Chapter 7

Definition and Presentation of Terms

Nuprl offers users an opportunity to extend the basic language of type theory by introducing new, abstract terms whose meaning is defined in terms of existing language constructs. In addition to that users may also modify the visual appearance of abstract terms[1] and adjust the presentation of formal material without changing the formal content itself, which makes it possible to use familiar notation or to present the same material differently to different target groups.

Users can create several kinds of library objects for this purpose. *Abstractions* are used to introduce the *abstract definition* of a new term, *display forms* define the *textual presentation* of abstract terms, and *precedence objects* assign *precedences* for terms to control automatic parenthesization. In Section 4.3.2.2 we briefly described how to create abstractions and display forms using the navigator's `AddDef*` button. In this chapter we will describe the contents and features of abstractions, display forms, and precendences as well as editor support for defining them.

7.1 Abstractions

Abstractions are terms that are definitionally equal to other terms. In Nuprl they may be defined in terms of language primitives and other abstractions, but the dependency graph for abstractions should be acyclic. In particular, an abstraction not depend on itself. Recursive definitions can be introduced using the `AddRecDef` button as described in Section 4.3.2.2.

Abstraction definitions have form

lhs `==` *rhs*

where *lhs* and *rhs* are *pattern* terms that may contain free variables. The latter are implicitly universally quantified. When Nuprl *unfolds* some *lhs-inst* instance of *lhs*, it first matches this instance against the pattern *lhs* and generates bindings for the free variables of *lhs* accordingly. It then applies these bindings to the free variables in *rhs* to calculate the term *rhs-inst* into which *lhs-inst* unfolds. Therefore, all free variables of *rhs* must also occur free in since otherwise unfolding a definition would yield a term with unbound variables. An example of an abstraction object is given below.

- ABS: int_seg

$\{i..j^{-}\}$ `==` $\{ k:\mathbb{Z} \mid i \leq k < j \}$

[1] This includes user defined terms, primitive terms of Nuprl's type theory, and even the terms used for describing Nuprl editing features such as the navigator, proof terms, or the appearance of abstractions and display forms.

The abstraction defines a type of segments of integers. The abstraction object `int_seg` already consults a display form in the presentation of the left hand side of the definition. The structure of the left hand side becomes more readily apparent if we write it in uniform syntax (which can be made visible by exploding the term as described in Section 5.4.5). ⌞`{i..j⁻}`⌟ is ⌞`int_seg(i;j)`⌟, a term with opid `int_seg`, no parameters, and two subterms. An instance of the left hand side is ⌞`{0..10⁻}`⌟, which would unfold to ⌞`{ k:ℤ | 0 ≤ k < 10 }`⌟.

Just as abstractions can be unfolded by applying their definition left-to-right, so instances of their right hand sides can be *folded* up to be instances of their left hand sides. Folding, however does not always work, as information can be lost in the unfolding process. For instance, an abstraction can have variables and parameters that are not used in its definition but are only used for "book-keeping purposes". In this case the variables and parameters only occur on the left hand side of the definition and would have to be inferred when folding up a specific instance of the right hand side.

7.1.1 Bindings in Abstractions

In additions to ordinary variables, abstractions can have binding structure. Consider, for instance, the definition of the *unique existence* quantifier below.

- ABS: exists_uni
`∃!x:T. P[x] == ∃x:T. P[x] ∧ (∀y:T. P[y] ⇒ y=x∈T)`

Here `x` represents a variable that becomes bound in the term `P[x]` and this binding structure must be mapped from the abstract term ⌞`exists_uni(T; x.P[x])`⌟ to its definition

First-order matching and substitution are inadequate for handling terms with binding structure, since they consider variables to be independent from each other and thus cannot express the dependency between `x` and `P[x]`. NUPRL's therefore uses *second-order* matching and substitution functions to handle abstractions with binding variables in a systematic way.

