the Netty Project

Eric Hehner, Robert Will, Lev Naiman, David Kordalewski, Botond Ballo, Anya Tafliovich

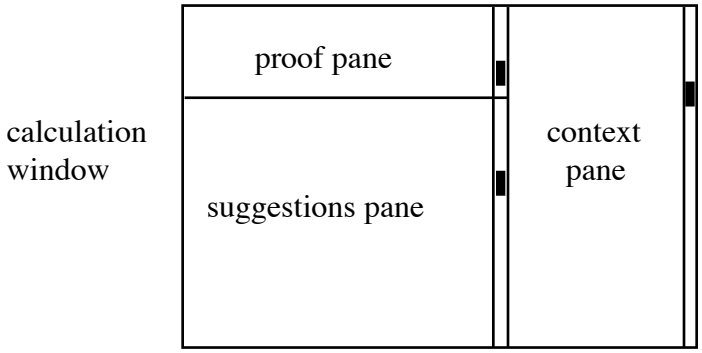
Netty is a proof tool named after Antonetta Johanna Maria van Gasteren 1952-2002, who was a pioneer of calculational proving. It is based on the MSc thesis of Robert Will titled "Constructing Calculations from Consecutive Choices – a Tool to Make Proofs More Transparent", completed at the University of Toronto in 2010 under the supervision of Eric Hehner. Further design and implementation were by Lev Naiman, David Kordalewski, Botond Ballo, and Anya Tafliovich. The syntax and semantics of the mathematical expressions and specifications and programs understood by Netty are those of the book *a Practical Theory of Programming*. This document is an informal, and unfinished, specification. [Note added 2018: the Netty project is currently dormant. Anyone wanting to wake it is welcome to do so. It will have to be renamed because Netty now refers to a library of open source IO software for Java.]

Overview

Netty is a prover's assistant. It is not a prover. It supports a calculational style of proof in which the lines may be of any type. They may be of type binary, or number, or set, or string, or list, or function, or other types including types invented by the user, and the type can change as the user zooms in and out of subexpressions. The type can be specification and program, supporting refinement from specification to program, or verification that all executions of a program satisfy a specification.

Netty keeps track of the proof direction, and enforces a consistent direction. It keeps track of local context. It makes suggestions for the next step of the proof, not by saying what laws are applicable, but by saying what are the results of those law applications. It allows its users to experiment with different lists of laws.

Menu item **New** creates a new calculation window, which consists of three panes, all initially empty, arranged as follows.



As many calculation windows as desired can be open at the same time. The division lines between the panes can be dragged. Each pane is separately scrollable. The proof pane will show the current state of the proof. The context pane will show the laws due to the current proof context. And the suggestions pane will show suggestions for the next step of the proof.

Proof

Here is an example short proof as it would appear in the proof pane.

⇐	$a \Rightarrow (b \Rightarrow a)$	portation
=	$a \wedge b \Rightarrow a$	specialization
=	$\top$	

Without the boxed symbol and the vertical line hanging from it, we would have an ordinary calculational proof. The boxed symbol and line serve a structural purpose. The vertical line hanging from the box shows the extent of the proof. It is unnecessary for the outermost proof, but it will be helpful for subproofs. In the box we have a direction. For any data type, there are three directions. For numbers, they are $\leq$ $=$ $\geq$. For sets, they are $\subseteq$ $=$ $\supseteq$. For binary values, they are $\Rightarrow$ $=$ $\Leftarrow$. For bunches, they are $:$ $=$ $::$. For strings and lists they are the direction appropriate to the items in the strings and lists. For functions they are the directions appropriate to the result (body, range) of the function.

- If the direction is $=$, then all connectives in the proof (at this level) must be $=$.
- If the direction is $\leq$, then the connectives in the proof (at this level) can be $=$ $\leq$ $<$.
- If the direction is $\geq$, then the connectives in the proof (at this level) can be $=$ $\geq$ $>$.

And similarly for other data types.

The direction symbol limits what connectives can be used, and therefore what laws are applicable, but does not say what the proof proves.

- If the connectives are all $=$, the proof proves topline=bottomline.
- If the connectives include at least one $\leq$ but no $<$, then the proof proves topline≤bottomline.
- If the connectives include at least one $<$, then the proof proves topline<bottomline.
- If the connectives include at least one $\geq$ but no $>$, then the proof proves topline≥bottomline.
- If the connectives include at least one $>$, then the proof proves topline>bottomline.

And similarly for other data types. In the special case that the proof proves $a=\top$ or proves $a \Leftarrow \top$, we say that the proof proves a. So this example proof proves $a \Rightarrow (b \Rightarrow a)$. In the special case that the proof proves $a=\bot$ or proves $a \Rightarrow \bot$, we say that the proof proves $\neg a$.

In the phrase “line of a proof”, the word “line” is misleading. Depending on the length of a formula and the width of the proof pane, a formula may require several lines. When that is the case, the formula will be formatted nicely. But we still say “line of a proof” for the formula.

