

# Interaction with a (proof) editor

## What, but not how

Bengt Nordström

Computing Science, Chalmers and University of Göteborg

JAIST-AIST Workshop on Verification Technology, Senri,  
May 2006

## Key word: flexibility for the user:

- Flexible granularity of type checking (from batch to interactive)
- Text and structure editing
- Flexible verbosity of the interface.
- Manual or automatic layout

## Time granularity:

When should the buffer be checked?

The user should be able to set the granularity of checking, e.g.

**superfine:** After each change of the buffer.

**fine:** Only when a non-character is pressed or when the focus is leaving the current identifier.

**medium:** Not as long as the previous modification was done in a neighbouring place.

**coarse:** Only when the buffer has been changed.

An independent question: What part of the structure being edited must be checked? It is clear that rechecking the whole structure is always possible, but maybe it is possible to check only a part?

## Space granularity:

What is the least checkable unit?

**Fine** Each substructure.

**Medium** A declaration of a constant, including its entire definition.

**Coarse** The whole buffer is checked.

## Two perspectives

The editor is text oriented in the sense that the user can use the mouse to move freely in the text, it is possible to do textual editing without any special command. The user should have a feeling that she is editing a text, but that the computer is doing typechecking and helps to fill in the object. An object can be built by typing its concrete syntax, one character after the other.

It is also possible to edit a structure in a structural way, i.e. if we have a correct partial object, we can fill in the incomplete parts by using commands based on the abstract syntax (or by typing the parts using the concrete syntax).

# Editing text

This should work like in a normal text-editor. The region is used to mark an editable part of the text, i.e. where a character can be inserted or where a bigger piece of text can be put (for instance from the clipboard or from a file). The mouse and the arrow-keys are used in their normal way to move within the text.

# Structural editing

- If we have a well-formed partial structure (i.e. a buffer which is type-correct and possibly have placeholders for not yet filled-in parts), then it should be possible to use structural commands to edit the structure.
- The invariant should be that a structural command always take a well-formed structure to a well-formed structure.
- In the case of errors, the erroneous parts will be saved as texts in the placeholders.

# What is the focus of editing?

## placeholder

- Everywhere where a part can be inserted.
- Placeholders for lists (like in a list of characters, or a list of arguments) are not visible until you put the cursor on a position in the text where a list element can be inserted.
- Structural placeholders (i.e. in a correct context) are marked.

## region

- Marks a subpart of the structure
- Can be correct or not (and so marked).

The region is used to mark a subpart of the structure we are editing. A typecorrect (structural) region has always a certain color. The region can be created by double-clicking on a part of the text. This will mark the smallest type-correct structure containing point.

# Operations on the region

- Navigation
- Deletion and copying
- textual editing

# Navigation

We can use the S-arrow keys to move and resize a typecorrect region. The operation will always take a correct region to a correct region.

- S-up extend the region to contain the smallest enclosing structure.
- S-down move the region to its first substructure.
- S-right, S-left move the region to a sibling.

The region can also be seen textually, so all normal operations to textually change the region is available.

# Operations on the region

- Deletion:** We can use the backspace to replace the typecorrect region with a placeholder. This will save the region in a clipboard.
- Copying:** We can copy the region to a clipboard, which later can be pasted to a placeholder.
- Building:** Either by inserting characters from the keyboard or by issuing various commands. The effect is to first replace the region by a placeholder and then building.

# Generalized completion

- The user can use the tab-key to ask the system for completions.
- At the end of an initial part of an identifier this is normal completion.
- In other contexts new placeholders will be inserted when they are known to exist.

# Structural editing: expressions

It is the structure (= abstract syntax) of expressions which define the operations to build expressions. (not obvious)

## Concrete syntax

```
e ::= \i.e | i e1 ... en
```

## Abstract syntax

```
Exp ::= lambda Id Exp | apply Id [Exp]
```

# Building an application

[.] : A

Then the screen will gradually change:

input	screen
	[ ]
a	[a ]
b	[ab ]
c	[abc ]
d	[abcd ]
blank	

The response after the insertion of an identifier:

- If the type of the identifier is A, then the identifier changes colour to show that it is a correct expression.
- If the user hits the tab, and if the identifier has a type which ends with A, then the screen will look like:

```
(abcd [.] [] ... [])
```

The correct number of placeholders are inserted.