A *second-order binding* is a binding $v \mapsto x_1,\ldots,x_{a_n}.t$ of a second-order variable v to a second-order term $x_1,\ldots,x_{a_n}.t$. A *second-order variable* is essentially an identifier as with normal variables, but it also has an associated *arity* $n{\geq}0$. *Second-order terms* are a generalization of terms that can be thought of as 'terms with holes', i.e. as terms with missing subtrees.[2] They can be represented by bound-terms such as $x_1,\ldots,x_{a_n}.t$, where the binding variables are place-holders for the missing subtrees. In a second-order binding $v \mapsto x_1,\ldots,x_{a_n}.t$, the arity of v must be equal to n.

An *instance* of a second-order variable v with arity n is a term $v[a_1;\ldots;a_n]$, where $a_1,\ldots,a_n$ are terms, also called the *arguments* of v. A *second-order substitution* is a list of second-order bindings. The result of applying the binding $[v \mapsto w_1,\ldots,w_n.t_{w_1,\ldots,w_n}]$ to the variable instance $v[a_1;\ldots;a_n]$, is the term $t_{a_1,\ldots,a_n}$ – the arguments of the instance of the second-order variable fill the holes of the second-order term.

In our above example, `P` is a second order variable with arity 1, and the terms `P[x]` and `P[y]` are second-order-variable instances. Consider unfolding an instance of the left-hand side, say the term ⌞`∃!i:ℤ. i=0∈ℤ`⌟. The substitution generated by matching this against ⌞`∃!x:T. P[x]`⌟ would be

`[ P ↦ i. i=0∈ℤ ; T ↦ ℤ ]`

and the result of applying this to the right hand side of the definition would be

`∃x:T. x=0∈ℤ ∧ (∀y:ℤ. y=0∈ℤ ⇒ y=x∈ℤ)`

[2]Roughly, second-order terms are like functions on terms but there are subtle differences between the two concepts.

Actually, the matching and substitution functions used by NUPRL are a little smarter than shown above, as they try to maintain names of binding variables. The result one would get in NUPRL would be ⌞∃i:T. i=0 ∈ ℤ ∧ (∀y:ℤ. y=0 ∈ ℤ ⇒ y=i ∈ ℤ)⌟.

NUPRL does not allow nested bindings on the left-hand side of abstraction definitions. All variables must either be first-order or second-order variables with first-order variable arguments.

7.1.2 Parameters in Abstractions

Abstractions can also contain *meta-parameters*, i.e. placeholders for parameters that matching and substitution treat as variables. We usually indicate that a parameter is meta by prefixing it with a $ sign. For example, we might define an abstraction `label`{x:`t`,i:`n`} as shown below

```
- ABS: label

label{$tok:t,$nat:n}() == <"$tok",$nat>
```

Meta-parameters make it possible to map parameters in newly defined abstractions onto parameters of existing terms. In the above example labels are defined as pairs of tokens and natural numbers and the parameter `$tok` is mapped onto the parameter of the term `token` while the parameter `$nat` is mapped onto the parameter of the term `natural_number`, which is revealed when the right hand side of the definition is exploded into ⌞`pair(token{$tok:t};natural{$nat:n})`⌟.

Level-expression variables occurring in level-expression parameters of abstraction definitions are always considered meta-parameters, so there is no need to designate them explicitly. However, indicating meta-parameters explicitly makes it easier to identify them as such.

In general, the term on the left-hand side of an abstraction can have a mixture of normal and meta-parameters. You can define a family of abstractions which differ only in the constant value of some parameter. However, it is an error to make two abstraction definitions with left-hand sides that have some common instance.

7.1.3 Attributed Abstractions

A recently added feature of abstraction definitions is an optional list of *attributes* or *conditions*. An attribute is simply an alpha-numeric label associated with the abstraction and the general form of an abstraction with conditions $c_1, \ldots, c_n$ is:

$$(c_1, \ldots, c_n)\texttt{::}\,lhs\ \texttt{==}\ rhs$$

Abstraction conditions can be used to hold information about abstractions that may be useful to tactics and other parts of the NUPRL system. They could, for instance, be used to group abstractions into categories, and when doing a proof, one could ask for all abstractions in a given category to be treated in a particular way, e.g. to unfold all abstractions of a category "`notational abbreviations`".