Following the first formula we see the word “portation”. That says that a law named “portation” was used to get from the first line to the middle line. Similarly a law named “specialization” was used to get from the middle line to the last line. Laws are not required to have names; any law without a name is labelled “a X law” where X is the name of a law list. Law names are not necessarily unique; all the laws in the context pane are named “context”.

Law lists are plain text files that can be created, viewed, and deleted in the usual way that plain text files are created, viewed, and deleted, and they can be modified using any text editor. A law in a law file consists of an optional name and a binary expression. Standard law files include binary, generic, number, bunch, set, string, list, function, quantifier, limit, specification, and program laws. Anyone may experiment by adding, deleting, or modifying law lists.

Using menu item **Add to Law File**, whatever is proven by the proof in the proof pane can be added to a selected law file and optionally given a name. If the proof contains any logical gaps or warnings (see later), you will be warned against adding it to a law file, but you can still do so.

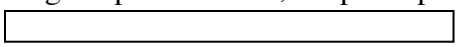
Using menu item **Save**, a proof in a proof pane can be saved as a file in a format that can be written and read only by the Netty tool. Using menu items,
- the proof in the proof pane can be printed.
- the proof in the proof pane can be saved to a new proof file.
- the proof in the proof pane can be saved to an existing proof file, replacing what was there.
- the proof pane can be loaded from a proof file, and the context and suggestions will be recalculated.
- a proof file can be deleted.

These are the only ways proof files can be created, used, and deleted.

Getting Started

Clicking on the **Laws** menu item reveals the names of the lists of laws that can be used; initially it is empty. Under the **Laws** menu item there is also an **Add** menu item that allows you to select a file and add it to the list of law lists. Under the **Laws** menu item there is also a **Delete** menu item. If you select the names of one or more law lists, the **Delete** menu item turns from grey to black, and can be used to remove files from the list of law lists. Adding and deleting law lists can be done at any time.

To get a proof started, the proof pane of a new calculation window has a dialog box



for entering (typing in) the proof direction and first line of the proof. This dialog box is called the "focus". When a proof direction and first line are entered into the focus, they become a line of the proof, and a new focus (dialog box) appears below it.

Continuing

There is always at most one focus. The focus can be created or changed by clicking on the connective in the left margin of a line of the proof; the focus is then between the selected line and the following line. The focus can be filled in four ways.
- A suggestion can be selected from the displayed suggestions.
- A main operand of the preceding line can be selected and we zoom in.
- If we have previously zoomed in more times than we have zoomed out, then we can zoom out.
- We can simply enter (key in) the next line we want directly.

We discuss these in turn.

Suggestions

The laws in the law lists and in the context pane are used to create the suggestions in the suggestions pane. The order in which suggestions appear will initially be SOMETHING, after which it will be adjusted automatically by a NOT YET DETERMINED machine learning technique according to patterns of use. To take one of the suggestions from the suggestions pane, just click on it. The suggestion becomes the next line of the proof, and a new focus appears below it.

There are four ways that a law might create a suggestion.
- If the line before the focus is $\top$, then the law is a suggestion, with $=$ in the left margin.
- If the line before the focus is an instance of the law, then $\top$ is a suggestion, with $=$ in the left margin.

- If the law has a main operator that is allowed by the current proof direction, and the line before the focus is an instance of the left side of the law, then the right side of the law, instantiated the same way as the left side, is a suggestion, with the law's main operator in the left margin.
- If the law has a main operator whose reverse is allowed by the current proof direction, and the line before the focus is an instance of the right side of the law, then the left side of the law, instantiated the same way as the right side, is a suggestion, with the reverse of the law's main operator in the left margin.

Sometimes the line before the focus is an instance of one side of a law, but some variables in the other side of the law remain unconstrained. For example, if the line before the focus is 0, the right side of the law $x–x = 0$ matches, but leaves x unconstrained. For another example, if the line before the focus is $\exists Q$, the right side of the law $P\,x \Rightarrow \exists P$ matches, but leaves x unconstrained. In that way, applying a law can introduce new variables.

The law $a \wedge b \Rightarrow a$ mentioned previously is actually

$$\forall a, b: bin\cdot\ a \wedge b \Rightarrow a$$

and its application requires that the expressions unifying with a and b be binary. The law $P\,x \Rightarrow \exists P$ mentioned previously is actually

$$\forall P: A \rightarrow bin\cdot\ \forall x: A\cdot\ P\,x \Rightarrow \exists P$$

Its application requires that the expression unifying with P be a function with range bin, and that the expression unifying with x be in the domain of the function unifying with P. If the type checker is unable to determine that the unifying expressions have the correct types, a warning is placed on the line to which the law is being applied. The warning can be removed by supplying the proof that the expressions have the right types; see the section titled **Gaps**.

Zoom

If the line has a main operator with one or more operands, click on one of these main operands to zoom in to it. Zooming is one level at a time; to zoom in to something smaller, zoom in several times. If the main operator is associative, for example, $a+b+c$, then any one of the operands can be selected. (See **Associativity**, later.)

