Then we either have reached our destination (is-leaf-path(π) = true), or we are at the point where we have to move to the child of the current node (head-zero(π) = true), as shown in Figure 2.3.

[hz-t] head-zero(path([0, t^+])) = true
[hz-f] head-zero(path([t^+])) = false **otherwise**

The function is-child-path checks whether its first argument is a child (i.e. a Path to a node lower in the tree) of its second argument. In Figure 2.2, for instance, path([0,1,1,1,0]) is a child path([0,1]), and all nodes are children of path([0]).

It assumes that the value "empty-path" ([]) is not used as an actual Path, but only as a stop-condition while performing operations on a Path. Were path([]) to be a meaningful Path then this function fails to perform as intended.

[child-t] is-child-path(path([t^+]), path([ ])) = true

$$\text{[child-n]}\ \frac{\text{empty}(\pi_2) = \text{false},\quad \pi_1[\text{head}] = \pi_2[\text{head}]}{\text{is-child-path}(\pi_1, \pi_2) = \text{is-child-path}(\pi_1[\text{tail}], \pi_2[\text{tail}])}$$

[child-f] is-child-path(π_1, π_2) = false **otherwise**

Moving through a Tree

context-free syntax
decrease-head(Path) $\rightarrow$ Path

Since all Paths are relative to the current node, we need to be able to decrease the head of a Path as we move to the next node. For example, we want to go to path($[0,3]$) starting at the root node of a tree. We will then go to the first child of the root, such that the relative Path to our target becomes path($[3]$). Next we move to the first sibling of the current node, and we decrease the head of the relative Path to the target, resulting in path($[2]$). We repeat this procedure until we reach path($[0]$), which, as the predicate tells us, is the "leaf" (or subtree) of interest. (The name *leaf* could be misleading here, as this node is the root of the targeted subtree, and not necessarily a leaf node).

[Path4'1] decrease-head(path($[t_i,\, t^+]$)) = path([pred(t_i), t^+])
[Path4'2] decrease-head(path($[t_i]$)) = path([pred(t_i)])

Access Functions on Paths

These functions select specific parts of a Path. Note that throughout this specification the square brackets (sortname "[" literal "]") denote an access function, i.e., a function that returns the named (by literal) part of the sortname.

context-free syntax
Path "[" "head" "]" $\rightarrow$ AInt
Path "[" "tail" "]" $\rightarrow$ Path

The function head returns the first element of the path-list. Note that these functions are undefined if Path is the empty Path.

[Path2'1] path($[t_i,\, t^+]$)[head] = t_i
[Path2'1] path($[t_i]$)[head] = t_i

The function get-tail-path returns the tail of a path list (all elements except for the head). It returns the empty Path if the original Path only has one element.

[Path5'1] path($[t_i,\, t^+]$)[tail] = path($[t^+]$)
[Path5'1] path($[t_i]$)[tail] = path([ ])

Constructing Paths

We construct Paths with the arrow functions defined below. Each arrow takes one step in the direction it points to. For instance, a Path to node (0,3,2,1) can be created with root$_{\downarrow\rightarrow\rightarrow\rightarrow\downarrow\rightarrow\rightarrow\downarrow\rightarrow}$. The up-arrow ($_\uparrow$) is a Path destruction function. It removes the last element of the Path.

context-free syntax
Path "$_\downarrow$" $\rightarrow$ Path
Path "$_\uparrow$" $\rightarrow$ Path
Path "$_\rightarrow$" $\rightarrow$ Path
Path "$_\leftarrow$" $\rightarrow$ Path

These are Path construction functions. $\mathsf{Path}_{\downarrow}$ adds a new level of children to an existing Path.

[Path7'1] $\mathsf{path}([t^+])\downarrow = \mathsf{path}([t^+] +\!\!+ [0])$

$\mathsf{Path}_{\uparrow}$ modifies an existing Path, such that it now is a Path to the parent of the node it originally pointed to.

[Path8'1] $\pi\uparrow = \mathsf{remove\text{-}last\text{-}element}(\pi)$

$\mathsf{Path}_{\rightarrow}$ moves the Path to the next sibling.

[Path9'1] $\pi\rightarrow = \mathsf{increase\text{-}last\text{-}element}(\pi)$

$\mathsf{Path}_{\leftarrow}$ moves the Path to the previous sibling.

[Path10'1] $\pi\leftarrow = \mathsf{decrease\text{-}last\text{-}element}(\pi)$

Auxiliary Functions

The functions that follow here are hidden functions, they cannot be called outside the module Paths. Instead the arrow functions defined above should be used.

hiddens

context-free syntax

remove-last-element(Path) $\rightarrow$ Path
decrease-last-element(Path) $\rightarrow$ Path
increase-last-element(Path) $\rightarrow$ Path

The function remove-last-element is the reverse of get-tail-path, that is, it returns all elements of the Path except for the single last element.

[Path6'2] $$\frac{\mathsf{remove\text{-}last\text{-}element}(\mathsf{path}([t_{i2},\, t_1^+])) \;=\; \mathsf{path}([t_2^+])}{\mathsf{remove\text{-}last\text{-}element}(\mathsf{path}([t_{i1},\, t_{i2},\, t_1^+])) \;=\; \mathsf{path}([t_{i1},\, t_2^+])}$$

[Path6'3] $\mathsf{remove\text{-}last\text{-}element}(\mathsf{path}([t_{i1},\, t_{i2}])) = \mathsf{path}([t_{i1}])$
[Path6'2] $\mathsf{remove\text{-}last\text{-}element}(\mathsf{path}([t_i])) = \mathsf{path}([t_i])$

decrease-last-element is equivalent to decrease-head in the sense that it decreases a single element of the Path. Only, as its name suggests, here it is the single last element, instead of the first.

[Path7'1] $$\frac{\mathsf{decrease\text{-}last\text{-}element}(\mathsf{path}([t_1^+])) \;=\; \mathsf{path}([t_2^+])}{\mathsf{decrease\text{-}last\text{-}element}(\mathsf{path}([t_i,\, t_1^+])) \;=\; \mathsf{path}([t_i,\, t_2^+])}$$

[Path7'2] $\mathsf{decrease\text{-}last\text{-}element}(\mathsf{path}([t_i])) \;=\; \mathsf{path}([\mathsf{pred}(t_i)])$

increase-last-element increases the last element of a Path.

[Path7'1] $$\frac{\mathsf{increase\text{-}last\text{-}element}(\mathsf{path}([t_1^+])) \;=\; \mathsf{path}([t_2^+])}{\mathsf{increase\text{-}last\text{-}element}(\mathsf{path}([t_i,\, t_1^+])) \;=\; \mathsf{path}([t_i,\, t_2^+])}$$

[Path7'2] $\mathsf{increase\text{-}last\text{-}element}(\mathsf{path}([t_i])) \;=\; \mathsf{path}([\mathsf{succ}(t_i)])$

2.1.5 The Sort of a Subtree

The sort of a subtree is encoded as an ATerm. This information is already available from the actual subtree itself, the sort of a subtree is encoded in the root of the subtree. Keeping this information in the Focus datatype as well simplifies the interaction with both the parser (see Section 3.2.3) and the text editor (see Section 3.2.2).

The Module Sorts

module Sorts
imports AsFix-Utilities$^{A.2}$
exports
  sorts Sort

Syntactically a Sort can be any ATerm. Throughout this specification, however, we assume a Sort to be the function symbol sort applied to the sort name, i.e. sort("Integer").

Following the motivation for the "syntax definition", it can be left out altogether, and plain ATerms can be used wherever Sorts are used now. This would not change the functionality, and could even (as it eliminates a non-terminal symbol from the specification) be considered "better". Unfortunately, it would render the specification unreadable.

context-free syntax
  ATerm $\rightarrow$ Sort
  "error" $\rightarrow$ Sort
  invalid $\rightarrow$ Sort
variables
  "σ"[*0-9'*]$*$ $\rightarrow$ Sort

2.1.6 Area occupied by Subtree

This is the part of the Focus where the relation between parse tree (as returned by a parser) and text string (as typed by the user in a text editor) becomes explicit.

For each Focus, the position of the characters that correspond to this particular subtree is saved in the Focus. As with Sort, keeping this information in the Focus datatype is essentially redundant, as it is also encoded in the parse tree itself. In keeping these data in the Focus datatype the complexity of interacting with the outside world (see Section 3.2.2) is reduced considerably.

The Module AreaSyntax

module AreaSyntax
imports AsFix-Utilities$^{A.2}$
exports
  sorts Area

The Area Datatype

An Area is a tuple (s, l), $s, l \in \mathbb{N}$, with s the starting point of the Focus, and l the length of the Focus. The *starting point* is the character position on which the character string that forms the subtree of the Focus begins. The *length* of the Focus means here the number of characters this subtree consists of. So, $s + l$ is the *last* character of the subtree.

context-free syntax
"(" Int "," Int ")" $\rightarrow$ Area

Variables for Area

variables
"α" [0-9′]* $\rightarrow$ Area

Access Functions for Area

The above syntax can at any point in time be changed to something else. Most notably it could be written as an ATerm, to facilitate all kinds of predefined operations on ATerms. Because of this it is undesirable to use the Area datatype directly. The following access functions are provided to modify and retrieve information from an Area without having to know the actual syntax of this type.

context-free syntax

Area "[" "length" "]"	$\rightarrow$ Int
Area "[" "start" "]"	$\rightarrow$ Int
Area "[" "length" ":=" Int "]"	$\rightarrow$ Area
Area "[" "start" ":=" Int "]"	$\rightarrow$ Area
area(Int, Int)	$\rightarrow$ Area

The first two functions are information retrieval functions. start returns the starting point of the area, and length returns the length of the area.

equations
[as] $(i_1, i_2)[\mathsf{start}] = i_1$
[al] $(i_1, i_2)[\mathsf{length}] = i_2$

The next two are modification functions. They replace either the length, or the starting point of an area.

[rl] $(i_1, i_2)[\mathsf{length} := i'] = (i_1, i')$
[rs] $(i_1, i_2)[\mathsf{start} := i'] = (i', i_2)$

And finally, the creation function is introduced. It takes two integers and creates a new Area

[sa] $\mathsf{area}(i_1, i_2) = (i_1, i_2)$

Observe that the sort Int is used here for the integers, and not the sort AInt as used in Section 2.1.4. This is because on the sort AInt no standard arithmetic (addition, subtraction) is defined but only the predecessor and successor functions. On the sort Int the standard arithmetic functions *are* defined. This situation is obviously undesirable, and, in the future, these two sorts should be merged into one.

2.1.7 The Editor Datatype

Now that we have defined the various subtypes of the editor datatype, it is time to define the Editor datatype itself.

The Module Editor

module Editor
imports Position-information$^{2.2.5}$ Focuses$^{2.2.4}$
exports
  sorts Editor

The Editor Datatype

A parse tree and a list of focuses form an Editor.
  context-free syntax
    "(" ATerm "," FocusList ")" → Editor
  variables
    "ϵ" *[0-9′]*∗ → Editor

Access functions for Editor

Access, replacement and creation functions for the Editor:
  context-free syntax
    Editor "[" "tree" "]" → ATerm
    Editor "[" "focus" "]" → FocusList
    Editor "[" "tree" ":=" ATerm "]" → Editor
    Editor "[" "focus" ":=" FocusList "]" → Editor
    editor(ATerm, FocusList) → Editor
equations

[gpt] $(t, \phi^*)[\text{tree}] = t$
[gf] $(t, \phi^*)[\text{focus}] = \phi^*$
[rt] $(t, \phi^*)[\text{tree} := t'] = (t', \phi^*)$
[rf] $(t, \phi^*)[\text{focus} := \phi^{*\prime}] = (t, \phi^{*\prime})$

This is the Editor creation function. It is used to hide the actual syntax of the Editor datatype

[se] $\text{editor}(t, \phi^*) = (t, \phi^*)$

2.2 Functions for Structure Editing

2.2.1 An Overview

Figure 2.4 gives an overview of the modules of the specification of a structure editor. The arrows in this figure show which modules are imported by other modules. An arrow from module A to module B means that module A imports module B.

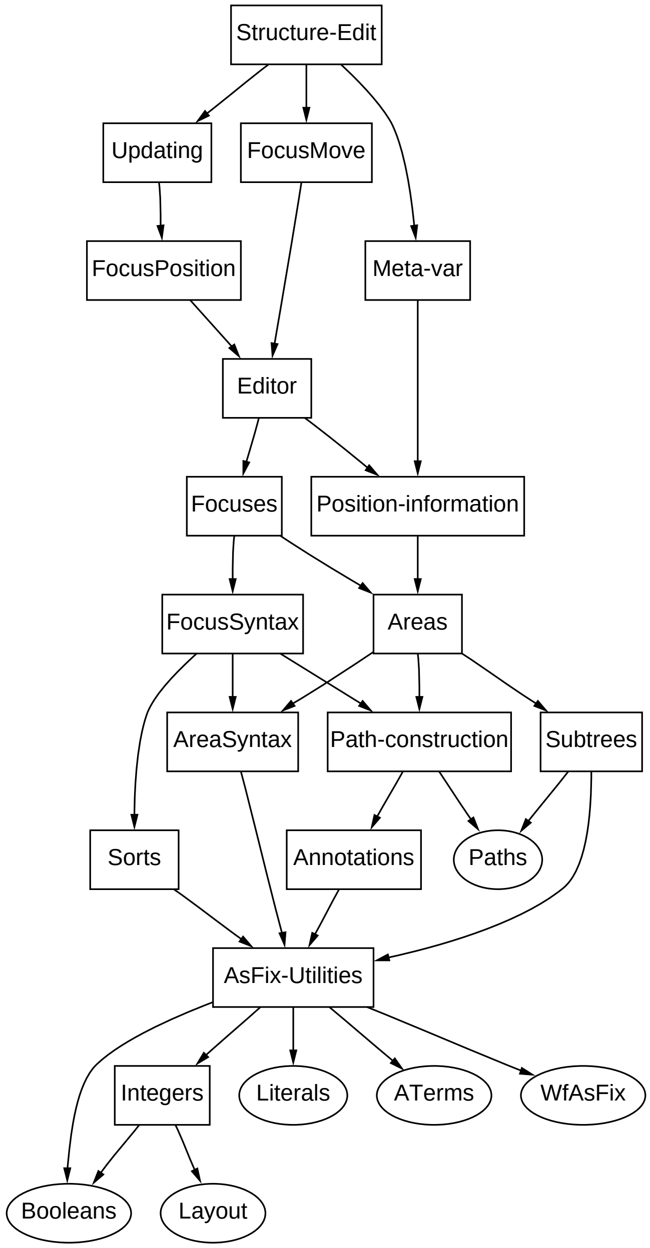

Figure 2.4: The import structure of all the modules discussed here for the structure editor.

We will now give a short overview of all modules and what is specified in them. The superscripted number with each module name is the section in which this module is presented. This section is mainly meant to be a reference section. All concepts that are introduced here will be explained in later sections.

The top module is Structure-Edit[2.2.2]. It specifies all the access functions for the structure editor. It is the only entry point for this specification.

All functions that update the Editor datatype as a result of an action by the user are in the module Updating[2.2.3].

All operations on the Focus datatype are specified in Focuses[2.2.4], except for the functions that calculate the position of a Focus in the source text; these are in module FocusPosition[2.2.8]. The functions needed for moving a Focus through the tree are in module FocusMove[2.2.9]. The syntax for the Focus datatype itself, and access functions on this datatype are in module FocusSyntax[2.1.2].

The Editor datatype is specified in the module Editor[2.1.7].

Parse trees for meta-variables are specified in the module Meta-var[A.3].

The operations that annotate a parse tree with their position information, and the operations that manipulate this position information are in the module Position-information[2.2.5].

We have defined some generic operations on annotations. These are in the module Annotations[A.4].

A Focus encloses a specific Area. These Areas are specified as a separate datatype. Their syntax is in the module AreaSyntax[2.1.6], and operations on them are specified in the module Areas[A.5].

Another element of each Focus is its Path (the location of the Focus in the parse tree). The syntax of Paths is defined in the module Paths[2.1.4]. The functions that construct a Path to a specific location in a parse tree are specified in module Path-construction[2.2.6].

If we have a Path and a parse tree, we can retrieve the subtree to which this Path leads. The function that retrieves such a subtree is specified in the module Subtrees[2.2.7].

Every subtree has a specific Sort. A Sort is also an element of a Focus. The syntax for Sorts is specified in the module Sorts[2.1.5].

The module AsFix-Utilities[A.2] provides all modules with a uniform interface to a number of modules that are needed by almost all other modules. Apart from acting as a "library" module, it also implements some conversion functions on the parse tree format, ATerm. The syntax for these parse trees is in the module ATerms[A.1].

The booleans (Bool) have already been specified, as an example, in the module Booleans[1.4.2].

The other modules in Figure 2.4 will not be specified in this thesis. These modules specify the literals (module Literals), the integers (module Integers), the module that defines the layout for this specification (module Layout) and the module that specifies the well-formedness property of the parse tree format (module WfAsFix).

2.2.2 Structure Editing Functions

Most of the functions defined for structure editing are internal functions. They should be hidden from the outside world. Currently, it is not quite clear if and how an outside-world interface of this specification (for instance via the

ToolBus (Section 3.1)) would preserve the hiding of functions. Therefore a special module is created, called Structure-Edit. Only the functions defined in this module should be used externally. All other modules in this specification can be considered internal (i.e. not accessible from the outside world) to the specification.

We will first define and describe the exported functions. Following this definition, all internal (auxiliary) functions will be defined.

The Module Structure-Edit

module Structure-Edit
imports Updating$^{2.2.3}$ Meta-var$^{A.3}$ FocusMove$^{2.2.9}$

Inserting and Deleting Characters

We are specifying a so-called *hybrid* editor (as stated before), therefore we need to have a way to edit text (instead of just structure). The function insert-chars allows us to insert any character string, anywhere in the text. Its second argument is the location of the insertion (the number of characters from the start of the text). If no Focus is present at that location, one will be created.