7.1.4 Editor Support

In this section, we describe the editor support for abstraction objects. An abstraction can be viewed by opening it with the navigator (Section 4.3.1.1) or by using the VIEW-ABSTRACTION command (⟨C-`X`⟩`ab`) on a term containing an instance of it (Section 5.7). The following commands and key sequences may be used for editing abstractions.

⟨C-M-`I`⟩	INITIALIZE	initialize object / condition
`so_var`n	INSERT-TERM`so_var`n	insert second order var with n args
⟨C-`M`⟩	CYCLE-META-STATUS	make parameter meta / normal
⟨C-M-`S`⟩	SELECT-TERM-OPTION	open condition sequence
⟨C-`O`⟩	OPEN-SEQ-TO-LEFT	open slot in cond seq to left
⟨M-`O`⟩	OPEN-SEQ-TO-RIGHT	open slot in cond seq to right

Since most abstraction objects are created using the `AddDef*` mechanism described in Section 4.3.2.2, the left and right hand side of the abstraction is already present when the object is opened. In the rare case that the abstraction object was created with the `MkObj*` command button (Section 4.3.2.1) it will contain an empty term slot when it is first visited, which must be initialized before a definition can be entered.

The INITIALIZE command will create an uninstantiated abstraction definition term, which looks like:

```
[ab lhs] == [ab rhs]
```

To enter the abstract term on the left hand side of the definition, one has to provide its *object identifier*, its *parameters*, and a list of its *subterms* together with the variables to be bound in these subterms. Ways to create new terms with the term-editor are described in Sections 5.4.4 and 5.4.5. The term for the right hand side of the definition is entered in the usual structural top-down fashion of the term-editor as explained in Section 5.4.

`so_var`n has to be used to enter second order variable instances on the left and right hand sides of the definition. This will insert a term of the form $\llcorner$`variable{`x`:v}(`a_1`;`$\ldots$`;`a_n`)`$\lrcorner$ where x is the variable's name, and $n > 0$ is a natural number. The library display form object for this term is named `so_var`n so this family of names can be used to reference them.

Note that abstraction objects are the *only* places where these second-order variable instances are used. When writing propositions, second-order variable instances are simulated using the `so_apply`n abstraction.

CYCLE-META-STATUS converts a parameter into a meta-parameter if the text cursor is in the parameter's text slot. If the parameter is already meta, using this twice will cycle its status back to being a normal parameter.

SELECT-TERM-OPTION enables a user to add conditions to an abstraction. By default, an abstraction definition term has an empty condition sequence as a subterm, which is hidden by the display form for abstractions. Moving the term cursor over the whole abstraction term and using SELECT-TERM-OPTION will add an empty term slot for a condition. The condition term is much like the term for variables; it has a single text slot, and otherwise no other display characters. To get additional slots for condition terms one may use OPEN-SEQ-TO-LEFT or OPEN-SEQ-TO-RIGHT.

7.2 Term Display

Display form objects are used to control the visual presentation of formal mathematical concepts. They define how a term shall appear when it is being displayed on the screen or printed on paper. This enables users to present formal content within a variety of notations without having to change the internal logical representation of these terms. Display forms are commonly created whenever a definition is introduced using the `AddDef*` mechanisms (Section 4.3.2.2), but they may also be added for existing abstractions as well as for the primitive terms of the library.

$$
\begin{array}{rcl}
\textit{def-seq} & ::= & \textit{definition}\ \texttt{;;} \\
 & | & \textit{definition}\ \texttt{;;}\ \textit{def-seq} \\
\textit{definition} & ::= & \textit{format-seq}\ \texttt{==}\ \textit{term} \\
 & | & \textit{attr-seq}\ \texttt{::}\ \textit{format-seq}\texttt{==}\ \textit{term} \\
\textit{format-seq} & ::= & \textit{format} \\
 & | & \textit{format}\ \textit{format-seq} \\
\textit{attr-seq} & ::= & \textit{attribute} \\
 & | & \textit{attribute}\ \texttt{::}\ \textit{attr-seq}
\end{array}
$$