An operand selected for zooming in is in a positive, neutral, or negative position. Here are some examples.

¬ negative
– negative
positive + positive
positive – negative
positive ↑ positive
positive ↓ positive
negative ≤ positive
negative < positive
positive ∧ positive
positive ∨ positive
negative ⇒ positive
neutral = neutral
neutral ≠ neutral
negative : positive

positive , positive
positive ‘ positive
positive ∪ positive
positive ∩ positive
positive ; positive
[positive]
positive ;; positive
if neutral **then** positive **else** positive **fi**
⟨*v*: negative → positive ⟩

Zooming in creates the first line of a subproof. Its direction is created from the direction of the previous line and the selected operand position as follows.
• positive position and any old direction makes the same new direction
• negative position and old direction ≤ makes new direction ≥
• negative position and old direction ≥ makes new direction ≤
• any position and old direction = makes new direction =
• neutral position and any old direction makes new direction =
In these five rules, the old direction is written as ≤ or ≥ or = , which is appropriate if the line we are zooming in from is of number type (or string of numbers, or list of numbers, or function with number range); if it is of another type, the old direction symbols are those appropriate for the type of the subexpression we are zooming in to, which may not be the same as the type of the expression we are zooming in from. For example, if the expression we zoom in from is binary, and the expression we zoom in to is a set, the first rule becomes the three rules
• positive position and old direction ⇒ makes new direction ⊆
• positive position and old direction ⇐ makes new direction ⊇
• positive position and old direction = makes new direction =

What the subproof proves is not determined by the subproof direction. It is determined solely by the connectives appearing in the subproof in exactly the same way as a main proof.

Zooming out is caused by the **Zoom Out** menu item or by pressing the up-arrow key. Here are some scenarios in which a , b , and c are of number type. Note the corner brackets.

scenario 0

≤	$a+\lfloor b \rfloor$
≤	b
<	c
<	$a+\lceil c \rceil$

scenario 1

≤	$a+\lfloor b \rfloor$
≤	b
≤	c
≤	$a+\lceil c \rceil$

scenario 2

≤	$a+\lfloor b \rfloor$
≤	b
=	c
=	$a+\lceil c \rceil$

scenario 3

≤	$a-\lfloor b \rfloor$
≥	b
>	c
<	$a-\lceil c \rceil$

scenario 4

⇐	$a\geq\lfloor b \rfloor$
≤	b
≤	c
⇐	$a\geq\lceil c \rceil$

scenario 5

⇐	$a>\lfloor b \rfloor$
≤	b
≤	c
⇐	$a>\lceil c \rceil$

scenario 6

⇐	$a\geq\lfloor b \rfloor$
≤	b
<	c
⇐	$a\geq\lceil c \rceil$

scenario 7

⇐	$a>\lfloor b \rfloor$
≤	b
<	c
⇐	$a\geq\lceil c \rceil$

scenario 8

$$\boxed{\Leftarrow}\quad a=\lfloor b\rfloor$$
$$\boxed{=}\quad b$$
$$=\quad c$$
$$=\quad a=\lceil c\rceil$$

The leading connective of the new zoom-out line is determined as follows:
- If the zoom position is neutral, the new leading connective is $=$.
- If the subproof proves equality, the new leading connective is $=$.
- Otherwise the new leading connective is the direction of the line we zoomed in from.

These rules are not optimal; they do not do justice to scenarios 0 and 3.

The new zoom-out line is the same as the old zoom-in line except that the subexpression we zoomed in to, which became the top line of the subproof, is replaced by the bottom line of the subproof. This is not optimal; it does not do justice to scenario 7.

Zooming out immediately after zooming in is the same as undo (see later): the zoom-in line is deleted. Whenever two zoom-ins in a row are matched by the corresponding two zoom-outs in a row, they are combined to a single zoom-in and a single zoom-out. The corner brackets indicate the innermost zoom. For example, the proof on the left below becomes the proof in the middle. Furthermore, if the subproof consists of only a single law application, as in this example, the subproof disappears, and the law name moves up, giving us the proof on the right.

$$\boxed{=}\quad \lfloor a+b\rfloor - c \quad \text{zoom in}$$
$$\boxed{=}\quad a+\lfloor b\rfloor \quad \text{zoom in}$$
$$\boxed{=}\quad b \quad \text{identity}$$
$$=\quad b\times 1 \quad \text{zoom out}$$
$$=\quad a+\lceil b\times 1\rceil \quad \text{zoom out}$$
$$=\quad \lceil a+b\times 1\rceil - c$$

$$\boxed{=}\quad a+\lfloor b\rfloor - c \quad \text{zoom in}$$
$$\boxed{=}\quad b \quad \text{identity}$$
$$=\quad b\times 1 \quad \text{zoom out}$$
$$=\quad a+\lceil b\times 1\rceil - c$$

$$\boxed{=}\quad a+\lfloor b\rfloor - c \quad \text{identity}$$
$$=\quad a+\lceil b\times 1\rceil - c$$

Zooming in adds laws to the context, and zooming out removes them.