The function delete-char deletes exactly one character from the given location.

context-free syntax
insert-chars(Editor, Nat, Literal) → Editor
delete-char(Editor, Nat) → Editor

For text insertion there are two possibilities:

1. A Focus exists, so we only need to update the editor with the length of the string that is inserted. That is, the subtree which the Focus is on will be extended to include the extra characters.

   equations

   $$[\mathsf{ic}]\ \frac{\mathsf{f\text{-}empty}(\epsilon[\mathsf{focus}]) \ = \ \mathsf{false}}{\begin{array}{l}\mathsf{insert\text{-}chars}(\epsilon,\ \mathit{n},\ \mathit{lit}) \ = \\ \mathsf{update\text{-}edit}(\epsilon[\mathsf{tree}],\ \epsilon[\mathsf{focus}],\ \mathit{n},\ \mathsf{length}(\mathit{lit}))\end{array}}$$

2. There is no Focus. Now we need to create a new Focus, and, after that, extend the subtree that the Focus is on.

   $$[\mathsf{ic\text{-}0}]\ \frac{\mathsf{f\text{-}empty}(\epsilon[\mathsf{focus}]) \ = \ \mathsf{true}}{\begin{array}{l}\mathsf{insert\text{-}chars}(\epsilon,\ \mathit{n},\ \mathit{lit}) \ = \\ \mathsf{update\text{-}edit}(\epsilon[\mathsf{tree}],\ [\mathsf{create\text{-}focus}(\epsilon[\mathsf{tree}],\ \mathit{n})],\ \mathit{n},\ \mathsf{length}(\mathit{lit}))\end{array}}$$

Deleting characters is similar to inserting characters. The difference is that characters are deleted one at a time.

1. If a Focus exists, update-edit is called with three arguments (instead of four, as with insert-chars). This implies that at the current character position

exactly one character will be deleted.

$$
[\mathsf{dc}]\ \frac{\mathsf{f\text{-}empty}(\epsilon[\mathsf{focus}]) \ = \ \mathsf{false}}{\begin{array}{l}\mathsf{delete\text{-}char}(\epsilon,\ n) \ = \\ \mathsf{update\text{-}edit}(\epsilon[\mathsf{tree}],\ \epsilon[\mathsf{focus}],\ n)\end{array}}
$$

2. If there is no Focus yet, first a Focus is created, and then the current character is removed.

$$
[\mathsf{dc\text{-}0}]\ \frac{\mathsf{f\text{-}empty}(\epsilon[\mathsf{focus}]) \ = \ \mathsf{true}}{\begin{array}{l}\mathsf{delete\text{-}char}(\epsilon,\ n) \ = \\ \mathsf{update\text{-}edit}(\epsilon[\mathsf{tree}],\ [\mathsf{create\text{-}focus}(\epsilon[\mathsf{tree}],\ n)],\ n)\end{array}}
$$

Replacing a Focus

The function replace-focus replaces the term which the current (primary) Focus points to with the specified term. No typechecking is done here. The new term is inserted in the tree without any restrictions. That is, it is perfectly acceptable to use this function to replace a subterm of sort A with a subterm of sort B (with B not a subsort of A). Because the editor is language independent, it does not have any knowledge whatsoever of (sub)sorts. The only check that could be made here is whether two terms have an identical sort. If, for instance A is a subsort of B, any term of sort B could be replaced with a term of sort A, but the structure editor does not have this information. Therefore, any form of typechecking needs to be done outside the structure editor. The ToolBus architecture lets us easily add a typechecker. The ToolBus script that controls the editor ensures that before each structure editing operation the term to be replaced and the replacement term are of compatible type (see Section 3.2.4).

Calling replace-focus on an editor that does not have a Focus (or rather an empty Focus list) will return the original editor. The replacement term will be lost in this situation. This should never happen, as either the ToolBus script or the text-editor should make sure there *is* a Focus before trying to change it.

context-free syntax
replace-focus(Editor, ATerm) $\rightarrow$ Editor

When replacing the current Focus with a specific term we first bring the tree up-to-date, using upd-term. Then we need to update the Focus, or rather, the sort and the area of the Focus, since the path of the Focus will remain the same by definition. This update is done in the last condition.

$$
[\mathsf{rf}]\ \frac{\begin{array}{rcl}\epsilon[\mathsf{tree}] &=& t, \\ \epsilon[\mathsf{focus}] &=& \phi^*, \\ \mathsf{f\text{-}empty}(\phi^*) &=& \mathsf{false}, \\ \phi^*[\mathsf{head}] &=& \phi, \\ \phi^*[\mathsf{tail}] &=& \phi^{*\prime}, \\ \mathsf{upd\text{-}term}(t,\ \phi[\mathsf{path}],\ t') &=& t_1, \\ \phi[\mathsf{sort} := \mathsf{type\text{-}of}(t')][\mathsf{area} := \phi[\mathsf{area}][\mathsf{length} := \mathsf{area\text{-}length}(t')]] &=& \phi_1\end{array}}{\mathsf{replace\text{-}focus}(\epsilon,\ t') \ = \ \mathsf{editor}(t_1,\ \phi_1 + \!\!+\ \phi^{*\prime})}
$$

However, it might be that there is no current Focus to replace. In that case the original Editor will be returned.

$$[\text{rf-0}]\ \frac{\text{f-empty}(\epsilon[\text{focus}]) \ = \ \text{true}}{\text{replace-focus}(\epsilon,\, t') \ = \ \epsilon}$$

Creating a New Editor

The function new-editor initializes a new structure editor. It can be called with a string. This is specifically the case when a file containing source code is edited before it is parsed. Normally this would happen just once, when the file is retrieved from disk, and the user wants to edit it without having to wait for the parser to parse the file. It can also be called with an empty string, resulting in a empty editor, into which characters can be inserted. This happens normally when a user decides to edit a new (empty) file.

After new-editor is called with an empty string, the next action could also be to replace the root of the tree with a new parse tree. This happens when a user edits an existing file, but parses it first (before editing).

context-free syntax
new-editor(Literal) → Editor

The implementation of the new-editor function is somewhat arbitrary. It would be obvious to encode the parse tree of an empty editor as an empty termlist []. Since we like to be able to create a new editor containing an (unparsed) text string (for instance when a file is loaded in the text-editor, and not yet parsed), we need to be able to annotate an empty term with position (length) information pertaining to this string. The ATerm list-notation ([]) has nothing to do with lists as would appear in a language definition (and thus in a parse tree); these lists are encoded as functions with function symbol list.

In this specification ATerm lists (with square brackets) are not annotated with position information, because they do not have a clear correspondence with a syntactic construct in the text file. Because we chose not to annotate ATerm lists we cannot use the empty list as empty editor tree. Our solution is to have an empty whitespace ATerm (w("")) as an empty editor.

For a new editor we also introduce the special sort invalid. This is the only time we can have a Focus with an invalid sort. Because the string associated with the new editor has never been parsed, we have no way of "guessing" what sort it could be. We could have used the sort error for this Focus, but, as this is not an error but merely an exception value, we thought it better to introduce a new sort.

[ne] new-editor(*lit*) = editor([w(""){ln(to-lit(length(*lit*)))}],
[focus(root, invalid, area(1, length(*lit*)))])

Replacing a Focus with a Meta-variable

The function replace-with-meta replaces the current Focus with the meta-variable of the sort of the current Focus.

context-free syntax
replace-with-meta(Editor) → Editor

Replacing the Focus with the corresponding placeholder, or meta-variable, first of all requires a Focus. In order to build a term representing a meta-variable, we must first find out what the sort of the current Focus is, and whether or not it already *is* a meta-variable. If the subterm is not a meta-variable the (subterm representing a) meta-variable is constructed.

$$[\text{rwm-f}]\ \frac{\begin{array}{rcl} \text{f-empty}(\epsilon[\text{focus}]) &=& \text{false},\\ \epsilon[\text{focus}][\text{head}][\text{sort}] &=& t,\\ \text{create-metavar}(t) &=& t',\\ t' &\neq& [\,] \end{array}}{\text{replace-with-meta}(\epsilon) \;=\; \text{replace-focus}(\epsilon,\, t')}$$

The default case returns the original editor. This happens when there is no Focus, or the Focus is already a meta-variable.

$[\text{rwm-nf}]\ \text{replace-with-meta}(\epsilon) = \epsilon \quad$ **otherwise**

For the same reason there is no typechecking specified, there also is no function to expand a meta-variable; there is no knowledge of the language constructs that can be used in expanding a meta-variable present in the structure editor. Instead, as with typechecking, a separate tool should be created that lists all possible expansions for a specific meta-variable. Then, using replace-focus the meta-variable can be replaced with any of these expansions.

Moving a Focus

move-focus is called to move the Focus. Valid values for Move are up, down, left and right. If there is no active Focus, or if an illegal movement is requested (for instance a move up, when the Focus is already at the top of the parse tree) the original editor will be returned.

context-free syntax
move-focus(Editor, Move) $\rightarrow$ Editor

Moving the Focus up enlarges the Focus to hold the parse tree node above the current one. Take for example the following program, written in While (See Appendix B):

```
x := 10 ;
while x > 0 do {
 x := x - 1
}
```

where the underlined part is the current Focus. Moving the Focus up would then result in:

```
x := 10 ;
while x > 0 do {
 x := x - 1
}
```

Now we can move the Focus left to get:

```
x := 10 ;
while x > 0 do {
 x := x - 1
}
```

Moving the Focus down means setting the Focus to the child of a node, or, if a node has more than one child, to the first child of a node, as is demonstrated here:

```
x := 10 ;
while x > 0 do {
 x := x - 1
}
```

Finally, we move the Focus to the right:

```
x := 10 ;
while x > 0 do {
 x := x - 1
}
```

If a requested move is impossible, for instance moving down when the Focus is on a node without children, the sort of the Focus returned by move is error, and the original Editor will be returned.

$$[\text{mf-f}]\ \frac{\begin{array}{rcl}\text{f-empty}(\epsilon[\text{focus}]) &=& \text{false},\\ \text{move}(\epsilon,\ m) &=& \epsilon',\\ \epsilon'[\text{focus}][\text{head}][\text{sort}] &\neq& \text{error}\end{array}}{\text{move-focus}(\epsilon,\ m)\ =\ \epsilon'}$$

[mf-nf] move-focus(ϵ, m) = ϵ **otherwise**

Retrieving a Focus

This function can be called to identify the Focus associated with a specific character location. It does not change the Editor, but merely returns the Focus information (Sort, Path and Area) associated with a location.

context-free syntax
get-focus(Editor, Nat) $\rightarrow$ Focus

[gf] get-focus(ϵ, n) = create-focus(ϵ[tree], n)

2.2.3 Updating an Editor

We use the term updating for the actions that keep the information in the parse tree up-to-date with the information contained in the text editor. That is, any alteration to the state of the text editor (to the text string that is being edited, rather) requires a change in the parse tree contained in the Editor. Superficially, it would seem that *parsing* the text string every time it is altered would always keep the parse tree consistent with the text string. Unfortunately, this approach has a number of drawbacks. First of all, parsing is a relatively expensive operation, whereas typing a certain amount of text is not. To keep the parse tree perfectly consistent with the text string, we would have to reparse the text every time a character is added or removed. In an interactive system, such as this, any delay will be immediately noted by the user of such a system.

Secondly, the text string that is edited will, more often than not, be syntactically incorrect. Most syntactic constructs (in most languages) are longer than a single character, and therefore destined not to be correct after every single keystroke. In fact, this is exactly the idea of the Focus: the only place in a text string that is allowed to be syntactically incorrect, maintaining syntactic correctness everywhere outside the Focus.

This brings us to the third drawback: reparsing the *whole* text string, when only part of the string (only one character, in this scenario) has changed. Especially with large strings, this might be a serious loss of information and resources. Maintaining a Focus within the text string allows us to parse just very small portions of the text; anything outside the Focus is unchanged and still correct.

For now, we will explore the options of maintaining multiple Focuses without reparsing after every keystroke. We will, however, have to update our Focus information after every keystroke, or continuous sequence of keystrokes. In order to do this, we will now specify a number of functions that allow us to update the various datastructures of the Editor.

The Module Updating

module Updating
imports FocusPosition$^{2.2.8}$

Updates after a Character Insertion

The most common update is the result of a text insertion in the text editor. The function update-edit takes an ATerm and a FocusList as its first two arguments, and updates them with the number of characters inserted (Int, the last argument) at a specific location in the text (Nat, the third argument).

exports
  context-free syntax
    update-edit(ATerm, FocusList, Nat, Int) → Editor

The function update-edit is overloaded, if it is called with only the first three arguments of the previous function it will delete exactly one character from the specified location. This might seem odd. However, it poses seriouses difficulty to delete more than one character at a time. When inserting text, we simply increase the Focus containing the specified character location with the amount of characters inserted. It makes sense to have a function that can update for more than one character insertion at a time, because the number of actions taken for inserting one single character is exactly the same as the number of actions taken for the continuous insertion of a hundred characters.

When deleting characters, we can not simply decrease the current Focus, because a user can keep on deleting after the Focus is gone. So, after every character, we have to check if our original Focus still exists.

context-free syntax
update-edit(ATerm, FocusList, Nat) $\rightarrow$ Editor

Updating an Editor starts with finding the current Focus. We then update the length annotation at that Focus, to represent the newly inserted characters. Finally we update the other Focuses, such that the Area information in these Focuses is correct.

equations

$$[\text{ue-f}]\quad \frac{\begin{array}{rcl}\text{current-focus}((t, \phi^*), n) & = & \phi^{*\prime}, \\ \text{upd-anno}(t, \phi^{*\prime}[\text{head}][\text{path}], i) & = & t', \\ \text{upd-focus}(\phi^{*\prime}, n, i) & = & \phi^{*\prime\prime}\end{array}}{\text{update-edit}(t, \phi^*, n, i) \;=\; (t', \phi^{*\prime\prime})}$$

If update-edit is called with three arguments, then a single character is deleted from character position n.

$$[\text{ue-bs}]\quad \text{update-edit}(t, \phi^*, n) = \text{update-edit}(t, \phi^*, n, -1)$$

Updating a Focus list involves checking whether the character location at hand is *in* the Focus. In that case only the length of the Area needs to be updated. The start of the Area (the character position at which the Focus starts) stays the same.

$$[\text{uf-in}]\quad \frac{\begin{array}{rcl}\text{f-empty}(\phi^*) & = & \text{false}, \\ \phi^*[\text{head}] & = & \phi, \\ \phi^*[\text{tail}] & = & \phi^{*\prime}, \\ \nabla_\phi n & = & \text{true}, \\ \phi[\text{area}] & = & \alpha, \\ \alpha[\text{length}] & = & n_1\end{array}}{\begin{array}{l}\text{upd-focus}(\phi^*, n, i) \;= \\ \text{upd-focus}(\phi^{*\prime}, n, i) +\!\!+\; \phi[\text{area} := \alpha[\text{length} := n_1 + i]]\end{array}}$$

If the character insertion takes place *before* the Focus, then only the start of the Area of the Focus needs to be changed. The starting position of the Focus moves the same number of characters as just have been inserted. As the Focus itself is unchanged the length of the Area of the Focus is unchanged as well.

$$[\text{uf-af}]\quad \frac{\begin{array}{rcl}\text{f-empty}(\phi^*) & = & \text{false}, \\ \phi^*[\text{head}] & = & \phi, \\ \phi^*[\text{tail}] & = & \phi^{*\prime}, \\ \lhd_\phi n & = & \text{true}, \\ \phi[\text{area}] & = & \alpha, \\ \alpha[\text{start}] & = & n_1\end{array}}{\begin{array}{l}\text{upd-focus}(\phi^*, n, i) \;= \\ \text{upd-focus}(\phi^{*\prime}, n, i) +\!\!+\; \phi[\text{area} := \alpha[\text{start} := n_1 + i]]\end{array}}$$

If a character insertion took place *after* the Focus, then nothing changes, and we proceed with the other Focuses in the list.

$$[\text{uf-be}]\ \frac{\begin{array}{rcl}\text{f-empty}(\phi^*) & = & \text{false},\\ \phi^*[\text{head}] & = & \phi,\\ \phi^*[\text{tail}] & = & \phi^{*\prime},\\ \rhd_\phi n & = & \text{true}\end{array}}{\text{upd-focus}(\phi^*, n, i)\ =\ \text{upd-focus}(\phi^{*\prime}, n, i) +\!\!+\ \phi}$$

If there are no Focuses (left), we are finished updating Focuses.

$$[\text{uf-0}]\ \frac{\text{f-empty}(\phi^*)\ =\ \text{true}}{\text{upd-focus}(\phi^*, n, i)\ =\ \phi^*}$$

Updates after Replacing a Subtree

Another update occurs when a user decides to explicitly replace one part of the structure with another part. For instance when expanding a placeholder. upd-term takes the ATerm (the third argument) to be inserted in the parse tree (the first argument) at the position given by the Path (the second argument).

context-free syntax
upd-term(ATerm, Path, ATerm) $\rightarrow$ ATerm

Updating terms is rather straightforward. We follow the Path (the second argument of the function upd-term) and, when we get to the end of the Path, we replace a subterm of the parse tree with a given term (the third argument of the function).