Figure 7.1: Display Object Structure

In NUPRL the presentation of all formal content, including the appearance of the navigator, the editor, sequents, and proofs is controlled by display forms and may be adjusted according to the preferences of a user. Even the mechanisms for editing and presenting display forms themselves may be modified. The display forms for the quantifiers (`all_df`, `exists_df`) and the logical connectives (`and_df`, `or_df`, ...), for instance, appear as "display form generators".

The top level structure of a display form object is summarized by the grammar shown in Figure 7.1. An object contains one or more display form *definitions*. Each definition has a right-hand-side *term* to which the display form applies, and a sequence of *formats* that specify how to display the term. A definition also has an optional sequence of *attributes* that specify extra information about the definition.

Usually, all the definitions in one object refer to a closely related set of terms. When choosing a display form to use for a term, the layout algorithm tries definitions in a backward order, so definitions are usually ordered from more general to more specific.

7.2.1 Editing Display Form Objects

Since most display forms are created using the `AddDef*` mechanism described in Section 4.3.2.2, a right-hand-side term, a standard sequence of formats, and an alias attribute (see Section 7.2.4 below) are already present when the object is opened. If the object was created with the `MkObj*` command (Section 4.3.2.1) it will contain an empty term slot, which must be initialized before a display form definition can be entered.

The command ⟨C-M-`I`⟩ will create an initial display form definition, which looks like:

```
== [rhs]
```

To get additional slots for display form definitions one may use the commands ⟨C-`O`⟩ and ⟨M-`O`⟩.

An initial display form definition has an empty attribute sequence as a subterm, which is hidden by the display form for display form definitions. Moving the term cursor over the whole term and using ⟨C-M-`S`⟩ will add an empty term slot for an attribute.

7.2.2 Right-hand-side Terms

The right-hand-side term is a pattern. A definition applies to some term t if t is an instance of the right-hand-side term. The display definition matcher has a notion of *meta-variable* different from that of NUPRL's usual matching routines; it has 3 kinds of meta-variable: *meta-parameters*, *meta-bound-variables*, and *meta-terms*.[3] Meta-parameters and meta-bound-variables correspond to text slots on the left-hand side of a definition, and meta-terms correspond to term slots.

[3]The meta-parameters are different from those used in abstraction definitions. To be clear, we sometimes call those ones *abstraction-meta-variables* and the ones in display definitions, *display-meta-variables*.

The right-hand-side term is restricted to being a term whose subterms are either constant terms, i.e. terms with no meta-variables, or meta-terms. To enter a meta-term into a term slot one has to use the name `mterm`. To turn parameters and variable into meta-parameters or meta-bound-variables, position a text cursor in the appropriate parameter or bound variable slot and give the CYCLE-META-STATUS command ⟨C-M⟩ (twice). Display-meta-variables are readily recognized because they have <> as delimiters.

The rhs right-hand-side term may also contain normal parameters, bound variables and variable terms. These are treated like constants: for a definition to be applicable, they must match exactly.

7.2.3 Format Sequences

Format sequences are text sequences that may contain slots for meta-variables and commands for controlling the layout of formal material through insertion of optional spaces, line breaking, and indentation. Except for text strings, all formats must be entered into term slots, which may be created as described in Section 5.4.2.

The various kinds of formats are summarized in the table below. The *Name* column gives the name that has to be entered into a term slot to create the format, while the *Display* column describes how the format will be presented within a display form definition.

