FraGoLi

Documentation for the LaTeX package ‘fragoli’

fragoli \v 1.2.2
Paul Eduard Koenig
Goethe University Frankfurt,
Institute of Linguistics
pauleduardkoenig@gmail.com

June 5, 2025

Abstract

This document provides an overview of the LaTeX package `fragoli` (Frankfurt Goethe Linguistic). The package is primarily designed for use in the field of semantics, offering a streamlined syntax to facilitate the rapid derivation of semantic formulae. Its notation style is loosely based on the supplementary materials accompanying the introductory course in linguistic semantics by *Thomas Ede Zimmermann*.

Contents

1 Background **2**

2 Examples **3**

3 Usage **6**
3.1 Package Options . . . 6
3.1.1 language . . . 6
3.1.2 userainbow . . . 7
3.1.3 usetypes . . . 7
3.1.4 typestyle . . . 7
3.1.5 typesspacing . . . 7
3.1.6 typenestingstyle . . . 8
3.1.7 lambdaheadstyle . . . 8
3.1.8 lambdaheadspacing . . . 8
3.1.9 usedscount . . . 8
3.1.10 showawnser . . . 8
3.1.11 rulesituation . . . 9
3.1.12 situation . . . 9
3.1.13 situationspacing . . . 9
3.1.14 tracespacing . . . 9
3.1.15 semanticsuperscriptspacing . . . 9
3.1.16 assignmentspacing . . . 10
3.1.17 alternativecommands . . . 10

4 Commands **10**
4.1 Text . . . 10
4.1.1 Object Language . . . 11
4.1.2 Meta Language . . . 11
4.2 Bracketing . . . 12
4.3 Logic . . . 13
4.4 Lambda . . . 16
4.5 Semantics . . . 19
4.5.1 Direct Interpretation . . . 21
4.5.2 Indirect Interpretation . . . 22

4.5.3 Rules . . . 25
4.6 Constants . . . 25
4.6.1 Situations . . . 25
4.6.2 Lambda Heads . . . 25
4.6.3 Types . . . 25
4.6.4 Traces . . . 26

5 Environments **26**
5.1 exams . . . 26
5.2 fglfunc . . . 27
5.2.1 Commands . . . 27
5.3 cps . . . 28
5.4 fgls . . . 29
5.4.1 Constants . . . 29
5.4.2 Commands . . . 33
5.5 semderivation . . . 41
5.5.1 Constants . . . 41
5.5.2 Commands . . . 41
5.6 semcalc . . . 42
5.6.1 Commands . . . 42
5.7 semlang . . . 43
5.7.1 Commands . . . 43
5.8 semtree . . . 44
5.9 semtreesem . . . 44
5.10 semtreesyn . . . 44
5.11 semrule . . . 45
5.11.1 Commands . . . 45
5.12 semrulesyn . . . 53
5.13 semlex . . . 53
5.13.1 Commands . . . 54
5.14 semderi . . . 62
5.14.1 Constants . . . 62
5.14.2 Commands . . . 62

6 Changelog **63**

1 Background

This package brings together and refines a collection of LaTeX commands and concepts developed over the years within the institute of linguistics at the Goethe University Frankfurt. In the process of preparing research papers, assignments, and examinations, numerous custom LaTeX headers and commands were shared within the department—some mutually compatible, others not. This package consolidates these resources into a cohesive system. Special thanks are due to *Thomas Ede Zimmermann*, *Cécile Meier*, *Daniel Gutzmann*, and *Jan Köpping*, whose files form the foundation of this package. Also thanks to *Manuel Lipstein* for his feedback.

The primary goal of this package is to provide a minimal and user-friendly syntax for constructing large and complex semantic derivations, following the specific notational style used at Goethe University Frankfurt. It includes a comprehensive set of commands for text formatting and various types of bracketing, ensuring a consistent style—particularly when distinguishing between meta-language and object-language within a single derivation or formula.

The semantics-specific commands in this package are organized into three categories: direct interpretation, represented using standard bracketing $[\![\textbf{expression}]\!]$; indirect interpretation, which is further divided into translation || and denotation $\|\textbf{expression}\|$. These commands offer extensive customization options and include predefined macros for commonly used constants, lambda expressions, and logical notation in formal semantics and type theory.

Additionally, the package includes specialized commands and environments that extend beyond standard semantic notation. One such feature is the *cps* environment, which implements a rainbow bracketing system inspired by modern programming editors and IDEs. This system assigns matching colors to parentheses at the same depth, with each level of nesting receiving a distinct color. This visual aid significantly enhances the readability of complex derivations and formulae, making them easier to process and debug. However,

due to the intricate way LaTeX handles command execution order, this feature currently does not integrate well with commands that generate parentheses as part of their output. While future updates may address this limitation, at present, only manually written parentheses within the *cps* environment are affected by the coloring scheme. As an example see:
`\begin{cps}(level1(level2(level3)(level3)))\end{cps}` which results in:
(level1(level2(level3)(level3))). This feature is automtically enabled in some other env. like the *semcalc* framework. To change the colors one needs to manually override the colors *fglcpcolor0* to *fglcpcolor5* like this:
`\definecolor{fglcpcolor0}{RGB}{000, 000, 000}`.