If we have a leaf Path and the term to be updated is a ATermList, then the Path points at the first element of the list. That element will be replaced by the replacement term.

$$[\text{ut-r-nw}]\ \frac{\text{is-leaf-path}(\pi)\ =\ \text{true}}{\text{upd-term}(tl, \pi, t)\ =\ \text{replace}(tl, \text{zero}, t)}$$

If the term to be replaced is not a ATermList, then it is an AsFixTerm.

$$[\text{ut-r-at}]\ \frac{\begin{array}{rcl}\text{is-leaf-path}(\pi) & = & \text{true},\\ \begin{array}{l}\text{my-is-cons}(\text{var}, t)\\ \vee\ \text{my-is-cons}(\text{lex}, t)\\ \vee\ \text{my-is-cons}(\text{appl}, t)\\ \vee\ \text{my-is-cons}(\text{list}, t)\end{array} & = & \text{true}\end{array}}{\text{upd-term}(t, \pi, t')\ =\ t'}$$

ATermLists can have an annotation, in which case we cannot match directly. In this specification ATermLists are not annotated, so we just remove any annotation we find, update the term, and replace the annotation.

[ut-lst-ann] $\text{upd-term}(tl\{t'\}, \pi, t) = \text{upd-term}(tl, \pi, t)\{t'\}$

If a Path leads to an application, we have to check whether this is a branch node (an internal node) in the tree. Otherwise it is a leaf and it will be handled by another equation. If it is a branch node we proceed with the children of this node.

$$[\text{ut-app}]\ \frac{\begin{array}{rcl}\text{get-cons}(t_1) & = & \text{appl},\\ \text{head-zero}(\pi) & = & \text{true}\end{array}}{\text{upd-term}(t_1, \pi, t_2) \ = \ t_1[\text{args} := \text{upd-term}(t_1[\text{args}], \pi[\text{tail}], t_2)]}$$

The same holds for elements with function symbol list.

$$[\text{ut-lst}]\ \frac{\begin{array}{rcl}\text{get-cons}(t_1) & = & \text{list},\\ \text{head-zero}(\pi) & = & \text{true}\end{array}}{\text{upd-term}(t_1, \pi, t_2) \ = \ t_1[\text{elems} := \text{upd-term}(t_1[\text{elems}], \pi[\text{tail}], t_2)]}$$

If we have a ATermList, and the head of the Path is zero, then we replace the first element of the list with the updated term.

$$[\text{ut-l-z}]\ \frac{\text{head-zero}(\pi) \ = \ \text{true}}{\text{upd-term}(tl, \pi, t) \ = \ \text{replace}(tl, \text{zero}, \text{upd-term}(\text{first}(tl), \pi, t))}$$

If the head element of the Path is non-zero, then we decrease the head, and try the next element.

$$[\text{ut-l-nz}]\ \frac{\begin{array}{rcl}\text{head-zero}(\pi) & = & \text{false},\\ \text{is-leaf-path}(\pi) & = & \text{false}\end{array}}{\begin{array}{l}\text{upd-term}(tl, \pi, t) \ = \\ [\text{first}(tl)] \mathbin{+\!\!+} \text{aterm-list}(\text{upd-term}(\text{rest}(tl), \text{decrease-head}(\pi), t))\end{array}}$$

Auxiliary Functions

The hidden functions are auxiliary functions to update-editor and update specific datatypes used in the structure editor.

hiddens
  context-free syntax
    upd-anno(ATerm, Path, Int) → ATerm
    upd-focus(FocusList, Nat, Int) → FocusList

When we update an annotation we distinguish between two cases. If the current Path is the root (see Section 2.1.4 for an explanation of the confusing name of the function is-leaf-path) then we update the current parse tree (the current term) by adding i to its length.

$$[\text{ua}]\ \frac{\text{is-leaf-path}(\pi) \ = \ \text{true}}{\text{upd-anno}(t, \pi, i) \ = \ \text{upd-term}(t, \pi, \text{add-to-len}(t_\pi, i))}$$

If the current Path is not root we first update the annotations of the parent of this node, and, after that, update this node with the number of inserted characters.

$$[\text{ua}]\ \frac{\text{is-leaf-path}(\pi) \ = \ \text{false}}{\begin{array}{l}\text{upd-anno}(t, \pi, i) \ = \\ \text{upd-term}(\text{upd-anno}(t, \pi\uparrow, i), \pi, \text{add-to-len}(t_\pi, i))\end{array}}$$

These two equations let us update only that part of the tree that is affected by the change in the text editor. We first move down in the tree, until we come to the leaf node that has been edited. Then we change only the annotations of this nodes ancestors.

2.2.4 Operations on Focuses

The Module Focuses

module Focuses
imports FocusSyntax$^{2.1.2}$ Areas$^{A.5}$

Operations on Lists of Focuses

We define a number of functions that allow Focuses and FocusLists to be concatenated to form a new FocusList. The predicate f-empty is defined on the sort FocusList. It is called f-empty rather then the more common empty because there already is a function empty defined on ATermLists which caused an ambiguity while parsing.

context-free syntax

f-empty(FocusList)	$\rightarrow$	Bool
FocusList "++" FocusList	$\rightarrow$	FocusList
Focus "++" Focus	$\rightarrow$	FocusList
FocusList "++" Focus	$\rightarrow$	FocusList
Focus "++" FocusList	$\rightarrow$	FocusList

Following are the equations for FocusList concatenation, and the f-empty predicate.

equations

[cc-1]	$[fl_1]$ ++ $[fl_2]$	$= [fl_1; fl_2]$	
[cc-2]	ϕ ++ $[fl]$	$= [\phi; fl]$	
[cc-3]	$[fl]$ ++ ϕ	$= [fl; \phi]$	
[cc-4]	ϕ ++ ϕ'	$= [\phi; \phi']$	
[efl-t]	f-empty([])	= true	
[efl-f]	f-empty(ϕ^*)	= false	**otherwise**

Focus Creation and Maintenance

When an Editor has a Focus, and a text-insertion (insert-chars) takes place outside this Focus, we have to create a new Focus. If the new Focus is a parent (a node higher up in the tree) of the first Focus, the first one can be removed, because the new Focus already contains the old one. In order to accomplish this, three functions are defined to operate on Focuses. Creating a new Focus is done by create-focus, which takes the character position where the Focus should be as its second argument, and a parse tree as its first.

If there already were one or more Focuses we use remove-children to remove all Focuses from the FocusList that are children of the Focus. The function remove-children uses the predicate is-child to see whether its first argument is a child of its second argument.

exports
context-free syntax

remove-children(Focus, FocusList)	$\rightarrow$	FocusList
is-child(Focus, Focus)	$\rightarrow$	Bool
create-focus(ATerm, Int)	$\rightarrow$	Focus
create-focus(ATerm, Path)	$\rightarrow$	Focus

We will now show the equations for the above mentioned functions.

If the focus list is empty, no children have to be removed.

$$[\text{rc-1}]\ \frac{\text{f-empty}(\phi^*)\ =\ \text{true}}{\text{remove-children}(\phi,\ \phi^*)\ =\ [\phi]}$$

Now, the focus list is not empty, and ϕ' is a child of ϕ. ϕ' is removed, and the rest of the list will be checked.

$$[\text{rc-n-t}]\ \frac{\begin{array}{rcl}\text{f-empty}(\phi^*) & = & \text{false},\\ \phi^*[\text{head}] & = & \phi',\\ \phi^*[\text{tail}] & = & \phi^{*\prime},\\ \text{is-child}(\phi',\ \phi) & = & \text{true}\end{array}}{\text{remove-children}(\phi,\ \phi^*)\ =\ \text{remove-children}(\phi,\ \phi^{*\prime})}$$

If ϕ' is not a child of ϕ, then the rest of the list is checked, and ϕ' (which has now been checked) is added at the end of the resulting FocusList.

Note that although we have attributed a special meaning to the *first* focus in the list (being the *current* or *primary* focus), the other elements of a FocusList are without specific order. They can be moved around at will.

$$[\text{rc-n-f}]\ \frac{\begin{array}{rcl}\text{f-empty}(\phi^*) & = & \text{false},\\ \phi^*[\text{head}] & = & \phi',\\ \phi^*[\text{tail}] & = & \phi^{*\prime},\\ \text{is-child}(\phi',\ \phi) & = & \text{false}\end{array}}{\text{remove-children}(\phi,\ \phi^*)\ =\ \text{remove-children}(\phi,\ \phi^{*\prime})\ +\!\!+\ \phi'}$$

A focus is a child of another focus if and only if its Path is a child-path of the other focus. That is, the Path of ϕ_1 leads to a node that is a child of the node pointed to by the Path of ϕ_2. See Section 2.1.4 for the exact specification of is-child-path.

$$[\text{ic}]\ \text{is-child}(\phi_1,\ \phi_2) = \text{is-child-path}(\phi_1[\text{path}],\ \phi_2[\text{path}])$$

If a character is inserted outside an existing focus (or a focus is requested through get-focus from section 2.2.2) a focus is created by

1. Creating a Path to the node in the parse tree that embeds the requested character position.
2. Retrieving the Sort of the subterm this Path leads to.
3. Calculating the Area for this subterm, that is, finding the character position at which the subterm starts, and the number of characters the subterm is long.

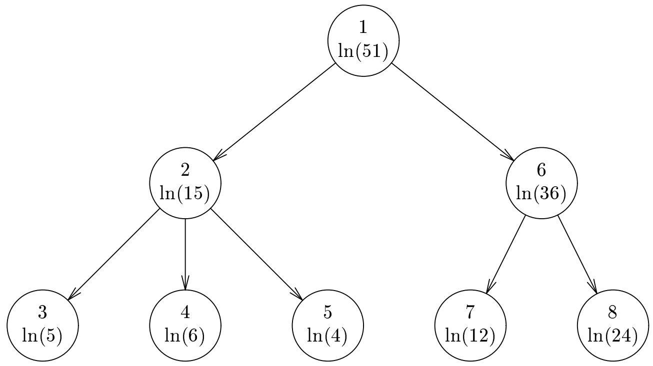

Figure 2.5: An example of how the position information is annotated throughout the tree. The first number is a label for the node. ln(nat) defines the length of this node to be nat.

$$[\textsf{cf-chr}]\ \frac{\begin{array}{rcl}\textsf{get-path}(t,\, i) & = & \pi, \\ \textsf{type-of}(t_\pi) & = & \sigma, \\ \textsf{create-area}(\pi,\, t) & = & \alpha\end{array}}{\textsf{create-focus}(t,\, i) \;=\; \textsf{focus}(\pi,\, \sigma,\, \alpha)}$$

Of course, we can also create a focus starting with different information. All we need is a way to encode an exact location in a tree, and the tree itself. Before, we used a character location to define the place in the tree that the focus was on. We could also use a Path in a tree to achieve the same effect.

$$[\textsf{cf-pth}]\ \frac{\begin{array}{rcl}\textsf{create-area}(\pi,\, t) & = & \alpha, \\ \textsf{type-of}(t_\pi) & = & \sigma\end{array}}{\textsf{create-focus}(t,\, \pi) \;=\; \textsf{focus}(\pi,\, \sigma,\, \alpha)}$$

2.2.5 Calculating Position Information

Because both text and structure can be edited using this editor, we have to make sure that there is a direct and close relation between the two. At all times should we be able to return the exact character location of any one part of the parse tree. The formalism we use for representing parse trees (ATF, see Section 2.1.1) allows us to reconstruct the source text from the parse tree. However, it would be very inefficient to reconstruct the source text, and then calculate the position information, each time a request is made for this information. Instead, we annotate the various nodes in the tree with their length in characters. See Figure 2.5 for an example. Having such an annotated tree, together with the Path information from Section 2.1.4 we can retrieve the position of a subtree much more efficient. In fact, the number of nodes that need to be visited to calculate the position (or Area, see section 2.1.6) can be calculated using the function number-of-nodes, as specified below. If we consider a parse tree to be a tree with n nodes, than the worst case (the maximum number of nodes we need to visit) is still n, but this case happens only a very small number of times. In fact, it happens only when the parse tree contains only one lexical, and no whitespace, such as the parse tree of the Boolean true.

The Module Position-information

module Position-information
imports Areas[A.5]

Character Variables

variables
Char [*0-9'*]$*$ → CHAR
"*Char+*"[*0-9'*]$*$ → CHAR+
"*Char$*$*"[*0-9'*]$*$ → CHAR$*$

Length Annotations

A function for manipulating (add-to-len) and one for creating (annotate) length annotations is specified here.

exports
context-free syntax
add-to-len(ATerm, Int) → ATerm
annotate(ATerms) → ATerm

When a character is inserted in the text editor, the position information in the parse tree needs to be updated. On the lowest level, the length annotation of a single parse tree node needs to be updated. Or, more precisely, the number of inserted characters should be added to the length annotation of the node where the characters were inserted. add-to-len adds an integer to the annotation of a single term t. add-to-len(t, i) adds i to the position annotation of t. This function is incompletely specified in the sense that it assumes that there already is a length annotation.

equations

$$[\mathsf{aa}]\quad \frac{\begin{array}{rcl} t[\mathsf{anno}][\mathsf{ln}] & = & t', \\ t' & = & \mathsf{ln}(\mathit{lit}), \\ t[\mathsf{anno} := \mathsf{replace\ ln\ with\ ln(to\text{-}lit(to\text{-}int(}\mathit{lit}) + i))\ \mathsf{from}\ t[\mathsf{anno}]] & = & t'' \end{array}}{\mathsf{add\text{-}to\text{-}len}(t,\ i) \ = \ t''}$$

Currently, parse trees that are returned by the parser have no position information annotated to them. This should change in the future, as the most obvious place to add position information is the parser. Since the parser can simply count the number of characters it has already parsed, adding position information does not increase the complexity of parsing a string.

For now, the annotation is done inside this specification. The function annotate should be seen as a temporary solution. Because it passes over every node in the parse tree, it is an expensive operation. annotate is a pre-order traversal function. annotate(t) annotates a term with its position information. Rather, for each subterm of t annotate will calculate the length of the character string that t represents, including whitespace. In this way, the position information in one part of the tree (in one subterm) is independent from the position information in another subtree. When we replace a subtree with another subtree, we only have to update the length annotation for that particular subtree and not the rest of the tree.

Preorder traversal of the parse tree is defined by:

$$
\begin{array}{lll}
\text{[a-app]} & \mathsf{annotate}(t,\, t^{+}) & = \mathsf{annotate}(t),\, \mathsf{annotate}(t^{+}) \\
\text{[a-lst]} & \mathsf{annotate}([t^{+}]) & = [\mathsf{annotate}(t^{+})]
\end{array}
$$

If a leaf is found, it is annotated with the number of characters it consists of.

$$
\text{[a-wl]}\quad \frac{\begin{array}{rcl} \mathsf{my\text{-}is\text{-}cons}(\mathsf{l},\, t) \vee \mathsf{my\text{-}is\text{-}cons}(\mathsf{w},\, t) & = & \mathsf{true}, \\ t[\mathsf{chars}] & = & \mathit{lit} \end{array}}{\mathsf{annotate}(t) \;=\; t[\mathsf{anno} := t[\mathsf{anno}] \mathbin{+\!\!+} [\mathsf{ln}(\mathsf{to\text{-}lit}(\mathsf{length}(\mathit{lit})))]]}
$$

If a parent node is found (either an appl or a list), first all its children will be annotated, and then the parent will be annotated with the sum of the annotations of all its children. This allows for faster calculation of positions of later nodes. As an example, take the tree from Figure 2.5. Suppose we want to calculate at which character position the text string corresponding to node 7 starts. We now follow the Path to node 7 and add up all lengths of each node we visit, to end at node 7, with the exact position. If we decide not to annotate the so-called *branch* nodes (the internal vertices of the tree) we would have to do a full preorder traversal up to node 7 each time we want the character location for that node. In this case that would mean we had to visit 6 nodes (we do not have to visit the last node). With our method we only need to visit 3 nodes.

$$
\text{[a-appl]}\quad \frac{\begin{array}{rcl} \mathsf{get\text{-}cons}(t) & = & \mathsf{appl}, \\ \mathsf{annotate}(t[\mathsf{args}]) & = & t^{+\prime}, \\ \mathsf{subtrm\text{-}len}(t^{+\prime}) & = & i, \\ t[\mathsf{args} := [t^{+\prime}]] & = & t' \end{array}}{\mathsf{annotate}(t) \;=\; t'[\mathsf{anno} := t'[\mathsf{anno}] \mathbin{+\!\!+} [\mathsf{ln}(\mathsf{to\text{-}lit}(i))]]}
$$

$$
\text{[a-lst]}\quad \frac{\begin{array}{rcl} \mathsf{get\text{-}cons}(t) & = & \mathsf{list}, \\ \mathsf{annotate}(t[\mathsf{elems}]) & = & t^{+\prime}, \\ \mathsf{subtrm\text{-}len}(t^{+\prime}) & = & i, \\ t[\mathsf{elems} := [t^{+\prime}]] & = & t' \end{array}}{\mathsf{annotate}(t) \;=\; t'[\mathsf{anno} := t'[\mathsf{anno}] \mathbin{+\!\!+} [\mathsf{ln}(\mathsf{to\text{-}lit}(i))]]}
$$

Auxiliary Functions

To calculate the number of nodes that need to be visited to find the position information for a certain node we provide this function. It is not used anywhere and is included for demonstration purposes only.

context-free syntax

$$
\begin{array}{ll}
\mathsf{number\text{-}of\text{-}nodes}(\mathsf{Path}) & \to \mathsf{AInt} \\
\mathsf{length}(\mathsf{Literal}) & \to \mathsf{Int}
\end{array}
$$

The following are auxiliary functions to the length annotation functions.

hiddens
context-free syntax
subtrm-len(ATerms) $\to$ Int

To get the position information we need to traverse number-of-nodes nodes.

[nn-1] number-of-nodes(path(0)) = succ(0)

$$\text{[nn-n]}\quad \frac{t_i \neq 0}{\text{number-of-nodes(path}(t_i)) = \text{succ(number-of-nodes(path(pred}(t_i))))}}$$

[nn-1-n] number-of-nodes(path(0, t^+))
= succ(number-of-nodes(path(t^+)))

$$\text{[nn-n-n]}\quad \frac{t_i \neq 0}{\text{number-of-nodes(path}(t_i, t^+)) = \text{succ(number-of-nodes(path(pred}(t_i), t^+)))}}$$

subtrm-len(t) calculates the length (in terms of characters) of a term. It adds up all length annotations of all subterms of t. It is used as an auxiliary function to annotate.