Name	*Display*	*Description*
slot	<*id*:*ph*>	text slot format
lslot	<*id*:*ph*:L>	term slot format
eslot	<*id*:*ph*:E>	term slot format
sslot	<*id*:*ph*:*>	term slot format
pushm	{→*i*}	push margin
popm	{←}	pop margin
break	{*a*}	break
sbreak	{\\?*a*}	soft break
hzone	{[HARD}	start hard break zone
szone	{[SOFT}	start soft break zone
lzone	{[LIN}	start linear break zone
ezone	{]}	end break zone
space	{Space}	optional space

7.2.3.1 Slot Formats

Slot formats are placeholders for the children of a display form instance. Text slots are generally used for meta-parameters and meta-bound-variables, while term slot formats contain meta-terms.

The *id* in a slot format is the name of the slot. The slot corresponds to the meta-variable of the right-hand-side term with the same name. *ph* is place-holder text, which will appear (enclosed within []'s) in the slot whenever it is uninstantiated in some instance of the display form. The L,E and * options on the term slot formats control parenthesization of the slot and are discussed in Section 7.2.4.2.

7.2.3.2 Margins

The margin control format {→i}, where $i \geq 0$, pushes a new left margin i characters to the right of the format position onto the *margin stack*. The layout algorithm uses the top of the margin stack to decide the column to start laying out at after a line break. To create the format, enter `pushm` into a term slot and edit the number 0 in the format `{→0}` accordingly.

The margin control format `{←}` (`popm`) pops the current margin off the top of the margin stack and restores the left margin to a previous margin. Usually display forms should have matching `pushm`'s and `popm`'s.

7.2.3.3 Line Breaking

Line-breaking formats divide the display into nested *break zones*. There are 3 kinds of break zone: *hard*, *linear*, and *soft*. The effect of the `break` format `{\\`a`}` depends on the break zone kind:

- In a *hard* zone, `{\\`a`}` always causes a line break.
- In a *soft* zone, either none or all of the `{\\`a`}` are taken.
- In a *linear* zone, `{\\`a`}` never causes a line break. Instead, its position is filled by the text string a, which usually is a sequence of blank characters.

Break zones are started and ended by zone delimiters. Display form format sequences should usually include matching start and end zone formats. There is one end delimiter `{]}` (`ezone`) for all kinds of zones. Each kind of zone has its own start delimiter:

- `{[HARD}` (`hzone`) starts a hard zone.
- `{[SOFT}` (`szone`) starts a soft zone.
- `{[LIN}` (`lzone`) starts a linear zone.

A linear zone is special in that all zones nested inside are also forced to be linear. Therefore a linear zone contains no line-breaks and always is laid out on a single line. If a linear zone doesn't fit on a single line, the layout algorithm chooses subterms to elide (see Section 5.3) to try and make it fit.

When laying out a soft zone, the layout algorithm first tries treating it as a linear zone. If that would result in any elision, then it treats the zone as a hard zone.

The *soft break format* `{\\?`a`}` (`sbreak`) is similar to the break format but is not as sensitive to the zone kind. Soft breaks in linear zones are never taken, but otherwise, the layout algorithm uses a separate procedure to choose which soft breaks to take and which not. This procedure uses various heuristics to try and layout a term sensibly in a given size window with at little elision of subterms as possible.

7.2.3.4 Optional Spaces

The `space` format `{Space}` inserts a single blank character if the character before it isn't already a space. Otherwise it has no effect.

7.2.4 Attributes

Attributes specify extra information about display form definitions. By default, display form definitions are created with a right-hand-side term, a standard sequence of formats, and a single alias attribute. Moving the cursor over the whole attribute term and using ⟨C-`O`⟩ or ⟨M-`O`⟩ will create additional attribute slots to the left or right of this attribute.

Possible display form attributes are summarized in the table below. The *Name* column gives the name that has to be entered into a term slot to create the attribute, while the *Display* column describes how it will be presented within a display form definition.