- From $a\wedge b$, if we zoom in on a, we gain context b.
- From $a\wedge b$, if we zoom in on b, we gain context a.
- From $a\vee b$, if we zoom in on a, we gain context $\neg b$.
- From $a\vee b$, if we zoom in on b, we gain context $\neg a$.
- From $a\Rightarrow b$, if we zoom in on a, we gain context $\neg b$.
- From $a\Rightarrow b$, if we zoom in on b, we gain context a.
- From $a\Leftarrow b$, if we zoom in on a, we gain context b.
- From $a\Leftarrow b$, if we zoom in on b, we gain context $\neg a$.
- From **if** a **then** b **else** c **fi**, if we zoom in on a, we gain context $b\neq c$.
- From **if** a **then** b **else** c **fi**, if we zoom in on b, we gain context a.
- From **if** a **then** b **else** c **fi**, if we zoom in on c, we gain context $\neg a$.

If the context added is a conjunction, we gain each of the conjuncts as a separate law. If the context added is a negation, the negation is pushed inwards. In detail,

$a \wedge b$	becomes	a and b
$\neg\neg a$	becomes	a
$\neg(a \wedge b)$	becomes	$\neg a \vee \neg b$
$\neg(a \vee b)$	becomes	$\neg a$ and $\neg b$
$\neg(a = b)$	becomes	$a \neq b$
$\neg(a \neq b)$	becomes	$a = b$
$\neg(a \Rightarrow b)$	becomes	a and $\neg b$
$\neg(a \Leftarrow b)$	becomes	$\neg a$ and b

For example: from $a \wedge b \Rightarrow c$, if we zoom in on c, we gain both a and b; from $a \vee \neg b$, if we zoom in on a, we gain b; from $a \vee b \neq c$, if we zoom in on a, we gain $b = c$.

Example

$$
\begin{array}{ll}
\boxed{\Leftarrow} & \llcorner(\neg a \Rightarrow \neg b) \wedge a \neq b\lrcorner \vee (a \wedge c \Rightarrow b \wedge c) \quad \text{zoom in} \\
\boxed{\Leftarrow} & (\neg a \Rightarrow \neg b) \wedge \llcorner a \neq b\lrcorner \quad \text{exclusion} \\
= & (\llcorner \neg a \Rightarrow \neg b\lrcorner) \wedge \ulcorner a = \neg b\urcorner \quad \text{zoom in} \\
\boxed{\Leftarrow} & \neg a \Rightarrow \llcorner \neg b\lrcorner \quad \text{context} \\
= & \neg a \Rightarrow \ulcorner a\urcorner \quad \text{indirect proof} \\
= & a \quad \text{zoom out} \\
= & \ulcorner a\urcorner \wedge \llcorner a = \neg b\lrcorner \quad \text{zoom in} \\
\boxed{\Leftarrow} & \llcorner a\lrcorner = \neg b \quad \text{context} \\
= & \ulcorner \top\urcorner = \neg b \quad \text{identity} \\
= & \neg b \quad \text{zoom out} \\
= & a \wedge \ulcorner \neg b\urcorner \quad \text{duality} \\
= & \neg(\llcorner \neg a \vee \neg\neg b\lrcorner) \quad \text{zoom in} \\
\boxed{\Rightarrow} & \neg a \vee \llcorner \neg\neg b\lrcorner \quad \text{double negation} \\
= & \neg a \vee \ulcorner b\urcorner \quad \text{inclusion} \\
= & a \Rightarrow b \quad \text{zoom out} \\
= & \neg(\ulcorner a \Rightarrow b\urcorner) \quad \text{zoom out} \\
= & \ulcorner \neg(a \Rightarrow b)\urcorner \vee (\llcorner a \wedge c \Rightarrow b \wedge c\lrcorner) \quad \text{zoom in} \\
\boxed{\Leftarrow} & \llcorner a\lrcorner \wedge c \Rightarrow b \wedge c \quad \text{context} \\
\Leftarrow & \ulcorner b\urcorner \wedge c \Rightarrow b \wedge c \quad \text{reflexive} \\
= & \top \quad \text{zoom out} \\
\Leftarrow & \neg(a \Rightarrow b) \vee \ulcorner \top\urcorner \quad \text{symmetry} \\
= & \top \vee \neg(a \Rightarrow b) \quad \text{base} \\
= & \top
\end{array}
$$

Associativity

If the main operator of the line before the focus is associative, for example, $a+b+c$, then clicking on any one of the operands zooms in on that operand. There is no need for associative laws for the following associative operators.

$\wedge$ $\vee$ $+$ $\times$ $\uparrow$ $\downarrow$, ‘ $\cup$ $\cap$; ;; | . ||

To parenthesize some of the operands in an association of operators, shift-click-sweep through the operands to be parenthesized. If the gesture is applied to all the operands in an unnecessarily parenthesized expression, the parentheses disappear.