[len-app] subtrm-len(t, t^+) = subtrm-len(t) + subtrm-len(t^+)
[len-lst] subtrm-len([t^+]) = subtrm-len(t^+)

$$\text{[len-ann]}\quad \frac{\begin{array}{rcl} \text{my-is-cons(appl}, t) \lor \text{my-is-cons(list}, t) \lor \text{my-is-cons(w}, t) \lor \text{my-is-cons(l}, t) & = & \text{true}, \\ t\text{[anno][ln]} & = & t', \\ t' & = & \text{ln}(lit) \end{array}}{\text{subtrm-len}(t) = \text{to-int}(lit)}$$

[len-n] subtrm-len(t) = 0 **otherwise**

Finally, we need an operation that actually counts the length of a literal.

length("$Char*$") calculates the length of a single character string. If the character is not a slash it counts as one character. If there is a slash it escapes the next character, so the two characters count as one character. These equations imply that no string can end on a single slash, which is correct, because if a string would end on a single slash, the escaping mechanism of the parser would escape it with an extra slash.

[len-n] length(literal(""" *Char Char** """))
= length(literal(""" *Char** """)) + 1
otherwise

[len-slash] length(literal(""" "\" *Char Char** """))
= length(literal(""" *Char** """)) + 1

[len-0] length("") = 0

2.2.6 Constructing Paths to Subtrees

This section specifies the construction of Paths in a tree. Although encoding addresses in a tree as Paths allows us to move to a specific location in the tree much more quickly, there is of course one point where we do not have this Path information, and have to calculate it. This algorithm operates under the assumption that only Path addresses that are actually needed will be calculated. That is, only if a specific node is of interest (appears in a Focus) will we calculate its Path. If the text corresponding to a certain node in the tree is never edited, the Path to that node will never be calculated. Due to this strategy, any operation that requires us to create a new Focus is delayed some time to calculate the Focus. It remains to be seen whether or not this delay is acceptable in an interactive system. (See Section 4.1 for performance experiments).

Another approach would be to calculate Path information for *all* nodes in the tree. This would increase the amount of calculations we have to do, but would change the time of the calculation from "during editing" to "during parsing". As a user expects parsing to take some time it is possible that, although this strategy takes more (much more, in a worst case scenario) time, it might feel to the user as being faster. The user expects editing to be immediate, because it is perceived as a "simple" operation; any delay is felt immediately.

We did not try the second scenario. However, it is not difficult to change the specification in such a way that calculating the Path information is no longer a separate function, but integrated in the function that annotates trees with their position information (The function annotate from Section 2.2.5).

The Module Path-construction

module Path-construction
imports Paths$^{2.1.4}$ Annotations$^{A.4}$

Creating a Path

The Path creation function is called get-path. It returns a Path in the tree ATerm to the smallest subtree that encloses character position Int.

exports
  context-free syntax
    get-path(ATerm, Int) $\rightarrow$ Path

The function get-path uses an auxiliary overloaded get-path function. (The extra Path argument is used to store intermediate results).

hiddens
context-free syntax
get-path(ATerm, Int, Path) $\to$ Path

We will now construct a Path. We start at the root of the tree. There is no Path yet. We create a Path root and call get-path with root as an extra argument.

equations

[p-start] get-path(t, n) = get-path(t, n, root)

If the top constructor symbol is an application and the character position we are looking for is in this application then we add a child to the Path and look in the arguments of the application.

$$\text{[p-in-appl]}\ \frac{\begin{array}{rcl}\text{get-cons}(t) &=& \text{appl},\\ \text{get-ln}(t) &=& i,\\ n \le i &=& \text{true}\end{array}}{\text{get-path}(t, n, \pi) \;=\; \text{get-path}(t[\text{args}], n, \pi\downarrow)}$$

If the top symbol is a list and the character position is in this list then we look at the elements of the list.

$$\text{[p-in-list]}\ \frac{\begin{array}{rcl}\text{get-cons}(t) &=& \text{list},\\ t[\text{elems}] &=& t',\\ \text{get-ln}(t') &=& i,\\ n \le i &=& \text{true}\end{array}}{\text{get-path}(t, n, \pi) \;=\; \text{get-path}(t', n, \pi\downarrow)}$$

If the top symbol is either a lexical or a variable and the character position of interest is in such a lexical or variable we are finished. These terms do not have children.

$$\text{[p]}\ \frac{\begin{array}{rcl}\begin{array}{l}\text{my-is-cons}(\text{lex}, t)\\ \lor\ \text{my-is-cons}(\text{var}, t)\end{array} &=& \text{true},\\ \text{get-ln}(t) &=& i,\\ n \le i &=& \text{true}\end{array}}{\text{get-path}(t, n, \pi) \;=\; \pi}$$

(We assume that a top-symbol can never be whitespace. If whitespace does occur as a top symbol it is handled by the default (error) equation)

[error] get-path(t, n) = `"path-construction-error"` **otherwise**

We have now read the first (top) symbol and moved to a lower level. Only lists of ATerms live here. However, if a list has an annotation, we cannot match on a list as a list. We therefore remove the annotation first:

[gp-rm-anno] get-path($tl\{t\}$, i, π) = get-path(tl, i, π)

We check the first element of the list for its size and compare it to the text-location n. If n is larger then we move to the next child (the next element in the list).

$$[\mathsf{p'}]\ \frac{\begin{array}{rcl}\mathsf{get\text{-}ln}(\mathsf{first}(tl)) & = & i', \\ i > i' & = & \mathsf{true}\end{array}}{\mathsf{get\text{-}path}(tl,\ i,\ \pi)\ =\ \mathsf{get\text{-}path}(\mathsf{rest}(tl),\ i - i',\ \pi_{\rightarrow})}$$

If the text location of interest *is* within the first element of the list we have to take a closer look. In case of an application or a list, we repeat the step at the top and move (yet again) to a lower level.

$$[\mathsf{p'}]\ \frac{\begin{array}{rcl}\mathsf{first}(tl) & = & t, \\ \mathsf{get\text{-}ln}(t) & = & i, \\ n \leq i & = & \mathsf{true}, \\ \mathsf{get\text{-}cons}(t) & = & \mathsf{appl}\end{array}}{\mathsf{get\text{-}path}(tl,\ n,\ \pi)\ =\ \mathsf{get\text{-}path}(t[\mathsf{args}],\ n,\ \pi_{\downarrow})}$$

$$[\mathsf{p'}]\ \frac{\begin{array}{rcl}\mathsf{first}(tl) & = & t, \\ \mathsf{get\text{-}ln}(t) & = & i, \\ n \leq i & = & \mathsf{true}, \\ \mathsf{get\text{-}cons}(t) & = & \mathsf{list}\end{array}}{\mathsf{get\text{-}path}(tl,\ n,\ \pi)\ =\ \mathsf{get\text{-}path}(t[\mathsf{elems}],\ n,\ \pi_{\downarrow})}$$

In case of a lexical or a variable we are finished.

$$[\mathsf{p'}]\ \frac{\begin{array}{rcl}\mathsf{first}(tl) & = & t, \\ \mathsf{get\text{-}ln}(t) & = & i, \\ n \leq i & = & \mathsf{true}, \\ \begin{array}{l}\mathsf{my\text{-}is\text{-}cons}(\mathsf{lex},\ t) \\ \vee\ \mathsf{my\text{-}is\text{-}cons}(\mathsf{var},\ t)\end{array} & = & \mathsf{true}\end{array}}{\mathsf{get\text{-}path}(tl,\ n,\ \pi)\ =\ \pi}$$

If the element containing the text location happens to be a whitespace element we are finished as well, only we delete the last step of Path construction. It is undesirable to have whitespace as “the smallest possible subtree” because it is layout, and of no interest for structure editing purposes. So we move the Focus one step up to include the subtree “surrounding” the whitespace.

$$[\mathsf{p'}]\ \frac{\begin{array}{rcl}\mathsf{first}(tl) & = & t, \\ \mathsf{get\text{-}ln}(t) & = & i, \\ n \leq i & = & \mathsf{true}, \\ \begin{array}{l}\mathsf{my\text{-}is\text{-}cons}(\mathsf{w},\ t) \\ \vee\ \mathsf{my\text{-}is\text{-}cons}(\mathsf{l},\ t)\end{array} & = & \mathsf{true}\end{array}}{\mathsf{get\text{-}path}(tl,\ n,\ \pi)\ =\ \pi_{\uparrow}}$$

2.2.7 Retrieving Subtrees

In this section the retrieval of specific subtrees of a parse tree is specified. The only way to retrieve such a subtree is by first specifying the Path leading to that tree (see section 2.2.6). Given such a Path we now specify how the actual subtree is returned.

The Module Subtrees

module Subtrees
imports AsFix-Utilities$^{A.2}$ Paths$^{2.1.4}$

Retrieving a Subtree

An ATerm "indexed" with a Path returns the subterm of the ATerm at the end of the Path. This function is used throughout this specification as $\mathrm{ATerm}_{\mathrm{Path}}$. It is shown in the syntax definition as ATerm_Path as a result of a limitation on our typesetting system.

exports
context-free syntax
ATerm "_" Path → ATerm

If the current Path is a leaf, then we are finished, except when the term is a list-term. In that case we have to get the first element out of the list.

equations

$$[\text{gs-root-nw}]\quad \frac{\begin{array}{rcl}\text{is-leaf-path}(\pi) & = & \text{true},\\ tl & \neq & [\,]\end{array}}{tl_{\pi} \;=\; \text{first}(tl)}$$

If the current term is not a list it must be an AsFixTerm, as defined in Appendix C.

$$[\text{gs-root}]\quad \frac{\begin{array}{rcl}\text{is-leaf-path}(\pi) & = & \text{true},\\ \begin{array}{l}\text{my-is-cons}(\text{var},\, t)\\ \vee\ \text{my-is-cons}(\text{lex},\, t)\\ \vee\ \text{my-is-cons}(\text{appl},\, t)\\ \vee\ \text{my-is-cons}(\text{list},\, t)\end{array} & = & \text{true}\end{array}}{t_{\pi} \;=\; t}$$

If the constructor is an application, and the head of the path is zero (i.e., this is the application of interest) then we move one level down in the tree.

$$[\text{gs-appl-z}]\quad \frac{\begin{array}{rcl}\text{head-zero}(\pi) & = & \text{true},\\ \text{get-cons}(t) & = & \text{appl},\\ t[\text{args}] & = & t',\\ \pi[\text{tail}] & = & \pi'\end{array}}{t_{\pi} \;=\; t'_{\pi'}}$$

The same holds if the constructor is a list.

$$[\text{gs-list-z}]\quad \frac{\begin{array}{rcl}\text{head-zero}(\pi) & = & \text{true},\\ \text{get-cons}(t) & = & \text{list},\\ t[\text{elems}] & = & t',\\ \pi[\text{tail}] & = & \pi'\end{array}}{t_{\pi} \;=\; t'_{\pi'}}$$

If the constructor would be anything else (e.g. lex) and the head of the path would be zero, an error is made while constructing the path. This equation is especially used when moving the Focus (Section 2.2.8). When moving the Focus, a Path is constructed relative to an existing Path. If there is no subtree corresponding to this Path the subtree retrieval function will return an error, which in turn is handled by the Focus moving function.

[error] $t_\pi = \texttt{"error"}$ **otherwise**

If the tree is a list of ATerms, and the first element of this list is an application function, *and* the head of the path is zero, then we proceed to the child of the application (its arguments).

$$[\text{gs-l-app-z}]\ \frac{\begin{aligned}\text{head-zero}(\pi) &= \text{true},\\ \text{my-is-cons}(\text{appl}, \text{first}(tl)) &= \text{true},\\ \text{first}(tl)[\text{args}] &= t,\\ \pi[\text{tail}] &= \pi'\end{aligned}}{tl_\pi = t_{\pi'}}$$

Indeed the same holds for the list constructor. Now the children are the elements of the list.

$$[\text{gs}]\ \frac{\begin{aligned}\text{head-zero}(\pi) &= \text{true},\\ \text{my-is-cons}(\text{list}, \text{first}(tl)) &= \text{true},\\ \text{first}(tl)[\text{elems}] &= t,\\ \pi[\text{tail}] &= \pi'\end{aligned}}{tl_\pi = t_{\pi'}}$$

If we are scanning a list of ATerms and the head of the path is not zero, then we proceed to the next ATerm in the list.

$$[\text{gs}]\ \frac{\begin{aligned}\text{head-zero}(\pi) &= \text{false},\\ \text{is-leaf-path}(\pi) &= \text{false}\end{aligned}}{tl_\pi = \text{rest}(tl)_{\text{decrease-head}(\pi)}}$$

2.2.8 The Location of a Focus in the Text

In the text editor, we allow so called *unrestricted* editing. As stated in Section 2.2.3, we will only allow the text string to be syntactically incorrect inside a Focus. Combining these two conditions means that we have to create a new Focus whenever a user decides to edit outside the current Focus. Moreover, it means that each time any text manipulation takes place, we have to decide whether this edit is in the current Focus, or outside it. If it is outside of the current Focus, it could be either on a location where there is no Focus at all, or it could be in a previous (now secondary) Focus, as we allow multiple Focuses to co-exist. This section specifies three functions on Focuses that check whether a character position is before ($\lhd$), after ($\rhd$) or inside ($\triangledown$) the current Focus. Using these functions we are able to set the *current* Focus, that is, the (new) Focus in which the current editing action takes place.

The Module FocusPosition

module FocusPosition
imports Editor$^{2.1.7}$

Relative Focus Locations

In order to find the current Focus, the location of all Focuses relative to the character position of interest need to be calculated. The functions $\lhd$, $\rhd$ and ∇ return a boolean corresponding to whether the character at location Nat is before, after, or inside the Focus. So if a Focus ϕ runs from character 1204 up to character 1320 (has Area) (1204, 116), and a character insertion takes place at position 352, then a call to $\nabla_\phi 352$ will yield false, a call to $\rhd_\phi$ 352 will also yield false and a call to $\lhd_\phi$ 352 will yield true

context-free syntax
"∇" "_" Focus Nat $\to$ Bool
"$\rhd$" "_" Focus Nat $\to$ Bool
"$\lhd$" "_" Focus Nat $\to$ Bool

If the position is between the starting location and the sum of the starting location and the length of a Focus, it is inside this Focus.
equations

$$[\text{if-n}]\ \frac{\begin{array}{rcl}\phi[\mathsf{area}] &=& \alpha,\\ \alpha[\mathsf{start}] &=& n_2,\\ \alpha[\mathsf{length}] &=& n_3,\\ n_2 \le n_1 &=& \mathsf{true},\\ n_1 \le n_2 + n_3 &=& \mathsf{true}\end{array}}{\nabla_\phi n_1 \ = \ \mathsf{true}}$$

Otherwise it is not.

[if] $\nabla_\phi n = \mathsf{false}$ **otherwise**

If a character is located before the starting location of a Focus, it is before the Focus.

$$[\text{bf-n}]\ \frac{\begin{array}{rcl}\phi[\mathsf{area}] &=& \alpha,\\ \alpha[\mathsf{start}] &=& n_2,\\ n_2 > n_1 &=& \mathsf{true}\end{array}}{\lhd_\phi n_1 \ = \ \mathsf{true}}$$

Otherwise it is not

[bf] $\lhd_\phi n = \mathsf{false}$ **otherwise**

Finally, if a character location is larger than the sum of the starting point and the length of a Focus, the character is after the Focus.

$$[\text{af-n}]\ \frac{\begin{array}{rcl}\phi[\mathsf{area}] &=& \alpha,\\ \alpha[\mathsf{start}] &=& n_2,\\ \alpha[\mathsf{length}] &=& n_3,\\ n_2 + n_3 < n_1 &=& \mathsf{true}\end{array}}{\rhd_\phi n_1 \ = \ \mathsf{true}}$$

And again, otherwise it is not.

[af] $\rhd_{\phi} n = \mathsf{false}$ **otherwise**

Finding the Current Focus

When a character is inserted or removed from a specified location, there has to be a Focus there. The function current-focus takes the FocusList from the Editor, and manipulates it in such a way that the first Focus in the resulting FocusList is the active Focus, the one containing at least the character at position Nat.

exports
context-free syntax
current-focus(Editor, Nat) → FocusList

We will now specify the function that locates the current Focus. If there indeed is a Focus, and the current character position n is inside it, then that is the current Focus.

$$[\text{ff-1}]\ \frac{\begin{array}{rcl} \epsilon[\mathsf{focus}] & = & \phi^*, \\ \mathsf{f\text{-}empty}(\phi^*) & = & \mathsf{false}, \\ \nabla_{\phi^*[\mathsf{head}]} n & = & \mathsf{true} \end{array}}{\mathsf{current\text{-}focus}(\epsilon,\ n) \ = \ \phi^*}$$

If there is a Focus, but the current character n is not inside it, then we find the current Focus in the rest of the FocusList. The original first Focus is saved, and put at the end of the new FocusList. This means that there has been a change in active or primary Focus. The old primary Focus (ϕ) is removed from the head of the FocusList, and replaced at the end of it. The new current Focus is the head of ϕ_3^*. Before returning this new list, we check whether our new current Focus has any children in the (old) FocusList. If so, they are removed and the resulting list is returned.

$$[\text{ff-n}]\ \frac{\begin{array}{rcl} \epsilon[\mathsf{focus}] & = & \phi_1^*, \\ \epsilon[\mathsf{tree}] & = & t, \\ \mathsf{f\text{-}empty}(\phi_1^*) & = & \mathsf{false}, \\ \phi_1^*[\mathsf{head}] & = & \phi, \\ \phi_1^*[\mathsf{tail}] & = & \phi_2^*, \\ \nabla_{\phi} n & = & \mathsf{false}, \\ \mathsf{current\text{-}focus}(\mathsf{editor}(t,\ \phi_2^*),\ n) & = & \phi_3^*, \\ \mathsf{remove\text{-}children}(\phi_3^*[\mathsf{head}],\ \phi_3^*[\mathsf{tail}] +\!\!+ \phi) & = & \phi_4^* \end{array}}{\mathsf{current\text{-}focus}(\epsilon,\ n) \ = \ \phi_4^*}$$

If there is no Focus at all, then we create a new Focus, and return it.

$$[\text{ff-i}]\ \frac{\begin{array}{rcl} \epsilon[\mathsf{focus}] & = & \phi^*, \\ \mathsf{f\text{-}empty}(\phi^*) & = & \mathsf{true}, \\ \mathsf{create\text{-}focus}(\epsilon[\mathsf{tree}],\ n) & = & \phi \end{array}}{\mathsf{current\text{-}focus}(\epsilon,\ n) \ = \ [\phi]}$$

2.2.9 Moving a Focus

This section specifies the *repositioning* of a Focus. The current Focus may be moved through the parse tree.