Name	*Display*	*Description*
`alias`	`EdAlias` a	alias for definition input
`ithd`	`#Hd` a	head of iteration family
`ittl`	`#Tl` a	tail of iteration family
`parens`	`Parens`	parenthesis control
`prec`	`Prec` a	precedence
`index`	`Index` a	definition name
`conds`	`(`c_1`,...,`c_n`)`	conditions

The `alias` attribute provides an alternate name which the input editor recognizes as referring to the definition. Alternate names are often convenient abbreviations for the full names of definitions. The iteration attributes `ithd` and `ittl` control selection of a definition by the display layout algorithm. They are used to come up with convenient notations for iterated structures, which are discussed in Section 7.2.4.1. The `parens` and `prec` attributes affect automatic parenthesization, described in Section 7.2.4.2. The `index` attribute together with the name of the object containing a definition give a unique name for the definition. Conditions specify requirements for using a display form definition. Each condition $c_1,\ldots,c_n$ in the `conds` term is a term with a alpha-numeric label associated with the display form definition.

7.2.4.1 Iteration

The iteration attributes control choice of display form definition based on immediately-nested occurrences of the same term. The idea is to group occurrences into *iteration families*. An iteration family has a *head* display form definition and one or more tail definitions. A tail definition can only be used as an immediate subterm of a head in the same family or another tail in the same family. Choice of display form is also affected by the use of the *iterate variable* `#` as the *id* of a term slot format (Section 7.2.3.1). If `#` is used in some term slot of a definition, then the definition is only usable if the same term occurs in the subterm slot that uses the `#`.

The following set of display forms for λ abstraction terms, for instance, makes sure that the λ character is suppressed on nested occurrences:

```
λ<x:var>.<t:term:E>== lambda(<x>.<t>)
#Hd A ::λ<x:var>,<#:term:E>== lambda(<x>.<#>)
#Tl A ::<x:var>.<t:term:E>== lambda(<x>.<t>)
#Tl A ::<x:var>,<#:term:E>== lambda(<x>.<#>)
```

Using these, the term `lambda(x.lambda(y.lambda(z.x)))` will be displayed as ⌞`λx,y,z.x`⌟ instead of ⌞`λx.λy.λz.x`⌟.

7.2.4.2 Parenthesization

Automatic parenthesization is controlled by certain display definition attributes, term slot options, and by definition *precedences*. A *precedence* is an element in the *precedence order*, which is determined by the precedence objects in the NUPRL library. A display form definition is assigned a precedence by giving it a `prec` attribute which names some precedence element.

Precedence Objects collectively introduce a set of precedence elements, and define a partial order on them. The components of a precedence object and the names used to enter them by are summarized in the table below. The `prser`, `preq`, and `prpar` terms are sequence constructors that may be nested. The standard editor commands described in Section 5.4.2 work on the sequences built with these terms.

Name	*Display*	*Description*
`prser`	(p_1>...>p_n)	serial precedence term
`preq`	{p_1=...=p_n}	equal precedence term
`prpar`	[p_1|...|p_n]	parallel precedence term
`prel`	*obname*	element of precedence order
`prptr`	**obname**	precedence object pointer

Object names and object pointers are the primitive elements in a precedence order. Serial precedence terms impose a linear order on a set of precedences $p_1 \ldots p_n$. Equal precedence term declare all precedences p_i to be equal in the precedence order. Parallel precedence terms declare all precedences p_i to have the same "rank" in the precedence order while being unrelated to each other.

Each display form not explicitly associated with any precedence element is implicitly associated with a unique precedence element unrelated to all other precedence elements. The uniqueness implies that two such display forms have unrelated precedence.

Examples of a base set of precedences set up for the current NUPRL theories can be found in the standard theory `core_1`.

Automatic Parenthesis Selection. The parenthesization of a term slot of a display form is controlled by the *parenthesis slot-option*, i.e. the third field of the term slot in the display form definition (see Section 7.2.3.1), by the `parens` attribute of the display form filling that term slot, and by the relative precedences of the term slot and the term filling it. The precedence of a term slot is usually that of the display form containing it, although it is possible to assign precedences to individual slots. The parenthesis control works as follows:

- Term slots are parenthesized only if the filling display form has a `parens` attribute. If this attribute is absent, the slot is never parenthesized. The `parens` attribute must be explicitly added to a display form definition for that definition to ever be parenthesized.
- Term slots are parenthesized unless parenthesization is suppressed by the parenthesis slot-options. These options have the following meanings:

  `L` : Suppress parentheses if the display-form precedence is *less than* the display-form precedence of the term filling the slot.