The operators $=$ and $\neq$ with binary operands are associative, but shift-click-sweep does not apply to them. They are continuing operators, like $\leq$ and $<$ and all other ordering operators. For example, $a=b=c$ means $(a=b)\wedge(b=c)$, and $a\leq b=c$ means $(a\leq b)\wedge(b=c)$. Associative laws are needed for binary $=$ and $\neq$. Associative laws are also needed for any new associative operators.

Direct Entry

The focus can be filled by directly typing in your choice of next line. The leading connective must be allowed by the current direction. You might do this with the intention to come back later and fill in the missing proof steps, or apply a law that you will add but haven't yet added to a law list. Direct entry should not be frequent.

Direct entry can also be used as a means of narrowing the choice of suggestions. With each character typed, the list of suggestions decreases to just those that still match.

When the focus is filled entirely by direct entry, it gets a warning. For example, if the current proof is

$\boxed{\Leftarrow}$ $\neg(a\Rightarrow b)\Rightarrow a$ material implication
$=$ $(a\Rightarrow b)\vee a$
$\square$

and we directly enter "$= \quad a \Rightarrow a\vee b$ ", then we see

$\boxed{\Leftarrow}$ $\neg(a\Rightarrow b)\Rightarrow a$ material implication
$=$ $(a\Rightarrow b)\vee a$ warning
$=$ $a \Rightarrow a\vee b$
$\square$

A warning indicates a logical gap in the proof. The gap can be filled, and the warning removed, as described later in the section on **Gaps**.

Law Query

In a proof, if a law is applied, and you click-and-hold on the law name, you will be shown the law that created the next line, and how it applies. Suppose, somewhere in a proof, we have the two lines

$x\wedge y \Rightarrow (y\vee z \Rightarrow (z\Rightarrow x))$ portation
$=$ $x\wedge y\wedge(y\vee z) \Rightarrow (z\Rightarrow x)$

If you click-and-hold on the word "portation", you will see the lines replaced by

$a \Rightarrow (b \Rightarrow c)$ portation
$=$ $a \wedge b \Rightarrow c$

The portation law is actually

$\forall a, b, c: bin\cdot\ (a\wedge b \Rightarrow c) = (a\Rightarrow(b\Rightarrow c))$

but it is displayed with the two sides of $=$ reversed, and its variables replace the subexpressions they unified with, and they are spaced out so that the other symbols (in the top line $\Rightarrow(\Rightarrow)$ and in the bottom line $\wedge\Rightarrow$) do not move. Releasing the click returns the lines as they were. It rarely happens that an

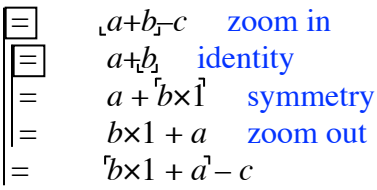
identifier in a law takes more space than the subexpression it unifies with. When it does happen, there is a jumble of overprinting.

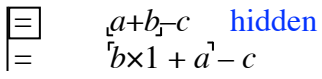
Deleting

A region (consecutive sequence of lines) of a proof can be selected for deletion by a shift-click-sweep action. If the first or last line of a subproof is selected, then the entire subproof is selected. When a region is selected, the delete button causes it to disappear, and causes a warning to appear on the line before the region that disappeared, indicating a logical gap in the proof. (The warning may have been there already, perhaps due to a previous deletion or direct entry.) If the first line of the deleted region was created by a law application, the law name also disappears. If the first line of the deleted region was created by zooming in, the low corner brackets on the line before the deleted region also disappear. If the last line of the deleted region was followed by zooming out, the high corner brackets on the following line also disappear.

Hiding

A region of a proof can be selected for hiding in the same way as for deletion. When a region is selected, SOME ACTION causes it to disappear, and causes a note to appear on the line before the region that disappeared. If the hidden region included a warning indicating a logical gap, or there was already a warning on the previous line, then that note is warning . But if there was no warning just before or within the selected region, then the note is hidden . For example, selecting the subproof (the inner proof) on the left, or any part of it that includes either its top line or its bottom line, and then SOME ACTION, results in the proof on the right.





Clicking on hidden or warning causes it to disappear and causes the hidden region to reappear.

Scope

If the line before the focus is a function $\langle v{:}d{\to}b\rangle$, we can zoom in on the domain d or the body b . To create the new subproof direction, the domain is in a negative position and the body is in a positive position. For the body, we gain the context $v{:}d$ and $\not\in v = 1$.

The new variable, whose visible name is v , is given a fresh new name internally that cannot be written by a user, so that the variable will be distinguished from any already existing variable of the same name. For the purpose of applying a law, the internal name is used, so there is never a problem of “variable capture” or “variable hiding”. If variable v occurs in any already existing context, it is greyed out. The domain d cannot mention v .