The Module FocusMove

module FocusMove
imports Editor$^{2.1.7}$
exports
  sorts Move
  variables
    $m[0\text{-}9']*$ $\rightarrow$ Move

Directions for Moving

In order to reposition the Focus we introduce a new sort Move specifying the directions the Focus can be moved to. They are, not surprisingly: up, down, right and left. See Section 2.2.2 for examples of moving the Focus.

context-free syntax
  up $\rightarrow$ Move
  down $\rightarrow$ Move
  left $\rightarrow$ Move
  right $\rightarrow$ Move

Moving a Focus

The constants of the sort Move are used in the function move. This function takes the current Focus from the Editor and moves it in the appropriate direction.

context-free syntax
  move(Editor, Move) $\rightarrow$ Editor

After retrieving the Path from the Editor, we construct a Path to its parent. We then try to construct the subtree pointed to by this new Path. If it exists (it is not an error, as it would typically be if the original Path would have been a Path to the root), we construct the new focus, and put it back ino the Editor.

equations

$$[\text{m-u-v}]\ \frac{\begin{array}{rcl} \epsilon[\text{focus}][\text{head}][\text{path}] & = & \pi, \\ \pi\uparrow & = & \pi', \\ \epsilon[\text{tree}]_{\pi'} & = & t, \\ t & \neq & \texttt{"error"}, \\ \text{create-focus}(\epsilon[\text{tree}], \pi') & = & \phi, \\ \epsilon[\text{focus}][\text{tail}] & = & \phi^* \end{array}}{\text{move}(\epsilon,\ \text{up}) \ = \ \epsilon[\text{focus} := \phi \mathbin{+\!\!+} \phi^*]}$$

If moving the Focus up is not possible, we return the original Editor.

$$[\text{m-u-e}]\quad \frac{\begin{array}{rcl}\epsilon[\text{focus}][\text{head}][\text{path}] &=& \pi,\\ \pi\uparrow &=& \pi',\\ \epsilon[\text{tree}]_{\pi'} &=& \texttt{"error"}\end{array}}{\text{move}(\epsilon,\ \text{up}) = \epsilon}$$

Moving a Focus down is slightly more difficult than moving it up. If we move the Focus to its first child, we have to make sure that this child does exist, and it is the proper subterm for a Focus (an application or a list). If this is the case, we then construct a new Focus, and put it back into the FocusList.

$$[\text{m-d-v}]\quad \frac{\begin{array}{rcl}\epsilon[\text{focus}][\text{head}][\text{path}] &=& \pi,\\ \pi\downarrow &=& \pi',\\ \epsilon[\text{tree}]_{\pi'} &=& t,\\ t &\neq& \texttt{"error"},\\ \text{is-cons}(\text{appl},\ t) \lor \text{is-cons}(\text{list},\ t) &=& \text{true},\\ \text{create-focus}(\epsilon[\text{tree}],\ \pi') &=& \phi,\\ \epsilon[\text{focus}][\text{tail}] &=& \phi^*\end{array}}{\text{move}(\epsilon,\ \text{down}) = \epsilon[\text{focus} := \phi +\!\!+ \phi^*]}$$

If there is no node under the current one (the current node is a leaf node) we will return the previous Editor.

$$[\text{m-d-e}]\quad \frac{\begin{array}{rcl}\epsilon[\text{focus}][\text{head}][\text{path}] &=& \pi,\\ \pi\downarrow &=& \pi',\\ \epsilon[\text{tree}]_{\pi'} &=& \texttt{"error"}\end{array}}{\text{move}(\epsilon,\ \text{down}) = \epsilon}$$

If there *is* a child of the current node, but it is not the right type of subterm, we proceed with the next child of the node. Consider the Focus in this example (in While):

```
x := 10;
while x > 0 do
  x := x - 1;
  y := y + 1
```

The underlined part of the program is the Focus. This Focus corresponds to the production rule

```
"while" Exp "do" Stat  ->  Stat
```

The first child of this Focus would therefore be the lexical element `while`. `while` is a terminal symbol in this language, so not a candidate for a Focus (Focuses can only be on non-terminal symbols). If such a symbol is encountered we try the next child of the original Focus: in this case that is an expression (`Exp`), which is a valid type for a Focus.

If the second child is again a terminal symbol the process is repeated until a nonterminal symbol is found, or, until there are no children left. In the latter case the sort of the Focus will be set to error, so the calling function can detect that something has gone wrong.

$$\text{[m-d-i]}\ \frac{\begin{array}{rcl} \epsilon[\mathsf{focus}][\mathsf{head}][\mathsf{path}] & = & \pi, \\ \pi\downarrow & = & \pi', \\ \epsilon[\mathsf{tree}]_{\pi'} & = & t, \\ t & \neq & \texttt{"error"}, \\ \mathsf{is\text{-}cons}(\mathsf{appl},\ t) \vee \mathsf{is\text{-}cons}(\mathsf{list},\ t) & = & \mathsf{false}, \\ \mathsf{create\text{-}focus}(\epsilon[\mathsf{tree}],\ \pi')[\mathsf{sort} := \mathsf{error}] & = & \phi, \\ \epsilon[\mathsf{focus}][\mathsf{tail}] & = & \phi^* \end{array}}{\mathsf{move}(\epsilon,\ \mathsf{down}) \ = \ \mathsf{move}(\epsilon[\mathsf{focus} := \phi \mathbin{+\!\!+} \phi^*],\ \mathsf{right})}$$

The algorithm for moving the Focus right is not unlike that for moving the Focus down. First, there is the case that indeed there is a node to the right of the current Focus, and its a nonterminal.

$$\text{[m-r-v]}\ \frac{\begin{array}{rcl} \epsilon[\mathsf{focus}][\mathsf{head}][\mathsf{path}] & = & \pi, \\ \pi\rightarrow & = & \pi', \\ \epsilon[\mathsf{tree}]_{\pi'} & = & t, \\ t & \neq & \texttt{"error"}, \\ \mathsf{is\text{-}cons}(\mathsf{appl},\ t) \vee \mathsf{is\text{-}cons}(\mathsf{list},\ t) & = & \mathsf{true}, \\ \mathsf{create\text{-}focus}(\epsilon[\mathsf{tree}],\ \pi') & = & \phi, \\ \epsilon[\mathsf{focus}][\mathsf{tail}] & = & \phi^* \end{array}}{\mathsf{move}(\epsilon,\ \mathsf{right}) \ = \ \epsilon[\mathsf{focus} := \phi \mathbin{+\!\!+} \phi^*]}$$

Then, there is the case that there is no node to the right of the current Focus. The original Editor is returned.

$$\text{[m-r-e]}\ \frac{\begin{array}{rcl} \epsilon[\mathsf{focus}][\mathsf{head}][\mathsf{path}] & = & \pi, \\ \pi\rightarrow & = & \pi', \\ \epsilon[\mathsf{tree}]_{\pi'} & = & \texttt{"error"} \end{array}}{\mathsf{move}(\epsilon,\ \mathsf{right}) \ = \ \epsilon}$$

Finally, there is the case that there *is* a node to the right of the Focus, only it is not a nonterminal symbol. We then call this function again, to proceed to move the Focus right, until there is a nonterminal (or an error, as described above).

$$\text{[m-r-i]}\ \frac{\begin{array}{rcl} \epsilon[\mathsf{focus}][\mathsf{head}][\mathsf{path}] & = & \pi, \\ \pi\rightarrow & = & \pi', \\ \epsilon[\mathsf{tree}]_{\pi'} & = & t, \\ t & \neq & \texttt{"error"}, \\ \mathsf{is\text{-}cons}(\mathsf{appl},\ t) \vee \mathsf{is\text{-}cons}(\mathsf{list},\ t) & = & \mathsf{false}, \\ \mathsf{create\text{-}focus}(\epsilon[\mathsf{tree}],\ \pi')[\mathsf{sort} := \mathsf{error}] & = & \phi, \\ \epsilon[\mathsf{focus}][\mathsf{tail}] & = & \phi^* \end{array}}{\mathsf{move}(\epsilon,\ \mathsf{right}) \ = \ \mathsf{move}(\epsilon[\mathsf{focus} := \phi \mathbin{+\!\!+} \phi^*],\ \mathsf{right})}$$

Moving the Focus left is exactly the same as moving the Focus right.

$$
[\text{m-l-v}]\quad \frac{\begin{array}{rcl} \epsilon[\text{focus}][\text{head}][\text{path}] & = & \pi, \\ \pi\leftarrow & = & \pi', \\ \epsilon[\text{tree}]_{\pi'} & = & t, \\ t & \neq & \texttt{"error"}, \\ \text{is-cons}(\text{appl}, t) \vee \text{is-cons}(\text{list}, t) & = & \text{true}, \\ \text{create-focus}(\epsilon[\text{tree}], \pi') & = & \phi, \\ \epsilon[\text{focus}][\text{tail}] & = & \phi^* \end{array}}{\text{move}(\epsilon, \text{left}) \;=\; \epsilon[\text{focus} := \phi +\!\!+\, \phi^*]}
$$

$$
[\text{m-l-e}]\quad \frac{\begin{array}{rcl} \epsilon[\text{focus}][\text{head}][\text{path}] & = & \pi, \\ \pi\leftarrow & = & \pi', \\ \epsilon[\text{tree}]_{\pi'} & = & \texttt{"error"} \end{array}}{\text{move}(\epsilon, \text{left}) \;=\; \epsilon}
$$

$$
[\text{m-l-i}]\quad \frac{\begin{array}{rcl} \epsilon[\text{focus}][\text{head}][\text{path}] & = & \pi, \\ \pi\leftarrow & = & \pi', \\ \epsilon[\text{tree}]_{\pi'} & = & t, \\ t & \neq & \texttt{"error"}, \\ \text{is-cons}(\text{appl}, t) \vee \text{is-cons}(\text{list}, t) & = & \text{false}, \\ \text{create-focus}(\epsilon[\text{tree}], \pi')[\text{sort} := \text{error}] & = & \phi, \\ \epsilon[\text{focus}][\text{tail}] & = & \phi^* \end{array}}{\text{move}(\epsilon, \text{left}) \;=\; \text{move}(\epsilon[\text{focus} := \phi +\!\!+\, \phi^*], \text{left})}
$$

Chapter 3

Interconnecting Software Components

3.1 Interaction between Tools

Now that we have a specification of how a structure editor should behave, we would like to create an actual working implementation of a hybrid text/structure editor. As we described in Section 1.4.2 we can compile the specification as presented in the previous chapter into an executable program. This program can rewrite all the functions in the specification to their normal-form. Obviously, this is not the sort of user-interface we would want for text and structure editing. Besides our structure editor, we also need (at least) a text editor, and a parser. More important, we need a mechanism to connect these three tools in such a way that they can communicate with each other.

3.1.1 The ToolBus

The ToolBus [BK96a, BK96b] was designed to make software components communicate with each other. It is based on the idea that large, monolithic computer software systems are difficult (and sometimes impossible) to understand and maintain. Because the demands posed on almost any program change over time (for instance, because the hardware on which the program is running is changed, or the user expects its software to incorporate new features), the program itself needs to be changed. This proves to be more difficult as the program grows larger, and at a certain point becomes (economically) impossible. At that point it is cheaper to throw the software away and rewrite it from scratch than it is to change the existing software.

Yet another point against large monolithic software systems is that there is not one programming language best suited for programming all functionality of a large system. With the advent of more and more special purpose languages, it becomes much easier (and faster) to program specific parts of a system in specific languages. For instance there are dedicated languages for writing user-interfaces, for writing mathematical functions, for writing (or generating) images, and so on. The ToolBus attempts to solve the communication problem by providing a uniform way for tools (software programs) to communicate with each other.

The ToolBus is a software bus, that allows software tools to communicate with each other, analogous to the more familiar hardware bus (as present in any modern computer) that allows hardware tools to communicate with each other. The ToolBus uses a protocol to arbitrate the communication over the bus. Communication over the ToolBus is either synchronous (messages) or asynchronous (notes). To manage and enforce the communication, the ToolBus must be programmed with so called ToolBus scripts (or T-scripts) that define the number of processes that are active inside the ToolBus. T-scripts also define how and when the processes inside the ToolBus communicate with each other, and how and when these processes communicate with the tools that are "plugged in" the ToolBus. T-scripts are based on discrete time process algebra [BB96]. T-scripts consist of actions and operators on these actions. The main actions in a T-script are send, receive, and create. The main operators are . for sequential composition, + for alternative composition, || for parallel composition, and * for repetition. The full specification of the ToolBus script language can be found in [BK96a]. A more detailed discussion of ToolBus scripts is outside the scope of this thesis, we will therefore give a very small example, just to get an impression of the script language. Consider the following pseudo T-script:

```
send(d1).receive(d2)
```

This script first sends a data-element `d1`, and then receives a data-element `d2`. Consider now another pseudo T-script:

```
receive(d1).send(d1)
```

This script starts by receiving a data-element. Then it sends the same element out. Assuming that there is a communication between `send` and `receive` we can now put these two scripts together to get:

```
send(d1).receive(d2)      ||       receive(d1).send(d1)
```

Now we have two processes running in parallel with each other. Because the communication between send and receive is defined to be *synchronous*, we now have a way to transfer data from one process to the other. In this example, suppose the first process executes the send action. Because of the synchronous communication, it can only execute this send if there is another process that simultaneously executes a receive. Luckily, in this example, that is the case. Now both processes have performed their first action. The first process now has to execute a receive, whereas the second process has to execute a send. Note that the data-element the first process receives is a different one from the one it sent. Normally, between receiving data and sending it out again, some operation is performed on the data. This is (in a very simple form) the way T-scripts usually work. They receive data, they perform some operation on the data, and they send the data out again. The recipient or sender of data can be either another ToolBus process, or a software tool that is attached to the ToolBus.

We will give a more elaborate example of a real T-script in Section 3.2.2.

To provide a uniform interface to tools written in any programming language, we need small translation programs, called ToolBus adapters. These adapters transform messages from the ToolBus into an appropriate format for a specific programming language, and provide programs with functions and procedures for accessing the ToolBus.

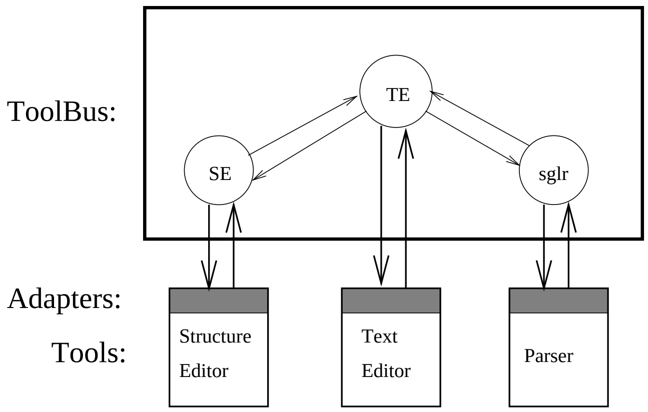

Figure 3.1: The connections between the tools and their corresponding processes in the ToolBus

3.2 The Tools

We will discuss the individual tools we connect to the ToolBus in order to get our structure editing system. We will give example (pseudo) ToolBus scripts for the interaction of three such tools, a text editor, a structure editor and a parser. Their interconnection is displayed in Figure 3.1. Two other tools, a structure manager and a source code typesetter, are discussed as well.

3.2.1 The Structure Editor

The main tool we want to connect to the ToolBus is the structure editor. In order to connect this specification, we first need to compile it to ARM code, using the AsFix compiler ac(1)[1]. The ARM-code we have generated can be interpreted by the ARM-interpreter [KW93, KW95]. The ARM-interpreter can work as a stand-alone interpreter, in which case it takes a term in AsFix format as its input, and produces the normal form of that expression, again in AsFix format. It can also operate as a tool in the context of the ToolBus, using the ARM-adapter. The ARM-adapter provides an interface to the ToolBus for the ARM-interpreter. This allows ToolBus processes to send arbitrary terms to the ARM-interpreter which it then rewrites to normal-form and returns to the ToolBus.

At the time of writing this thesis the ARM-adapter is in a very experimental phase. The interface of the adapter, as seen from the ToolBus is not yet defined. We will therefore not give a formal definition of the ToolBus process that interacts with the ARM-interpreter. We will, however, describe the proposed functionality of such a process.

We consider all functions that are specified in the module Structure-Edit (see Section 2.2.2) to be accessible from the ToolBus. These functions can be considered as the ToolBus interface for the structure editor. Note that as

[1] Created by P.A. Olivier and M.G.J. van den Brand at the Programming Research Group of the University of Amsterdam

there is currently no means of hiding functions from the ToolBus (by the ARM-interpreter), any function defined anywhere *can* be called from the ToolBus.