  `E` : Suppress parentheses if the display-form precedence is *less than or equal to* the display-form precedence of the term filling the slot.

  `*` : Always suppress parentheses.

The `L` and `E` options make it possible to represent the conventional precedences and associativity laws of standard infix operators. If they are used in the definitions of display forms for the arithmetic terms `plus(`a`;`b`)`, and `times(`a`;`b`)`, then the term `plus(`a`;times(`b`;`c`))` is displayed as $\llcorner a+b*c \lrcorner$, but `times(`$a$`;plus(`$b$`;`$c$`))` is displayed as $\llcorner a*(b+c) \lrcorner$. Similarly, `function(`$A$`;function(`$B$`;`$C$`))` is displayed as $\llcorner A{\rightarrow}B{\rightarrow}C \lrcorner$, but `function(function(`$A$`;`$B$`);`$C$`)` is displayed as $\llcorner (A{\rightarrow}B){\rightarrow}C \lrcorner$

The `L`, `E` and `*` characters in the display of term slot formats are display forms for parenthesization control terms. To change the parenthesis slot-options, one may delete the term and enter the new option using the names shown in the table below.

Name	*Display*	*Description*
`lparens`	`L`	L option
`eparens`	`E`	E option
`sparens`	`*`	* option

The parenthesization control terms also allow the specification of the delimiter characters used for parenthesization, and a precedence for the individual slot. No specific editor support has yet been provided for these features.

7.2.5 Examples

As an example, we walk through the entry of a display form definition from scratch. We start by creating a new display form object and viewing it. Click the `MkObj*` button, enter ⌞`tst_df`⌟ as name and ⌞`disp`⌟ as kind, and click the `OK*` button.[4]

Open the object by pressing the right arrow or clicking on it with the mouse. This will pop up a window, containing a highlighted empty term slot. Initialize the display form definition by entering ⟨C-M-`I`⟩. The window now looks like:

```
- DISP:: tst_df
== [rhs]
```

We begin by entering the right-hand side of the display form. Click LEFT on the `[rhs]` placeholder, and enter ⌞`exists_unique(0;1)`⌟ to create a new term (see Section 5.4.4). Do not fill in the variable slot or either of the subterm slots. The definition should now look like:

```
- DISP:: tst_df
== exists_unique([term];[binding].[term])
```

Enter ↵ or click LEFT on the left-most term slot and enter `<T>`, `<x>`, and `<P>` as meta-terms and meta-variable, respectively, by typing ⌞`mterm`↵`T`↵`x`⟨C-M⟩↵`mterm`↵`P`⌟. As a result you get:

```
- DISP:: tst_df
== exists_unique((<T>;<x>.<P>))
```

To create a display form for the term on the right hand side, click LEFT on the first `=` to get a text cursor in the empty format sequence on the left-hand side of the definition. Type ⌞⟨C-`#`⟩`163!`⟨C-`O`⟩`slot`↵`x`↵`var`⟨C-F⟩⌟ to generate an ∃ symbol and an exclamation mark as initial text and a slot for the variable `x`. The definition should now look like:

```
- DISP:: tst_df
∃!<x:var>== exists_unique((<T>;<x>.<P>))
```

Enter a colon, the type slot, a period, a space, and the second term slot:

⌞`:`⟨C-`O`⟩`sslot`↵`T`↵`type`⟨C-F⟩⟨C-F⟩⟨C-F⟩`. `⟨C-`O`⟩`eslot`↵`P`↵`prop`⌟

The definition should now look like:

[4] Note that the `MkObj*` button is not present in user theories. The only way to create display forms is through the `AddDef*` and `AddDefDisp*` buttons, which already generate a right-hand-side term, a standard sequence of formats, and an alias attribute.

```
- DISP:: tst_df

∃!<x:var>:<T:type:*>. <P:proP:E>== exists_unique((<T>;<x>.<P>)
```

The display form definition is now complete and may be saved by closing the display form object with ⟨C-Z⟩. However, the definition does not yet include line breaking or parenthesization information. In particular, it does not contain any visible delimiter for the end of the `prop` slot.