# Abstraction

The user can input the concrete syntax as text, or using tab to get some feedback from the system:

input	screen
	[ ]
$\lambda$	[ $\lambda$   ]
tab	$\lambda$ [ ] . [ ]
a	$\lambda$ [a   ] . [ ]
b	$\lambda$ [ab   ] . [ ]
c	$\lambda$ [abc   ] . [ ]
d	$\lambda$ [abcd   ] . [ ]
.	$\lambda$ abcd . [ ]

# Declarations

For simplicity we assume that we can declare explicitly defined constants with a syntax like:

```
def f (x1:A1)...(xn:An) : B = e
```

This means that the following structure is edited:

```
Program = (def Decl)*  
Decl = Ident VarDecls ':' TypeExp '=' Exp  
VarDecls = VarDecl*  
VarDecl = '(' Ident ':' TypeExp ')'  
TypeExp = Exp  
TypeExp = VarDecl TypeExp
```

When we want to build this kind of structure, we focus on a place where a declaration can be inserted and press the tab-key:

```
def [.Ident] [VarDecls] : [TypeExp] = [[Exp]]
```

Each placeholder gives information about what kind of thing which it stands for. The last placeholder has an indication that it cannot be filled in yet (assuming that we want to enforce that).

We can now type an identifier (for instance abcd) and end it by a blank or an equality sign, then the screen will change to

```
def abcd [.VarDecls] : [TypeExp] = [[Exp]]
```

We can now either start to type a list of variable declarations or hit the tab:

```
def abcd ([.Ident] : [TypeExp]) ... : [TypeExp] = [[Exp]]
```

The three dots indicating that this is a list which may be extended.

# Verbose mode

- This is a mode where each subexpression is displayed with its type. It looks like a tilted natural deduction tree and is more readable as a proof.
- It is the type of the subexpressions which tells what subgoals have been reached during the process of building the proof.
- To read the term as a proof without having the type of the subexpressions is exactly like only reading the names of the rules building up the proof. This is not readable.

```
{AxB -> BxA} >  
[.]
```

```
next_input: \x. 
```

```
{AxB -> BxA} >
```

```
\x : AxB
```

```
{BxA} >
```

```
[.]
```

```
{AxB -> BxA} >  
\x : AxB  
  {BxA} >  
  split  
    {AxB} >  
    [.]  
    {(u:A) (z:B) BxA} >  
    []
```

```
{AxB -> BxA} >  
\x : AxB  
  {BxA} >  
  split  
    {AxB} >  
    [x]  
    {(u:A) (z:B) BxA} >  
    [.]
```

```
{AxB -> BxA} >
\x : AxB
  {BxA} >
  split
    {AxB} >
    [x]
    {(u:A) (z:B) BxA} >
    \u : A
      {(z:B) BxA} >
      [.]
```

```
{AxB -> BxA} >
\x : AxB
  {BxA} >
  split
    {AxB} >
    [x]
    {(u:A) (z:B) BxA} >
    \u : A
      {(z:B) BxA} >
      \z : B
        {BxA} >
        [.]
```

```
{AxB -> BxA} >
\x : AxB
  {BxA} >
  split
    {AxB} >
    [x]
    {(u:A) (z:B) BxA} >
    \u : A
      {(z:B) BxA} >
      \z : B
        {BxA} >
        pair
          {B} >
          [.]
          {A} >
          []
```



```
{AxB -> BxA} >
\x : AxB
  {BxA} >
  split
    {AxB} >
    [x]
    {(u:A) (z:B) BxA} >
    \u : A
      {(z:B) BxA} >
      \z : B
        {BxA} >
        pair
          {B} >
          [z]
          {A} >
          u
```

# Summary

**Unlimited undo** This is extremely important

**Structure and text editing** The user sees the editor as a text editor with capabilities for structure editing.

**Generalized Completion** Asks the computer: Which kind of thing can be inserted here?

**Verbose mode** Important for reading the program as a proof.

**Hyperlinked identifiers** Identifiers are hyperlinked to their declaration.

**Flexible granularity of checking** The user can set the level of interaction with the checker.

**Colour syntax code** Colours are used to mark what part of the text is type-correct (i.e. the parts which can be used for structured editing).

# References

- IDE Integrated development environment — [www.eclipse.org/](http://www.eclipse.org/)
- Amaya — <http://www.w3.org/Amaya/>
- FrameMaker — <http://www.adobe.com/products/frame maker/main.html>
- MathType — <http://www.math type.com/en/products/math type/>.
- Scientific WorkPlace — <http://www.mackichan.com/products/swp.html>
- LyX
- Mathematica — <http://www.wolfram.com/products/mathematica/index.html>
- TEXmacs — <http://www.texmacs.org/>