We assume for this explanation that the sort Literal in the specification is equivalent to the type `str` in the ToolBus, and that the sorts Nat and Int can be easily transformed into the ToolBus type `int`.

To explain the actions taken for communication between the ToolBus and the actual structure editing tool (the ARM-interpreter), we have written a very basic, pseudo ToolBus script (as shown in Figure 3.2). It starts with a declaration of the name of the process (SE) and all the variables, followed by the atomic action `create` that starts the tool associated with this script. The command that is used to start the tool is defined in the last line of the script.

The first editing function that will be called is new-editor. This returns a term of type Editor. This term will contain a Focus, as defined in Section 2.1.2. In this Focus the character position of the text corresponding to this Focus is encoded. This position will be send to the text-editor process, using a synchronous message. (If other tools would be interested in the Focus position, we could use the asynchronous broadcast system of the ToolBus by sending a *note* saying where the new Focus is). The text editor can then take appropriate action, by, for instance, highlighting the Focus in the text.

The Editor data-element that was just returned by the structure editor is saved in the ToolBus to be used by other functions. For instance, when a message arrives at the structure editing process that says that a character has just been inserted in the text editor, the structure editing process uses the Editor term, the inserted characters and the location of these characters as received from the text editor, to create the function insert-chars (or delete-chars, if appropriate) and send that to the structure-editing tool (the ARM-interpreter).

Again, the structure-editing tool will return a term of type Editor. Because this is not the first time an Editor is created (that was with new-editor), the term can now contain more than one Focus. As stated before, we will consider the *first* Focus in a FocusList to be the *primary* Focus. That is, if an operation on a single Focus is requested, and more than one Focus exist, this operation will be performed on the first Focus. Because of this order on Focuses, it is appropriate for the text editor to display the primary Focus differently from the secondary Focus.

Once the term containing the Editor is received, a number of options are available. These depend on how exactly the ARM-adapter will be implemented. If we have the possibility to retrieve subterms of a term inside the ToolBus, we can retrieve the FocusList part of the Editor. From the FocusList we can then retrieve the separate Focuses, and, from them, their respective position information. This position information can then be send to the text editing tool.

If we cannot retrieve subterms from terms inside the ToolBus, we can use the access function we specified on Editor to let the ARM-interpreter retrieve the Focuses. We can then send the whole FocusList as one term to the text-editor, and let the text editor tool find the position information itself. We could also define a new function in the specification of the structure editor, that will only be used as an interfacing function, to combine a number of access functions on an Editor that returns exactly a list of Area pairs, corresponding with the Focus positions.

All the functions defined as ToolBus interface to the structure editor return

```
process SE is
let
  SE: se, E: term, Move: term, Flist: term,
  Pt: term, Loc: int, Str: str
in
  execute(se, SE?).
  ( rec-msg(new-edit(Str?)).
    snd-eval(SE,new-editor(Str)).
    rec-value(SE, E?).
```

Extract focus information, and put in `Flist`

```
    snd-msg(new-foc(Flist)).
    ((( rec-msg(char-insert(Loc?,Str?)).
        snd-eval(SE,insert-chars(E,Loc,Str)). )
      +
      ( rec-msg(char-delete(Loc?).
        snd-eval(SE,delete-chars(E,Loc)). )
      +
      ( rec-msg(foc-move(Move?)).
        snd-eval(SE,move-focus(E,Move)).
        rec-value(SE,E?). )
      +
      ( rec-msg(rep-foc(Foc?)).
        snd-eval(SE,replace-focus(E,Foc)) )
      +
      ( rec-msg(rep-meta).
        snd-eval(SE,replace-with-meta(E)) ) ).
    rec-value(SE,E?).
```

Extract focus information, and put in `Flist`, extract parse tree, and put in `Pt`.

```
    snd-msg(new-foc(Flist)).
    snd-note(new-pt(Pt)) )*
  delta
endlet
tool se is {command = "The name of the ARM-interpreter"}
```

Figure 3.2: A T-script for interacting with the Structure Editing Tool

a term of type Editor. As we just described, from such a term we can retrieve Focus information to send to the text editor for displaying. However, we can get more information from an Editor term. We can retrieve the parse tree of the Editor, which can be passed to an interpreter or compiler tool, or saved to disk for later use. The parse tree can also be send to a parse tree viewer, such as the one described in Section 3.2.5. In our example script, the parse tree is broadcasted using the `snd-note` mechanism in the ToolBus. Any tool can now receive this note, and decide to do something with it or to ignore it.

As stated before, this is a very simple scenario. It assumes, for instance, that only one Editor term per session exist, although there is nothing that stops us from editing more then one parse tree at a time. In that case we will have to index the variable `E` in our script with some unique identifier to distinguish it from the other Editors.

3.2.2 The Text Editor

Apart from a structure editing tool, another tool needed in a text/structure editing system is a text editor. A text editor is any system that can manipulate text. Examples of text editors include vi(1), emacs(1), but also Microsoft Word and Corel WordPerfect. They range from the very simple to the very elaborate.

In order to use a text editor with the ToolBus, or more specifically, with our structure editing system, an editor needs to be extendible with a number of procedures. These procedures allow the ToolBus to communicate with the text editor using a well-defined interface. The exact procedures that need to be understood by the text editor are in Appendix D.

We have currently implemented the text editing tool using jedit(1), a freely distributed text editor written in Tcl/Tk [Ous94]. This implementation is a prototype. It it mainly used to demonstrate the different procedures we use in communicating with the ToolBus.

Communication from the ToolBus

Apart from the obvious procedures that open specific files, create new windows, and create menus and buttons, there are a number of less obvious procedures defined. We will try to motivate these functions here. The procedure `TB_set-status` is used to send short messages to the user. The text editor should implement some sort of status field, a small section of the user interface that is used exclusively to give feedback on progress of the current action, encountered errors, and so on.

There is a separate status field, called *sort*, to display the sort of the current Focus. The text string in this field is set via the procedure `TB_set-sort`. The ToolBus script makes sure that text in this field is always consistent with the actual Sort of the Focus.

We can use the function `TB_highlight` to highlight, or mark, a range of characters. Highlighting is a general mechanism to draw the attention of a user to a specific piece of source text. Together with the status field, we can use this mechanism for any number of purposes. The main purpose would of course be to show the current Focus of the structure editor. Another use would be to highlight a piece of code where a parse error has occurred. Currently, only one highlighting function is implemented. This means that, although we can have more than one Focus, we can only highlight one Focus at a time. This is done deliberately. For a different approach, see Section 3.2.5 where a tool is described that allows for a more diverse way of marking text.

Apart from a generic procedure to define menus (`TB_make-local-menu` and `TB_make-global-menu`) there is one special menu that does always exist. It is the Language menu, from which a user can select a language. All text subsequently typed in the current window is considered to be a term over that language. In order for the text editor to know which languages are available, the procedure `TB_register-lang` is defined. Any language that can be parsed should also be passed to the text editor using this procedure.

The user can decide at any moment that he would like to parse the current text. Because currently our parser can handle parse requests only sequentially (only one at a time) we have introduced the special procedures `TB_enable-parse` and `TB_disable-parse`. After a `TB_disable-parse` no parse request can be sent from the text-editor until a `TB_enable-parse` is received.

Finally, the ToolBus can request an arbitrary string from the editor. That is, the ToolBus can (using `TB_get-string`) ask for the string that starts at character location x and is y characters long. Together with the generic menu creation function we can use this function to construct substrings to be parsed. For instance, a menu item "Parse Focus" could be created that sends the function parse-focus to the ToolBus. The ToolBus can then ask the structure editing process where the current Focus starts, and where it ends. These characters can be retrieved from the text editor subsequently, and sent to the parser.

Messages to the ToolBus

Using the generic menu mechanism, other functionality can be implemented in the text editor, without having to change the actual code of the editor. The Focus moving function as described in Section 2.2.2 can have its user interface implemented this way. We can use the procedure `TB_make-global-menu` to create a menu "Move" with four items, one for each direction we can move in. Each menu button can now send a specific term back to the ToolBus once it is invoked. For instance we can have a menu item labeled "Up" that sends the term `focus-up` to the ToolBus. The text editing process can then send a message to the structure editing process to move the Focus up.

There is one function that needs to be hardcoded in the text editor, however. To maintain consistency between the text that is being edited and the structure that is being edited, we have to have a function that lets the ToolBus know that the text has been changed in any way. Currently, we do this by sending a message to the ToolBus every time a character is inserted or deleted. This corresponds with the function insert-chars and delete-chars as they are defined in Section 2.2.2. After a number of experiments (none of them conclusive) we are led to believe that such behaviour will cause unacceptable delays during editing. (See Section 4.1 for details). As you may have noticed, the function insert-chars supports the insertion of more than one character. We could use this mechanism to delay the sending of *insert character* messages. For instance, we could save any character string that was typed in succession, only to transmit this information to the ToolBus if a user moved the cursor elsewhere in the text, or paused typing for a while. This functionality is currently not implemented, and we need to experiment to see what is the best strategy here.

```
process Text-edit is

let
    TE: text-edit, Id: str, Id2:str, L : str, Str: str,
    Sort: str, T1: term, T2: term, Ch: int, Loc:int,
    Start: int, End: int, Flist: term
in
    subscribe(language-available(<str>)).
    execute(text-edit, TE?) .
    (   rec-event(TE, insert-char(Ch?,Str?,Id?)).
        snd-msg(char-insert(Ch,Str)).
        rec-msg(new-foc(Flist)).
```

Now we have a mechanism to extract the location of the first Focus. The coordinates are put in `Start` and `End`. The Sort of the Focus is put in `Sort`

```
        snd-do(TE,TB_highlight(Id, Start, End)).
        snd-do(TE,TB_set-sort(Id,Sort).
        snd-ack-event(TE,insert-char(Ch,Str,Id))
        +
        rec-event(TE,parse-focus(L?,Id?)).
        snd-do(TE,TB_disable-parse(Id)).
```

We now get the Focus location from the structure editing process. The coordinates are put in `Start` and `End`.

```
        snd-eval(TE,TB_get-string(Id,Start,End)).
        rec-value(TE,string(Str?)).
        snd-msg(parse-string(L,Str,Id)).
        (   rec-msg(parse-result(Id,parsetree(T1?,T2?))).
            snd-do(TE,TB_set-status(Id,"Parsing succeeded")).
            snd-msg(rep-foc(T1)).
            rec-msg(new-foc(Flist))
            +
            rec-msg(parse-result(Id,eof-error)).
            snd-do(TE,TB_set-status(Id,"End of string unexpected"))
            +
            rec-msg(parse-result(Id,
                    parse-error([character(Ch?),pos(Loc?)]))).
            snd-do(TE,TB_set-status(Id,"Unexpected character").
            snd-do(TE,TB_set-cursor(Id,Li,Co)).
        snd-do(TE,enable-parse(Id)).
        snd-ack-event(TE,parse-string(L,Str,Id))
        +
        rec-note(language-available(L?)).
        snd-do(TE,registerlang(L))   ) *
    rec-event(TE,quit).
    shutdown("Editor terminated")
endlet

tool text-edit is {command = "wish-adapter -wish wish4.1
 -script sted.tcl"}
```

Figure 3.3: An example of a ToolBus script for the text editing process

Interaction

As an example of a (possible) interaction between the text editor, the parser and the structure editor we refer the reader to Figure 3.3. In this figure a part of a ToolBus script is given that controls the text editor, and performs communication with the other tools. The script starts by executing the text editing tool, and subscribing to messages of the type `language-available`. After this, there is an alternative composition of three events. As explained in the previous section, more events are possible. We do not attempt to provide a full T-script here, but merely want to demonstrate what a T-script looks like, and how communication between the different processes can be defined.

The first event in the script is the `insert-char` event. The script communicates this event to the structure editing process, using the term `char-insert`. Note that the structure editing script (Figure 3.2) indeed receives this term. The structure editor then sends a new FocusList back. From this FocusList the Sort and position information of the first Focus is retrieved and sent back to the text editing tool.

The second event, `parse-focus`, shows both interaction with the parser, and with the structure editor. It activates the parser to parse the string of the current Focus. If the parse succeeds (no syntax errors are encountered), then the parse tree that resulted from the parse is sent to the structure editor to replace the parse tree the current Focus points to. This is done using `rep-foc`. The structure editor returns the new FocusList, which is received, but ignored by the text editing process. (This is done for brevity, we could now highlight the new Focus, or remove the Focus altogether, since it has been successfully parsed).

The last event is an example of broadcast communication. Receiving the note `language-available` means that a new language has become available. This note can be sent from any tool that has introduced a new language to the system. In this configuration, the parser is the only tool that sends out these notes. (See Figure 3.4).

3.2.3 The Parser

Our structure editor is language independent because we use a parse tree encoding that abstracts from the actual syntax: AsFix. Through the use of this format we have moved the language dependency to the parser. The structure editing tool expects its input to be in AsFix, no matter what language it represents. The parser therefore has to be able to parse a language into AsFix. If our claim of language independence is to be correct, we would like to have a number of parsers, for a number of languages. In fact, we would like to have a parser for all languages, making the structure editing system as a whole (text editor, parser and structure editing tool) truly language independent.

Although we do not have a parser for all languages, we do have a parser for all context-free languages. To be more precise, there is a parser *generator* that can generate an AsFix parser from any context-free language definition in SDF[Vis97].

Communication from the ToolBus

Unsurprisingly, the number of functions supported in the parser is very limited. They are `parse-string` and `parse-file`. `parse-file` is mainly there for completeness and compatibility reasons; as the ToolBus is a distributed system, we can never be sure that a file that is visible to one tool can also be seen by another tool. Instead, all strings are supplied via the ToolBus and parsed using the function `parse-string`.

The parser has one other function: `open-language` that allows the ToolBus to tell the parser to read a language definition. (Actually, the parser reads in a parse table, and this parse table can be generated from any context-free syntax definition.) Currently, the language(s) that will be opened should be hardcoded in the ToolBus-script. We obviously need a more flexible way of doing this. The problem is that we need a way to find out which languages are available. We could write a separate tool (a "Syntax Manager") that keeps track of the different languages.

Messages to the ToolBus

After the parser has received a parse request (from the ToolBus), it parses the appropriate string and returns the parse tree that is the result. However, parsing sometimes fails, due to syntactic errors in the string to be parsed. If this is the case the parser returns a parse error as its result. See Figure 3.3 for the exact syntax of parse errors.

The parser also needs to have a mechanism to reply to the ToolBus if an `open-language` request has succeeded. It can either reply with the term `language-opened`, or with the term `open-language-failed`.

Interaction

An example of the use of the functions described in the previous section is provided in Figure 3.4, an example ToolBus script of the parsing process.

Apart from the use of the functions, the script also shows the interaction between the parse tool and the other tools in the ToolBus. The script can receive three messages from other processes.

The first message is the most common: `parse-string`. It receives a string, and a language, and parses the string over the language. A term containing the parse result (either succeeded or failed) is returned from the parse tool. Finally, the parse result is communicated to another process using `parse-result`. Note that the script in Figure 3.3 (The text editing process) contains the matching receive action.

The second message is currently not used. That is, there is no script that sends the message `parse-file`, and, as a result, this message is never received. If this message was received, however, it would act very much like `parse-string`.

The third message that can be received is `open-language`. As stated in the previous section, we currently need to hardcode the sending of this message in a ToolBus script. For instance, we could create a single initialization process that takes care of these messages. Upon receiving a `open-language` message, the parse process tells the parse tool to read in that language. The parse tool then returns a term that states whether or not the opening of the language definition

```
process Parser is
  let
    P  : sglr, L  : str, S  : str, Id : term, FN : str, T  : term
  in
    execute(sglr, P?) .
    ( rec-msg(parse-string(L?, S?, Id?)) .
      snd-eval(P, parse-string(L, S)) .
      rec-value(P, T?) .
      snd-msg(parse-result(Id, T))
    +
      rec-msg(parse-file(L?, FN?, Id?)) .
      snd-eval(P, parse-file(L, FN)) .
      rec-value(P, T?) .
      snd-msg(parse-result(Id, T))
    +
      rec-msg(open-language(L?, FN?)) .
      snd-eval(P, open-language(L, FN)) .
      (
    rec-value(P, language-opened(L?, FN)) .
    snd-note(language-available(L))
      +
    rec-value(P, open-language-failed(L, FN))
      )
    )
    * rec-disconnect(P)
  endlet
tool sglr is {command = "/home/visser/syntax/sglr/src/sglr"}
```

Figure 3.4: An example of a ToolBus script for the parsing process

file has succeeded. If it was successful, the parse process broadcasts a note to all processes containing the new language. If opening the language failed, then no further action is taken.

3.2.4 A Structure Manager

We stated at the beginning of this thesis that we set out to make a language independent structure editor. We have now shown that we indeed specified a language independent structure editing tool, but have also noted (in Section 2.2.2) that this language independence prohibits us from performing certain operations. Particularly, we cannot perform any type-checking and we cannot see the structure of language constructs.

The first problem occurs when we are replacing a subtree of the parse tree. In order to make sure that the new parse tree is a correct tree, we have to know if the subtree that we want to place in the tree has the same sort, or is a subsort of, the subtree we want to replace. From the current system we cannot see this, because we do not have access to a language definition.

The second problem occurs when expanding meta-variables. We would like to be able to expand a meta-variable to the lefthand side of any production rule that has a sort corresponding to the sort of the meta-variable. For instance if we have the meta-variable $\langle\text{Bool}\rangle$, we want to be able to replace it by $\langle\text{Bool}\rangle\vee\langle\text{Bool}\rangle$.

By creating a "Structure Manager" tool, we can solve these problems. The structure manager has knowledge of the syntactical structure of the language. It can obtain this knowledge from the language definition. The structure manager will then answer queries on the structure of this language.