Internally, there is a stack of variable names. When a scope is entered, the scope entry symbol $\langle$ is pushed onto the stack. When a variable is declared, a check is made to ensure that the visible name has not been declared since the most recent scope entry, and then it is pushed onto the stack along with its unique internal name. When a name is used, it is searched for from the top downwards, and then

replaced by its internal name. When a scope is exited, the stack is popped down to and including the topmost ⟨ .

If we enter the following as the first line of a proof

⇐ $(\neg a \Rightarrow \neg b)\wedge(a\neq b) \vee (a\wedge c \Rightarrow b\wedge c)$

then a , b , and c are not variables. To create variables, enter

⇐ $\langle a, b, c\colon bin \rightarrow (\neg a \Rightarrow \neg b)\wedge(a\neq b) \vee (a\wedge c \Rightarrow b\wedge c)\rangle$

and then zoom in on the body, gaining the context $a\colon bin$ and $b\colon bin$ and $c\colon bin$.

Function Application

If the line before the focus is a function application $\langle v{:}d\rightarrow b\rangle a$, the suggestion pane shows the result of application. If that suggestion is selected, the suggestion fills the focus, as usual. But there is a gap in the proof; what's missing is to prove

⇐ $¢a = 1 \ \wedge \ a{:}d$

with all the same context as the function application line. If the missing proof cannot be done by the type-checker, warning is placed beside the line before the old focus. To fill in the gap and remove warning , see **Gaps**, below.

Gaps

There are four ways in which a logical gap in the proof can be created. As we saw earlier, one is by direct entry of the next line of the proof. Another is by selecting and deleting a region of the proof. Another is by applying a law for which the type checker cannot determine that the unifying expressions have the right types. And as we have just seen, a gap is created by applying a function to an operand for which the type checker cannot determine that it has the right type. Any logical gap in the proof is marked by warning on the line just before the gap.

To fill in a gap, select (click on) warning . If the warning belongs to a hidden region, the hidden region will reappear with warning in it somewhere; click on that warning . The focus is then after the warning line. For example, if you click warning in the top line on the left side below, you get the right side below.

= $(a\Rightarrow b)\vee a$ warning
= $a \Rightarrow a\vee b$

= $(a\Rightarrow b)\vee a$ warning
= $a \Rightarrow a\vee b$

The focus can be filled in any of the usual four ways, causing the warning to be deleted or replaced, and the focus then follows the new line as usual, and the warning is now beside the newly created line. If the newly created line is the same as the line after the focus, the focus and warning beside it disappear, and the gap is closed.

For another example, if you click the warning on the left side below, you get the right side below.

```
=    ⟨n: nat → n−1⟩(i×i)    warning
=    i×i − 1
```

```
=    ⟨n: nat → n−1⟩(i×i)
⇐    i×i: nat    warning
     ┌──────────┐
     │          │
     └──────────┘
⇐    ⊤
=    i×i − 1
```

Program Example

```
⇐    s:= ΣL    frame
=    frame s· ⌞s′ = ΣL⌟    var
=    frame s· ⌜var n· ⌞s′ = ΣL⌟⌝    zoom in
⇐    s′ =⌞ΣL⌟    identity
=    s′ =⌜0 + ΣL⌝    substitution
=    s:= 0. s′ = s + Σ⌞L⌟    a list law
=    s:= 0. ⌞s′ = s + Σ⌜L[0,..#L]⌝⌟    substitution
=    s:= 0. ⌜n:= 0. s′ = s + ΣL[n;..#L]⌝    zoom out
=    frame s· var n· ⌜s:= 0. n:= 0. s′ = s + ΣL[n;..#L]⌝
```

At this point in the proof, we are tempted to zoom in on $s' = s + \Sigma L[n;..\#L]$ because that is what we want to refine now. But its refinement will be recursive, so for that reason, we do not zoom in. Instead, we start a new proof.

```
⇐    s′ = s + ΣL[n;..#L]    case creation
=    if n=#L then⌞n=#L ⇒ s′ = s + ΣL[n;..#L]⌟else n≠#L ⇒ s′ = s + ΣL[n;..#L] fi    zoom in
⇐    n=#L ⇒ s′ = s + ΣL[n;..#L]    context
=    #L=#L ⇒ s′ = s + ΣL[#L;..#L]    reflexive
=    ⊤ ⇒ s′ = s + ΣL[#L;..#L]    identity
=    s′ = s + ΣL[#L;..#L]    a string law
=    s′ = s + ΣL[nil]    a list law
=    s′ = s + Σ[nil]    a quantifier law
=    s′ = s + 0    identity
=    s′ = s    specialization
⇐    s′=s ∧ n′=n    definition
=    ok    zoom out
⇐    if n=#L then⌜ok⌝else⌞n≠#L ⇒ s′ = s + ΣL[n;..#L]⌟fi    zoom in
⇐    n≠#L ⇒ s′ = s + ΣL[n;..#L]    context
=    ⊤ ⇒ s′ = s + ΣL[n;..#L]    identity
=    s′ = s + ΣL[n;..#L]    warning
=    s′ = s + L n + ΣL[n+1;..#L]    substitution
=    s:= s + L n. ⌞s′ = s + ΣL[n+1;..#L]⌟    substitution
=    s:= s + L n. ⌜n:= n+1. s′ = s + ΣL[n;..#L]⌝    zoom out
⇐    if n=#L then ok else⌜s:= s + L n. n:= n+1. s′ = s + ΣL[n;..#L]⌝ fi
```

Execution

When the focus follows a program expression with no nonlocal variables and no input and no output, one of the suggestions is to execute it to obtain a value. WHAT IF IT HAS NO VALUE?