We therefore want the layout algorithm to automatically parenthesize the display form. To add parenthesizing attributes, click LEFT on the second `=` character to get a term cursor over the whole definition, and then enter ␣⟨C-M-S⟩ ↵ ⟨C-O⟩␣ to get two empty attribute slots, with a term cursor over the first:

```
- DISP:: tst_df

[attr]::[attr]::∃!<x:var>:<T:type:*>. <P:prop:E>
 == exists_unique((<T>;<x>.<P>)
```

To instantiate the attribute slots enter ␣`parens` ↵ `prec` ↵ `exists`␣ To get:

```
- DISP:: tst_df

Parens::Prec(exists|)::∃!<x:var>:<T:type:*>. <P:prop:E>
 == exists_unique((<T>;<x>.<P>)
```

This means that we assign the same precedence to the term $\exists!x{:}T.P_x$ as is assigned to the term $\exists x{:}T.P_x$ in the standard libraries.

We may also add a soft-break format such that the period separating the `type` slot from the `prop` slot does not appear if a break is taken. Click LEFT on the ␣.␣ character and delete it using ⟨C-D⟩. Enter ␣⟨C-O⟩`sbreak`SPC␣, click LEFT on the } after the `?` character in the soft break display form, and enter ␣.␣.

```
- DISP:: tst_df

Parens::Prec(exists)::∃!<x:var>:<T:type:*>{\?.|} <P:prop:E>
 == exists_unique((<T>;<x>.<P>)
```

Finally, we add a second display form for the term $\exists!x{:}T.P_x$, which drops the type information from the display whenever the type is $\mathbb{N}$.

Click LEFT on the second `=` character to get a term cursor over the whole definition, and then enter ␣⟨M-O⟩␣ to get a second initial display form definition *after* it.

```
- DISP:: tst_df

Parens::Prec(exists)::∃!<x:var>:<T:type:*>{\?.|} <P:prop:E>
 == exists_unique((<T>;<x>.<P>)

== [rhs]
```

Copy the first definition into the second as follows: enter ⟨C-K⟩ to replace the initial display form by an empty term slot, move the term cursor over the whole first definition, copy it with ⟨M-K⟩, move the term cursor back over the empty term slot, and paste the first definition with ⟨C-Y⟩.

```
- DISP:: tst_df

Parens::Prec(exists)::∃!<x:var>:<T:type:*>{\?.} <P:prop:E>
 == exists_unique((<T>;<x>.<P>)

Parens::Prec(exists)::∃!<x:var>:<T:type:*>{\?.} <P:prop:E>
 == exists_unique((<T>;<x>.<P>)
```

On the right hand side of the definition, replace the meta-term `<T>` by the type constant ℕ. Click LEFT on the `<` character, enter ⟨C-`K`⟩ and then type `nat ↵` to enter the type ℕ.

On the left hand side, remove the term slot for `T` by clicking LEFT on the `>` character (!) and entering ⟨C-`K`⟩. Remove the colon with BACKSPACE.

```
- DISP:: tst_df

Parens::Prec(exists)::∃!<x:var>:<T:type:*>{\?.} <P:prop:E>
 == exists_unique((<T>;<x>.<P>)

Parens::Prec(exists)::∃!<x:var>{\?.} <P:prop:E>
 == exists_unique((ℕ;<x>.<P>)
```

In a similar way, one may add further display form definitions with iteration families to suppress the `∃!` string and duplicate type information in nested occurrences of the term $\exists! x{:}T.P_x$. The display form object `exists_df` in the standard theory `core_1` can be used as an example for doing that.