Because the two problems described above are exactly the two problems we need this tool for, the tool only has to contain two procedures. One procedure should simply return a boolean when asked if sort A is a subsort of sort B. The other procedure should return a list of all (lefthand sides of) production rules that produce a certain sort. This list can then be passed to the user and the user can then decide which production rule he wants to use for expanding a meta-variable.

We do not currently have a structure manager. Even without such a tool, the current system is quite useful. The first problem we noted above is not as bad as it seems. Although we cannot check whether a sort is a subsort of another sort, we *can* check if two sorts are equal. (Although this currently is not done in the specification, it is simple to extend the specification with this capability). If the parser always returns trees that contain the topmost sort of a term, then we have more or less the same check. As an example consider the following syntax definition (again in SDF):

```
"a"  ->  A
 A   ->  B
 B   ->  C
```

Suppose a user decides to replace a subtree of sort `C` with the subtree produced by parsing the string "`a`". If the parse tree produces the smallest possible parse tree for `a` then the start symbol of that tree will be `A`. The structure editor is unable to decide if `A` is a subsort of `C`. If the parser produces the largest possible parse tree, then the startsymbol of the tree will be `C`. The two sorts are equal, and we can safely replace the subtree.

If however, a user decides to replace a subtree of sort B, then both parsing strategies will fail. A tool like this structure manager is therefore needed. This is also why the above described equality check on sorts is not specified for the structure editor.

A structure manager is also a necessity for the re-implementation of the ASF+SDF meta-environment.

3.2.5 A Tool for Formatting Source Text

We mentioned in Section 1.4.1 that we want to be able to reconstruct the exact source text from the parse tree. Reconstructing such a source text is done by a separate tool, called `asource` for "AsFix source". This tool can be connected to the ToolBus in the same way as the other tools. It implements a function `asource` that takes a parse tree in AsFix format as its input, and returns a string containing the exact source text that was used to create the parse tree.

Since the text in the text editor already *is* the source text for the parse tree we are editing, there is no need to call `asource` to reconstruct the text. It is in fact argued that one of the merits of having a hybrid editor is not needing an unparser [Koo94].

The unparse tool could be used as a test tool, to make sure that the parse tree indeed contains enough information to recreate the original source text. It could be used to do some form of consistency check every once in a while.

If the source text gets lost, or gets corrupted, we could use `asource` to retrieve the text. Since the parse tree only contains the text as it existed during the last parse action, it could also be used as a (crude) undo mechanism.

A more serious and interesting use we can put this tool to is formatting. We could have a separate tool that annotates parse trees with formatting information. Such a tool could for instance mark all keywords in a language, or mark all occurrences of variables of a specific type (see [BKV96] for a more elaborate example). We have written a version of `asource` that can recognize special annotations to subterms, and, depending on the annotation, can return text that is formatted in a specific colour, in a specific type font, or a specific type style (or any combination thereof). This text (together with the formatting information) can then be send via the ToolBus to the text-editor. The current implementation of the text editor has a special set of procedures to display formatted text.

Chapter 4

Conclusion

4.1 Results

In this thesis we have presented a specification for a structure editor, and given a model for a structure editing system. We have described what tools are needed for such a system.

4.1.1 Tools

The specification of the structure editor as it is presented in Chapter 2 has been successfully executed within the ASF+SDF Meta-Environment. We have compiled the specification into ARM-code, and used the ARM-interpreter to interpret this code.

We have simulated editing sessions using the ARM-interpreter, by creating terms that represent functions on the Editor datatype, and have the ARM-interpreter rewrite these terms. We have performed editing operations of limited complexity, and the results were satisfactory.

We have implemented a text editor with a ToolBus interface as described in Appendix D. This editor was written in Tcl/Tk, on top of `jedit`, a extendible editor that is freely available and incorporates a number of basic features for editing.

We have implemented a tool (`ts_asource`) that retrieves source text from AsFix terms, and typesets this text according to special annotations in the term, as described in Section 3.2.5.

We have successfully used the ToolBus to establish an interconnection between the text editor and parsers generated by the parser generator. We have also established a connection between the text editor and `ts_asource`.

The ARM-interpreter cannot yet be connected to the ToolBus, because there currently is no ToolBus adapter appropriate for the ARM-interpreter.

4.1.2 Efficiency

The time efficiency of the first tests was disastrous. It took about 40 seconds for an average editing operation (such as inserting a character) to complete. After a number of improvements on both the structure editor, the AsFix compiler and the ARM-interpreter, the average editing operation now takes 8.4 seconds to complete. In this time the ARM-interpreter has to read in all the ARM files for each rewrite session. A later version of the ARM-interpreter will keep all ARM files in main memory inbetween rewrite sessions. Experiments have shown that of the 8.4 seconds an average editing operation takes, 7.9 seconds is spent reading in the ARM files. In the best case this means that rewriting the term that encodes a structure editing operation could be done in 0.5 seconds.

These experiments were performed on a Sun Microsystems Ultra 2 running at 200 Mhz, with 512 Megabyte of memory. All these times are CPU times.

The half second it will take to rewrite a term does not take into account the time the user interface needs, and the time the ToolBus will spend communicating the results. All in all, we can safely say that these results are not good enough for an interactive system.

As experienced typers can easily type 5 characters per second, we have set our goal for the time it takes the system to process a single editing operation (inserting or deleting character, for instance) to a maximum of 0.2 seconds (real time, not CPU time). A system that can process 5 characters per second will appear very sluggish, but, at least for experimental purposes, will suffice.

The text editor is implemented in Tcl/Tk, and uses standard Tk built-in text datastructures. These operations are extremely memory inefficient. When editing a file (a text string), Tk builds a datastructure containing this string, and all sorts of position, font, colour, and other information. We have seen situations where this datastructure was 10 to 15 times as large as the string that was being edited. Because Tk keeps the whole datastructure in main memory, the system becomes intolerably slow when the string gets longer. Editing a string of 10 Megabyte, or even viewing such a string, is virtually impossible. When this string needs to be sent to the ToolBus, it is first copied within Tk, doubling the amount of memory Tk needs.

However, for small files the text editor works well. We have not performed exact timing experiments when interacting with the parser, but sending a string to the parser via the ToolBus, having the parser parse it, and sending the result back to the text editor, was all done in less than one second.

The interaction with `ts_asource` worked equally well. `ts_asource` reads the whole ASFIX term it is going to unparse in memory, before performing any action. This will cause a delay when unparsing large terms.

4.1.3 Questions Answered

At the beginning of this thesis we asked a number of questions.

- Is it possible to give a formal specification of a (language-independent) structure editor?

  We have given (in Chapter 2) a specification in ASF+SDF of the *core* of a structure editing systems. We have specified the basic functions needed for structure editing. We have given (part of) a ToolBus-script specification that controls the behaviour of this structure editor.

- If such a specification can be devised, can we generate executable code from this specification, and will this code run?

  We have compiled the specification, and performed a limited number of structure editing operations. These operations were all on very small parse trees. We do not know whether the size of the parse tree affects the performance of the compiled specification.

- Can we split the structure editor up in different components, and implement/specify these components separately?

  We have given a possible way to split the system up, and implemented a number of the needed components. We have also demonstrated that these components can and do interact with each other.

- Can we connect all these components to the ToolBus and therefore have them communicate with each other?

  Not currently. We have not connected the structure editor specification to the ToolBus, because there is no suitable adapter (see above). The other tools all have been connected to the ToolBus.

- Is this combined system of tools efficient enough to allow us to use it interactively?

  Not as we can see now. There is probably some efficiency to be won in the specification of the structure editor, but it will not be much. We have to see how fast the next version of the ARM-interpreter will be before we can make a decision on whether or not the specification of the structure editor can be used as its implementation.

  We also will need a different text editor, if we want to be able to edit large, generated terms. Terms that are smaller then 100 kilobytes are easily edited with the current text editor.

4.2 Related Work

Work on structure editors has been done by Wilco Koorn, who implemented GSE [Koo94], the editor that is currently in use in the ASF+SDF meta-environment.

The Synthesizer Generator [RT89] is probably the most famous system for generating hybrid editors.

The Danish company Mjølner Informatics has a structure editor (Sif) as part of their BETA system [KLMM93]. Unfortunately it is currently only available as a prerelease. It is a commercial product.

Arefi, Hughes and Workman have done work in the field of generating structure editors for visual languages [AHW90].

Apart from these attempts to create generic or language independent systems, there are a number of structure editors for one specific language. The Lilac system [Bro91] can be viewed as such a system, for instance.

4.3 Future Work

First and foremost, we should concentrate on connecting the ARM-interpreter, and thus the structure editor specification, to the ToolBus. Before this is done, no decision can be made on the future direction (if any) of this project.

When the interpreter has successfully been connected we need to finish the ToolBus scripts that were introduced in Chapter 3. Only then can we see if our system is efficient enough, and whether our multiple focus concept will work better than a single focus concept for structure editing.

To increase efficiency, we could decide to implement the specification in a more efficient language such as C. In [Hil96] such a implementation is described.

The structure manager, as described in Section 3.2.4 should be implemented.

Extensions to existing text editors can be made, to make them adhere to the ToolBus interface for text editors we defined. It would be interesting to extend `emacs` with such functionality.

Appendix A

Modules from the structure editor specification

A.1 ATerms

imports Literals Booleans$^{1.4.2}$ Numerals
exports
  sorts AFun AName AVar ATerm ATerms ATermList
An AName can be any string starting with a capital letter
  lexical syntax
    [A-Z][A-Za-z0-9\−]$*$ $\rightarrow$ AName
  This definition follows closely that of *ToolBus terms*, see [BK96a].
  A function symbol (AFun) can be a Literal.
  context-free syntax
    Literal $\rightarrow$ AFun
  AInt represents a common integer (for reasons of brevity, the specification of Integers is left out here). A single integer is an ATerm
  context-free syntax
    AInt $\rightarrow$ ATerm
  The following four definitions are for:

1. A simple variable
2. A typed variable (where ATerm represents the type)
3. A simple result variable
4. A typed result variable

and the fifth definition says that any single variable is again an ATerm.
  context-free syntax
    AName $\rightarrow$ AVar
    AName ":" ATerm $\rightarrow$ AVar
    AName "?" $\rightarrow$ AVar
    AName ":" ATerm "?" $\rightarrow$ AVar
    AVar $\rightarrow$ ATerm

An ATerm between angled brackets is a placeholder variable, or metavariable. It represents a sort variable, so <Int> means "any object of sort Int"

context-free syntax

"<" ATerm ">" → ATerm

A list of ATerms without square brackets is an ATerms, and with brackets it is an ATermList. An ATermList itself is again an ATerm.

context-free syntax

ATerm → ATerms
ATerm "," ATerms → ATerms
"[" "]" → ATermList
"[" ATerms "]" → ATermList
ATermList → ATerm

This is the function application, with the ATerm the function symbol, and the ATerms its arguments. For instance, sort("Bool") denotes the function application sort over the literal "Bool".

context-free syntax

AFun "(" ATerms ")" → ATerm

A function application can also have zero arguments. (The constant).

context-free syntax

AFun → ATerm

Using curly braces the annotation is defined. The ATerm is the actual term, and the ATerms are an arbitrary annotation on this term. In this sepcification, for instance, we use "string"{ln("6")}, where ln("6") is the annotation for "string" (Meaning that "string" is 6 characters long). (See section A.4).

context-free syntax

ATerm "{" ATerms "}" → ATerm

Both a single ATerm and a list of ATerms can be put inside brackets.

context-free syntax

"(" ATerm ")" → ATerm {**bracket**}
"(" ATerms ")" → ATerms {**bracket**}

priorities

{AFun "("ATerms ")" → ATerm, ATerm "{"ATerms "}" → ATerm} <
AName ":"ATerm → AVar

Finally, the equality and the conditional are defined for ATerm.

context-free syntax

ATerm "≡" ATerm → Bool
if Bool then ATerm else ATerm fi → ATerm

exports

variables

T [0-9′]∗ → ATerm
Ts [0-9′]∗ → ATerms
Tl [0-9′]∗ → ATermList

AFun [0-9′]∗ → AFun
AVar [0-9′]∗ → AVar
AName [0-9′]∗ → AName
"t_i"[0-9]∗ → AInt

equations

Equality and conditional for the sort ATerm.

[eq-1] $T \equiv T \quad = \mathsf{true}$
[eq-2] $T_1 \equiv T_2 = \mathsf{false}$ **otherwise**

[if-1] if true then T_1 else T_2 fi $= T_1$
[if-2] if false then T_1 else T_2 fi $= T_2$

A.2 AsFix Utilities

module AsFix-Utilities
imports Integers WfAsFix Booleans[1.4.2] ATerms[A.1] Literals
exports
  context-free syntax
    to-int(Literal) → Int
    to-lit(Int) → Literal
    term2tree(ATerm) → ATerm

    type-of(ATerm) → ATerm
    my-is-cons(AFun, ATerm) → Bool
  variables
    lit[*0-9′*]∗ → Literal
    t[*0-9′*]∗ → ATerm
    "*t*+"[*0-9′*]∗ → ATerms
    tl[*0-9′*]∗ → ATermList
hiddens
  variables
    Char [*0-9′*]∗ → CHAR
    "*Char*+"[*0-9′*]∗ → CHAR+
    "*Char*∗"[*0-9′*]∗ → CHAR∗
equations

to-int is a function that "casts" the sort of a string from literal to nat-con. It does not check whether the string is indeed a number, so to-int("aap") is a valid call to this function, and will return the number aap.

[tin] to-int(literal(""" $Char^+$ """)) = nat-con($Char^+$)

to-lit is very similar to to-int.

[tli] to-lit(nat-con($Char^+$)) = literal(""" $Char^+$ """)

These two functions are used to put integers in an annotation of an ATerm. So far, ATerms can only consist of literals, and not integers/numerals/naturals. Adding such a basic type to ATerms would eliminate the need for the two functions above.

$$[\text{to-appl}]\ \frac{\text{get-cons}(t) = \text{appl},\quad t[\text{prod}][\text{sym}] = t'}{\text{type-of}(t) = t'}$$

$$[\text{to-lst}] \quad \frac{\begin{array}{rcl}\text{get-cons}(t) & = & \text{list},\\ t[\text{sym}] & = & t'\end{array}}{\text{type-of}(t) \; = \; t'}$$

$$[\text{get-cons-ann}] \; \text{get-cons}(t\{t'\}) = \text{get-cons}(t)$$

$$[\text{mic-t}] \quad \frac{\text{get-cons}(t) \; = \; AFun}{\text{my-is-cons}(AFun,\, t) \; = \; \text{true}}$$

$$[\text{mic-f}] \quad \frac{\text{get-cons}(t) \; \neq \; AFun}{\text{my-is-cons}(AFun,\, t) \; = \; \text{false}}$$

A.3 Meta variables

module Meta-var
imports Position-information[2.2.5]
exports
context-free syntax
create-metavar(ATerm) → ATerm
equations

For each sort in a language we consider an implicit production rule
`meta-var(Sort)   -> Symbol`
This function creates the AsFix term corresponding with that production rule. It applies it over the lexical representation: `"<" Sort ">"`. Future parsers will generate these parse trees. In this light we could also create the whole parse tree outside the specification; by just sending a command to the parser. Currently, no such parser is available. The SDF2 parser generator will generate parsers that produce these parse trees for meta-variables[Vis97].

$$[\text{cm-is}] \quad \frac{\begin{array}{rcl} \text{is-cons}(\text{sort},\, t) & = & \text{true},\\ t[\text{chars}] & = & \mathit{lit},\\ \begin{array}{r}[\text{make}(\text{l},\, [\texttt{"<"}]),\, \text{make}(\text{w},\, [\texttt{""}]),\, \text{make}(\text{l},\, [\mathit{lit}]),\, \text{make}(\text{w},\, [\texttt{""}]),\\ \text{make}(\text{l},\, [\texttt{">"}])]\end{array} & = & t_1,\\ \begin{array}{l}\text{init(prod)}\\ [\text{module-name} := \text{id}(\texttt{"Generated␣by␣Structure␣Editor"})]\\ [\text{syms} := t_1][\text{sym} := \text{meta-var}(t)][\text{attrs} := \text{no-attrs}]\end{array} & = & t_2,\\ \text{init(appl)}[\text{prod} := t_2][\text{args} := t_1] & = & t_3,\\ \begin{array}{l}\text{init(prod)}\\ [\text{module-name} := \text{id}(\texttt{"Generated␣by␣Structure␣Editor"})]\\ [\text{syms} := [t_2[\text{sym}]]][\text{sym} := t]\end{array} & = & t_4,\\ \text{init(appl)}[\text{prod} := t_4][\text{args} := t_3] & = & t_5 \end{array}}{\text{create-metavar}(t) \; = \; \text{annotate}(t_5)}$$

$$[\text{cm-ins}] \; \text{create-metavar}(t) \; = \; [\,]$$
otherwise

A.4 Some operations on Annotations

imports AsFix-Utilities$^{A.2}$
exports
We first define the function symbol for length annotations.
context-free syntax
"ln" $\rightarrow$ AFun
context-free syntax

ATerm "[" "anno" "]"	$\rightarrow$ ATermList
ATerm "[" "anno" ":=" ATermList "]"	$\rightarrow$ ATerm
"get" AFun "from" ATermList	$\rightarrow$ ATerm
"replace" AFun "with" ATerm "from" ATermList	$\rightarrow$ ATermList
get-ln(ATerm)	$\rightarrow$ Int

equations

[ga-n] $t\{t^+\}[\text{anno}] = [t^+]$
[ga-0] $t[\text{anno}] = [\,]$ **otherwise**

[sa-n] $t\{t^+\}[\text{anno} := [t^{+\prime}]] = t\{t^{+\prime}\}$
[sa-0] $t[\text{anno} := [t^+]] = t\{t^+\}$ **otherwise**

Getting or replacing the ATerm with constructor AFun from a list is undefined when there is no function with constructor AFun in the list.