Undo

An undo key backs up one line in the proof pane. Further uses of undo back up further. DO WE WANT UNDO AS WELL AS DELETE? DO WE WANT REDO? REDO PREVIOUS?

Comments

PERHAPS a user should be permitted to add comments into the proof SOMEHOW.

Grammar

There is always a tension between the desire to make the notations as traditional as possible, so that they will be familiar, and the desire to make the notations as good as possible. For some notations, the answer is clear. For example, equality is denoted = by a 500 year-old tradition that's worldwide. Since equality is not self-dual, it would be better to use a symbol that is not vertically symmetric. But the tradition greatly outweighs the duality consideration. For another example, there is a tradition to use quantifiers such as Σ with a variable and starting value beneath it and ending value above it. That tradition is intolerable, so a better notation is used. For many other notations, it is not so clear whether to be traditional or good. In these unclear cases, aPToP leans toward the traditional, and Unified Algebra leans toward being good. The grammars offered below are exactly aPToP, except:

- The large versions of = ⇒ ⇐ have been left out. When an operator appears in the left margin of a proof, it has lowest precedence.
- The abbreviated function and quantifier notations that leave out the domain have not been included.
- A **let** construct has been added to Netty.
- Subscripting (string indexing) is denoted _ and superscripting (exponentiation) is denoted ^ .

Two grammars are offered in this section; the first is easier to read, and the second gives a more efficient parser. The first grammar is LR(1/2), meaning it has shift-shift choices, but no reduce-reduce choices and no shift-reduce choices, so follow sets are not needed. In the literature, it is called LR(0), but it shouldn't be called that because we do need to look at 1 symbol of input to choose among shift actions. The second grammar is LL(1/2). For each nonterminal, either: every production begins with a different terminal, or: every production except the last begins with a different terminal and the last either begins with a nonterminal or is empty; so director sets are not needed. For efficient parsing, the productions (except possibly the last) for each nonterminal should be put in order of decreasing frequency, but that hasn't been done.

In these grammars, “identifier” means something that appears in a declaration. Each identifier should be made internally distinct. If declared by ⟨ : → ⟩ it becomes an “elementvariable”. If declared by **var** it becomes a “boundaryvariable”. If declared by **ivar** it becomes an “interactivevariable”. If declared by **chan** it becomes a “channel”. If declared by **let** it becomes a synonym. And “empty” means the empty alternative.

LR(1/2) Grammar

expression	**let** identifier = expression · expression	raised dot
	var identifiers : expression · expression	raised dot
	ivar identifiers : expression · expression	raised dot
	chan identifiers : expression · expression	raised dot
	frame names · expression	raised dot

	quantifier identifiers : expression · expression	raised dot
	exp13	
exp13	exp13 . exp12	low dot
	exp13 ‖ exp12	
	exp12	
exp12	boundaryvariable := exp11	
	interactivevariable := exp11	
	channel ! exp11	
	channel !	
	channel ?	
	exp11	
exp11	exp11 ⇒ exp10	
	exp11 ⇐ exp10	
	exp10	
exp10	exp10 ∨ exp9	
	exp9	
exp9	exp9 ∧ exp8	
	exp8	
exp8	¬ exp8	
	exp7	
exp7	exp7 = exp6	
	exp7 ≠ exp6	
	exp7 < exp6	
	exp7 > exp6	
	exp7 ≤ exp6	
	exp7 ≥ exp6	
	exp7 : exp6	
	exp7 :: exp6	
	exp7 ∈ exp6	
	exp7 ⊆ exp6	
	exp7 ⊇ exp6	
	exp6	
exp6	exp6 , exp5	
	exp5 ,.. exp5	
	exp6	exp5	
	exp6 ◅ expression ▻ exp5	
	exp5	
exp5	exp5 ; exp4	
	exp4 ;.. exp4	
	exp5 ;; exp4	
	exp5 ‘ exp4	
	exp4	
exp4	exp4 + exp3	
	exp4 – exp3	
	exp4 ∪ exp3	
	exp3	
exp3	exp3 × exp2	
	exp3 / exp2	
	exp3 ∩ exp2	

	exp3 ↑ exp2	
	exp3 ↓ exp2	
	exp2	
exp2	– exp2	
	¢ exp2	
	$ exp2	
	↔ exp2	
	# exp2	
	exp1 * exp2	
	* exp2	
	~ exp2	
	♮ exp2	
	□ exp2	
	exp1 → exp2	function space
	√ channel	
	exp1 ^ exp2	exponentiation
	exp1 _ exp2	subscripting
	quantifier exp1	
	exp1	
exp1	exp1 @ exp0	
	exp1 exp0	list indexing, function application
	exp0	
exp0	(expression)	
	{ expression }	
	[expression]	
	⟨ identifiers : exp1 → expression ⟩	function
	if expression **then** expression **else** expression **fi**	
	⊤	
	⊥	
	number	
	text	
	name	
identifiers	identifiers , identifier	
	identifier	
names	names , name	
	name	
name	elementvariable	
	boundaryvariable	
	interactivevariable	
	channel	
	synonym	
quantifier	Σ	
	Π	
	∃	
	∀	
	§	

LL(1/2) Grammar

expression	**let** identifier = expression · expression
	var identifier maybemoreidentifiers : expression · expression
	ivar identifier maybemoreidentifiers : expression · expression
	chan identifier maybemoreidentifiers : expression · expression
	frame name maybemorenames · expression
	Σ afterquantifier
	Π afterquantifier
	∃ afterquantifier
	∀ afterquantifier
	§ afterquantifier
	exp13
maybemoreidentifiers	, identifier maybemoreidentifiers
	empty
name	elementvariable
	boundaryvariable
	interactivevariable
	channel
	synonym
maybemorenames	, name maybemorenames
	empty
afterquantifier	identifier maybemoreidentifiers : expression · expression
	exp11
exp13	exp12 maybemoreexp13
maybemoreexp13	. exp13
	‖ exp13
	empty
exp12	boundaryvariable := exp11
	interactivevariable := exp11
	channel afterchannel
	exp11
afterchannel	! exp11
	?
exp11	exp10 maybemoreexp11
maybemoreexp11	⇒ exp10 maybemoreexp11
	⇐ exp10 maybemoreexp11
	empty
exp10	exp9 maybemoreexp10
maybemoreexp10	∨ exp9 maybemoreexp10
	empty
exp9	exp8 maybemoreexp9
maybemoreexp9	∧ exp8 maybemoreexp9
	empty
exp8	¬ exp8
	exp6 maybemoreexp7
maybemoreexp7	= exp6 maybemoreexp7
	≠ exp6 maybemoreexp7
	< exp6 maybemoreexp7

	> exp6 maybemoreexp7	
	≤ exp6 maybemoreexp7	
	≥ exp6 maybemoreexp7	
	: exp6 maybemoreexp7	
	:: exp6 maybemoreexp7	
	∈ exp6 maybemoreexp7	
	⊆ exp6 maybemoreexp7	
	⊇ exp6 maybemoreexp7	
	empty	
exp6	exp5 maybemoreexp6	
maybemoreexp6	, exp5 maybemoreexp6	
	,.. exp5	
	| exp5 maybemoreexp6	
	◅ expression ▻ exp5 maybemoreexp6	
	empty	
exp5	exp4 maybemoreexp5	
maybemoreexp5	; exp4 maybemoreexp5	
	;.. exp4	
	;; exp4 maybemoreexp5	
	‘ exp4 maybemoreexp5	
	empty	
exp4	exp3 maybemoreexp4	
maybemoreexp4	+ exp3 maybemoreexp4	
	– exp3 maybemoreexp4	
	∪ exp3 maybemoreexp4	
	empty	
exp3	exp2 maybemoreexp3	
maybemoreexp3	× exp2 maybemoreexp3	
	/ exp2 maybemoreexp3	
	∩ exp2 maybemoreexp3	
	↑ exp2 maybemoreexp3	
	↓ exp2 maybemoreexp3	
	empty	
exp2	– exp2	
	¢ exp2	
	$ exp2	
	↔ exp2	
	# exp2	
	* exp2	
	~ exp2	
	♮ exp2	
	□ exp2	
	√ channel	
	exp1 maybemoreexp2	
maybemoreexp2	* exp2	
	→ exp2	
	^ exp2	exponentiation
	_ exp2	subscripting
	empty	

exp1	exp0 maybemoreexp1
maybemoreexp1	@ exp0 maybemoreexp1
	(expression) maybemoreexp1
	{ expression } maybemoreexp1
	[expression] maybemoreexp1
	⟨ identifier maybemoreidentifiers : exp1 → expression ⟩ maybemoreexp1
	if expression **then** expression **else** expression **fi** maybemoreexp1
	⊤ maybemoreexp1
	⊥ maybemoreexp1
	number maybemoreexp1
	text maybemoreexp1
	elementvariable maybemoreexp1
	boundaryvariable maybemoreexp1
	interactivevariable maybemoreexp1
	channel maybemoreexp1
	empty
exp0	(expression)
	{ expression }
	[expression]
	⟨ identifier maybemoreidentifiers : exp1 → expression ⟩
	if expression **then** expression **else** expression **fi**
	⊤
	⊥
	number
	text
	elementvariable
	boundaryvariable
	interactivevariable
	channel