[glst-n] get $AFun$ from tl = index(tl, position($AFun$, tl, zero))

[slst-n] replace $AFun$ with t from tl
= replace(tl, position($AFun$, tl, zero), t)

$$[\text{gl}]\ \frac{\text{get ln from } t[\text{anno}] = t', \quad t' = \text{ln}(lit)}{\text{get-ln}(t) = \text{to-int}(lit)}$$

A.5 Operations on Areas

module Areas
imports AreaSyntax$^{2.1.6}$ Path-construction$^{2.2.6}$ Subtrees$^{2.2.7}$
exports
context-free syntax

create-area(Path, ATerm)	$\rightarrow$ Area
area-length(ATerm)	$\rightarrow$ Int
area-start(Path, ATerm)	$\rightarrow$ Int

equations

[ar] create-area(π, t) = (area-start(π, t), area-length(t_π))

$$[\mathsf{abs\text{-}n}]\ \frac{\begin{array}{rcl}\mathsf{head\text{-}zero}(\pi) & = & \mathsf{true},\\ \mathsf{get\text{-}cons}(t) & = & \mathsf{appl}\end{array}}{\mathsf{area\text{-}start}(\pi,\ t)\ =\ \mathsf{area\text{-}start}(\pi[\mathsf{tail}],\ t[\mathsf{args}])}$$

$$[\mathsf{abs\text{-}n}]\ \frac{\begin{array}{rcl}\mathsf{head\text{-}zero}(\pi) & = & \mathsf{true},\\ \mathsf{get\text{-}cons}(t) & = & \mathsf{list}\end{array}}{\mathsf{area\text{-}start}(\pi,\ t)\ =\ \mathsf{area\text{-}start}(\pi[\mathsf{tail}],\ t[\mathsf{elems}])}$$

$$[\mathsf{ab}]\ \frac{\mathsf{head\text{-}zero}(\pi)\ =\ \mathsf{true}}{\mathsf{area\text{-}start}(\pi,\ tl)\ =\ \mathsf{area\text{-}start}(\pi,\ \mathsf{first}(tl))}$$

$$[\mathsf{ab}]\ \frac{\begin{array}{rcl}\mathsf{head\text{-}zero}(\pi) & = & \mathsf{false},\\ \mathsf{is\text{-}leaf\text{-}path}(\pi) & = & \mathsf{false}\end{array}}{\begin{array}{l}\mathsf{area\text{-}start}(\pi,\ tl)\ =\\ \mathsf{area\text{-}length}(\mathsf{first}(tl)) + \mathsf{area\text{-}start}(\mathsf{decrease\text{-}head}(\pi),\ \mathsf{rest}(tl))\end{array}}$$

A path to root always starts at character position 1

$$[\mathsf{abs\text{-}1}]\ \frac{\mathsf{is\text{-}leaf\text{-}path}(\pi)\ =\ \mathsf{true}}{\mathsf{area\text{-}start}(\pi,\ t)\ =\ 1}$$

$[\mathsf{rel}]\ \mathsf{area\text{-}length}(t) = \mathsf{get\text{-}ln}(t)$

Appendix B

Syntax of the language While

B.1 The Syntax of While

```
imports Expressions^B.2
exports
  sorts Stat
  context-free syntax
    Var ":=" Exp                       → Stat
    "skip"                             → Stat
    Stat ";" Stat                      → Stat  {right}
    "if" Exp "then" Stat "else" Stat   → Stat
    "while" Exp "do" Stat              → Stat
    "{" Stat "}"                       → Stat  {bracket}
  priorities
    "while" Exp "do" Stat → Stat > "if" Exp "then" Stat "else" Stat → Stat >
    Stat ";" Stat → Stat
  variables
    "s" [0-9']* → Stat
```

B.2 Expressions for While

```
imports Variables Values
exports
  sorts Exp
  context-free syntax
    Var              → Exp
    Value            → Exp
    Exp "+" Exp      → Exp  {left}
    Exp "−" Exp      → Exp  {left}
    Exp "<" Exp      → Exp  {non-assoc}
    Exp ">" Exp      → Exp  {non-assoc}
    Exp "=" Exp      → Exp  {non-assoc}
    "not" Exp        → Exp
```

Exp "and" Exp → Exp {**left**}
"(" Exp ")" → Exp {**bracket**}
priorities
{**left**: Exp "+" Exp → Exp, Exp "−" Exp → Exp} > {**non-assoc**:
Exp "<" Exp → Exp, Exp ">" Exp → Exp, Exp "=" Exp → Exp} >
"not" Exp → Exp > Exp "and" Exp → Exp
variables
"*e*" [*0-9'*]∗ → Exp

Appendix C

The Grammar of AsFix

```
lexical syntax
  chars:Text                 -> Layout        {cons(w)}
  chars:Text                 -> Literal       {cons(l)}
  chars:Text                 -> QLiteral      {cons(ql)}
  chars:Text                 -> Id            {cons(id)}
  chars:Text                 -> Sort          {cons(sort)}
  chars:Text                 -> Separator     {cons(sep)}
  chars:Text                 -> CharClass     {cons(char-class)}
  Literal                    -> AsFixTerm
  name:Text sym:Symbol       -> AsFixTerm     {cons(var)}
  name:Text sym:Symbol       -> AsFixTerm     {cons(lex)}
  chars:Text                 -> Production    {cons(aprod)}
  pairs:ATerm*               -> Abbreviations {cons(abbreviations)}

context-free syntax
  modules:Module* terms:Term*             -> AsFix    {cons(asfix)}
  "module" module-name:Id
         sections:Section* equations:Equations
         abbrevs: Abbreviations           -> Module   {cons(module)}

  "imports" ids:Id+                       -> Section  {cons(imports)}
  "exports" grammars:Grammar+             -> Section  {cons(exports)}
  "hiddens" grammars:Grammar+             -> Section  {cons(hiddens)}

  "sorts" sorts:Sort+                     -> Grammar  {cons(sorts)}
  "lexical syntax"
    productions:Production+               -> Grammar  {cons(lexical-syntax)}
  "context-free syntax"
    productions:Production+               -> Grammar  {cons(context-free-syntax)}
  "priorities" priorities:{Priority ","}+ -> Grammar  {cons(priorities)}
  "variables" productions:Production+     -> Grammar  {cons(variables)}

  module-name:Id syms:Symbol* "->" sym:Symbol
   attrs:Attributes                       -> Production {cons(prod)}

                                          -> Attributes {cons(no-attrs)}
  "{" attrs:{Attribute ","}* "}"          -> Attributes {cons(attrs)}

  Literal                                 -> Attribute
  fun:Literal
    "(" attrs: {Attribute ","}+ ")"       -> Attribute  {cons(attr)}

  Literal                                 -> Symbol
  QLiteral                                -> Symbol
```

```
Sort                                   -> Symbol
CharClass                              -> Symbol
"{" sym:Symbol sep:Symbol "}"
   iter:Literal                        -> Symbol     {cons(iter-sep)}
sym:Symbol iter:Literal                -> Symbol     {cons(iter)}
"~" char-class:CharClass               -> Symbol     {cons(neg)}
label:Id ":" sym:Symbol                -> Symbol     {cons(label)}

"left"                                 -> Assoc
"right"                                -> Assoc
"non-assoc"                            -> Assoc
"assoc"                                -> Assoc


syms:Symbol+ prod:Production           -> ProdSkel   {cons(prod-skel)}
ProdSkel                               -> Group
"{" prods : {ProdSkel ","}* "}"        -> Group      {cons(group)}
"{"assoc:Assoc ":"
  prods:{ProdSkel ","}* "}"            -> Group      {cons(agroup)}

Group                                  -> Priority
groups:{ Group ">" }+                  -> Priority   {cons(decr-chain)}
groups:{ Group "<" }+                  -> Priority   {cons(incr-chain)}
lhs:Group assoc:Assoc rhs:Group        -> Priority   {cons(prio-assoc)}

                                       -> Equations  {cons(no-equations)}
"equations"
  cond-equations:CondEquation*         -> Equations  {cons(equations)}

lhs:AsFixTerm cop:Literal rhs:AsFixTerm -> Condition  {cons(condition)}

module-name:Id tag:Literal
  lhs:AsFixTerm "=" rhs:AsFixTerm      -> CondEquation {cons(ceq-equ)}
module-name:Id tag:Literal
  lhs:AsFixTerm "=" rhs:AsFixTerm
  "when" conditions:{Condition ","}*   -> CondEquation {cons(ceq-when)}
module-name:Id tag:Literal
  conditions:{Condition ","}* impl:Literal
  lhs:AsFixTerm "=" rhs:AsFixTerm      -> CondEquation {cons(ceq-impl)}

prod:Production args:ExtAsFixTerm*     -> AsFixTerm   {cons(appl)}
sym:Symbol elems:ExtAsFixTerm*         -> AsFixTerm   {cons(list)}

AsFixTerm                              -> ExtAsFixTerm
Layout                                 -> ExtAsFixTerm
Separator                              -> ExtAsFixTerm

"term" name:Literal module-name:Id
  term:AsFixTerm  abbrevs:Abbreviations -> Term        {cons(term)}
```

Appendix D

The ToolBus Interface of a Text Editor

This is a list of all the procedures that should at least be implemented in a text editor that is to be used in cooperation with the structure editor. For instructions on how to implement these functions in your language of choice, refer to [Kli96]

- `TB_load-file`

  Function arguments: $string_1$

  Return value: $string_2$

  Description: Opens a new window containing a file that has $string_1$ as its full pathname. Pathnames are in UNIX form, that is, directory names will be delimited by `/`. This function returns $string_2$ containing a window identifier. A window identifier can be an arbitrary string, as long as each window has a unique window identifier.

- `TB_load-string`

  Function arguments: $string_1$

  Return value: $string_2$

  Description: Opens a new window containing $string_1$. Returns $string_2$ containing a window identifier.

- `TB_make-global-menu`

  Function arguments: *string list*

  Return value: *none*

  Description: Creates a menu that is visible in all windows. The menu's name will be *string*. *list* is a list of tuples $(string_1, string_2)$, where $string_1$ is the name for a menu item, and $string_2$ is the associated command string. When a user selects menu item $string_1$, the associated $string_2$ should be sent to the ToolBus using the `TBsend` function. The ToolBus then takes appropriate action, to make sure the command is completed. Examples of menu commands would be: "Parse buffer", "Parse focus", "Move focus up", "Move focus down", and so on. Of course, menu should not be taken literally in this context. An editor that lacks a menu building system (such as emacs) can implement the same functionality using keyboard shortcuts or command names.

- `TB_make-local-menu`

  Function arguments: $string_1$ $string_2$ *list*

  Return value: *none*

  Description: Is equivalent to `TB_make-global-menu`. $string_2$ and *list* correspond with *string* and *list* in `TB_make-global-menu`. $string_1$ is a window identifier for the window which needs to display this particular menu.

- `TB_kill-global-menu`

  Function arguments: *string*

  Return value: *none*

  Description: Removes the menu named *string* from all windows.

- `TB_kill-local-menu`

  Function arguments: $string_1$ $string_2$

  Return value: *none*

  Description: Removes the menu named $string_2$ from the window named $string_1$.

- `TB_make-global-button`

  Function arguments: $string_1$ $string_2$

  Return value: *none*

  Description: Creates a button with name $string_1$ and command string $string_2$ in every window in the editor. All windows that are created after this procedure is called will have this button too.

- `TB_kill-global-button`

  Function arguments: *string*

  Return value: *none*

  Description: Kills the button with name *string* in every window of the editor.

- `TB_make-local-button`

  Function arguments: $string_1$ $string_2$ $string_3$

  Return value: *none*

Description: Creates a button in window with identifier $string_1$. The name of this button is $string_2$. The associated command string is $string_3$. This command string is sent to the ToolBus with `TBsend` if the user clicks the named button.

- `TB_kill-local-button`

  Function arguments: $string_1$ $string_2$

  Return value: *none*

  Description: Kills the button named $string_2$ in window with identifier $string_1$.

- `TB_set-status`

  Function arguments: $string_1$ $string_2$

  Return value: *none*

  Description: Sets the status field of the window with window identifier $string_1$ to $string_2$, or otherwise reveals this status to the user.

- `TB_set-sort`

  Function arguments: $string_1$ $string_2$

  Return value: *none*

  Description: Sets the sort field of the window with window identifier $string_1$ to $string_2$, or otherwise reveals this sort to the user.

- `TB_set-cursor`

  Function arguments: *string integer*

  Return value: *none*

  Description: Set the cursor of the window with window identifier *string* to character position *integer*. Characters start counting from 0.

- `TB_highlight`

  Function arguments: *string* $integer_1$ $integer_2$

  Return value: *none*

  Description: Highlight (or select) a character range in the window with window identifier *string*. The range to be selected starts at character number $integer_1$ and ends with (and includes) character $integer_2$. Characters start counting from 0.

- `TB_register-lang`

  Function arguments: *string*

  Return value: *none*

  Description: Registers a language named *string* so that it can be selected by the user. Typically , this would be in a "Language" menu, where the user can select which language he wants to use. The text editor could also select a language itself from the list of possible languages if it is clear what language is going to be used, for example from the file-name extension.

- `TB_enable-parse`

  Function arguments: *string*

  Return value: *none*

  Description: Enables parsing events to be generated from the text editing window with identifier *string*. As long as this procedure is not called, the window is not allowed to generate parsing events.

- `TB_disable-parse`

  Function arguments: *string*

  Return value: *none*

  Description: Disables parsing events from the window with identifier *string*. No parse requests can be send after this procedure is called. Only after a `TB_enable-parse` can parse requests be sent again.

- `TB_get-string`

  Function arguments: $string_1$ $integer_1$ $integer_2$

  Return value: $string_2$

  Description: Retrieve a string from the window with window identifier $string_2$. The range of characters to be retrieved starts at character number $integer_1$ and is $integer_2$ characters long. Characters start counting from 0. The retrieved string is sent back to the ToolBus as $string_2$.

- `rec-ack-event`

  Function arguments: *integer*

  Return value: *none*

  Description: Required by ToolBus. Should be an empty (skip, no-op) procedure. *integer* argument can be ignored.

- `rec-terminate`

  Function arguments: *integer*

  Return value: *none*

  Description: Terminates the editor. *integer* argument can be ignored.

Appendix E

List of Variables

Variable name	Sort name	Defined in module
α	Area	AreaSyntax$^{2.1.6}$
ϵ	Editor	Editor$^{2.1.7}$
fl	{Focus ";"}*	FocusSyntax$^{2.1.2}$
i	Int	Integers
lit	Literal	AsFix-Utilities$^{A.2}$
m	Move	FocusMove$^{2.2.9}$
n	Nat	Integers
π	Path	Paths$^{2.1.4}$
σ	Sort	Sorts$^{2.1.5}$
t	ATerm	AsFix-Utilities$^{A.2}$
$t+$	ATerms	AsFix-Utilities$^{A.2}$
tl	ATermList	AsFix-Utilities$^{A.2}$
ϕ	Focus	FocusSyntax$^{2.1.2}$
ϕ^*	FocusList	FocusSyntax$^{2.1.2}$

Bibliography

[AHW90] F. Arefi, C.E. Hughs, and D.A. Workman. Automatically generating visual syntax-directed editors. *Communications of the ACM*, 33(3):349–360, March 1990.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.

[BB96] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. *Formal aspects of computing*, 8(2):188–208, 1996.

[BK96a] J. A. Bergstra and P. Klint. The discrete time ToolBus. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of LNCS, pages 288–305. Springer-Verlag, 1996.

[BK96b] J.A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*, pages 75–88. Springer-Verlag, 1996.

[BKOV] M.G.J. van den Brand, P. Klint, P.A. Olivier, and E. Visser. Syntax trees for distributed environments. Technical Report, University of Amsterdam. In preparation.

[BKV96] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[Bra92] M.G.J. van den Brand. *Pregmatic: A generator for incremental programming environments*. PhD thesis, Catholic University Nijmegen, The Netherlands, 1992.

[Bro91] K.P. Brooks. Lilac: A two-view document editor. *Computer*, 24(6):7–19, June 1991.

[BV96] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, January 1996.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[Hil96] J. Hillebrand. *Experiments in specification re-engineering.* PhD thesis, University of Amsterdam, 1996.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Kli96] P. Klint. *A Guide to ToolBus programming.* University of Amsterdam and Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 0.9 edition, 1996.

[KLMM93] J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, and B. Magnusson, editors. *Object-Oriented Environments: The Mjølner Approach.* Prentice Hall, 1993.

[Koo94] J.W.C. Koorn. *Generating Uniform interfaces fo interactive programming environments.* PhD thesis, Institute for Logic, Language, and Computation, University of Amsterdam, 1994.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, NJ., 1988.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as KW93.ps.Z.

[KW95] J. F. Th. Kamperman and H. R. Walters. The new ASF compiler — An exercise in self-applicability. In M. G. J. van den Brand, A. van Deursen, T. B. Dinesh, J. Kamperman, and E. Visser, editors, *Proceedings of ASF+SDF'95: a workshop on Generating Tools from Algebraic Specifications*, pages 271–293. Technical Report P9504, Programming Research Group, University of Amsterdam, May 11 & 12 1995.

[Moo96] L.M.F. Moonen. Data flow analysis for reverse engineering. Master's thesis, University of Amsterdam, Programming Research Group, 1996. Also appeared as Technical Report P9613.

[Ous94] J.K. Ousterhout. *Tcl and the Tk Toolkit.* Professional Computing Series. Addison-Wesley, Reading, MA., 1994.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992.

[RT89] T.W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language Based Editors.* Springer-Verlag, 1989.

[Vis95] E. Visser. ASF+SDF to LaTeX, 1995. Web site with links to the software and manuals. URL: http://adam.wins.uva.nl/~visser/tolatex/.

[Vis97] E. Visser. A family of parser generators. Technical Report, Programming Research Group, University of Amsterdam. In preparation, 1997.