The core component of this package is the *fgls* environment, which provides abbreviated versions of all commands and is designed for constructing larger semantic formulae and derivations. Given the extensive set of custom commands—ranging from one to four letters—potential conflicts with other packages may arise, so careful usage within other environments is advised. For example, the minimal syntax facilitated by this environment is illustrated below:

```
\begin{fgls}[\sone]
  \e{sees} = \ly\lx\n{\Mx sees \My in \cs}
\end{fgls}
```

which renders as:
$[\![\textbf{sees}]\!]^{s_1} = \lambda y.\lambda x.\vdash x$ sees y in $s_1 \dashv$. The advantage of this syntax lies in its ability to maintain consistent formatting of variables across various levels of nesting. Previously, manually handling the font styles of variables introduced the risk of inconsistencies—such as losing the correct formatting (e.g., boldface for object-language variables) when copying LaTeX code into new formulae. This issue is eliminated by using commands like `\mx`, which consistently produces an italic, non-bold x for meta-language variables. Additionally, the use of `\n{exp...}` ensures structural integrity, as failing to close the expression correctly now results in a compiler error, preventing unintended formatting issues. Also the optional parameter `[\sone]` automatically sets the current situation for all commands inside the local *fgls* env., which makes changes easy.

All of this is designed to make writing large semantic formulae faster and more error pronone and to provide beautiful and consistent output.

2 Examples

To construct a complex semantic derivation, use the *semderivation* environment and introduce each derivation step with the `\ds` command. Within this environment, both the *cps* and *fgls* environments are enabled, allowing for consistent formatting and structured derivations. The *semderivation* environment internally initializes an *itemize* environment, requiring each derivation step to be formatted as an item, which is automatically handled when using the `\ds` command. The first step of any derivation should always use the `\dS` command to omit the initial equal sign for proper formatting.
A local situation can be set within the environment using
`\begin{semderivation}[situation]derivation...\end{semderivation}`, overriding all default situations in commands that include one. Additionally, a custom reference name can be assigned using
`\begin{semderivation}[situation][refName]derivation...\end{semderivation}`.
This reference name allows for automatic referencing of each derivation step, making it possible to cite specific steps within the text. By default, references are numbered sequentially starting from 1. However, a fixed reference name can be specified for consistency across citations.

To reference a specific derivation step, use:
`\ref{sem:deri:refName:ds:dsCount}`

Here, `refName` refers either to the automatically assigned sequential derivation number or a user-defined reference name, while `dsCount` denotes the index of the derivation step, beginning at 1.

Example:

```
\begin{semderivation}[\szero][exampleone]
  \dS{\e{Peter loves Maria}}
  \ds{\e{loves Maria}(\e{Peter})}
  \ds{\e{loves Maria}(Peter)}
  \ds{\e{loves}(\e{Maria})(Peter)}
  \ds{\e{loves}(Maria)(Peter)}
  \ds{\ly\lx\n{\Mx loves \My in \cs}(Maria)(Peter)}
  \ds{\lx\n{\Mx loves Maria in \cs}(Peter)}
```

```
  \ds{\n{Peter loves Maria in \cs}}
\end{semderivation}
```

Output:

$[\![\textbf{Peter loves Maria}]\!]^{s_0}$

2. $= [\![\textbf{loves Maria}]\!]^{s_0}([\![\textbf{Peter}]\!]^{s_0})$
3. $= [\![\textbf{loves Maria}]\!]^{s_0}(\text{Peter})$
4. $= [\![\textbf{loves}]\!]^{s_0}([\![\textbf{Maria}]\!]^{s_0})(\text{Peter})$
5. $= [\![\textbf{loves}]\!]^{s_0}(\text{Maria})(\text{Peter})$
6. $= \lambda y.\lambda x.\vdash x \text{ loves } y \text{ in } s_0\dashv(\text{Maria})(\text{Peter})$
7. $= \lambda x.\vdash x \text{ loves Maria in } s_0\dashv(\text{Peter})$
8. $= \vdash \text{Peter loves Maria in } s_0\dashv$

Note that the default situation has been changed from $s^*$to s_0. To ensure consistency within a derivation, it is recommended to specify the desired situation as an environment parameter and reference it within the derivation using the `\cs` command, which represents the current situation. To reference a specific derivation step, use: `\ref{sem:deri:exampleone:ds:3}` which will produce: Step 3.
A complete semantic derivation may include not only derivation steps but also trees, rules, and lexical entries. In such cases, use the *semcalc* environment, which is specifically designed for educational purposes. Each *semcalc* environment should follow a standardized structure, though not all components are mandatory:

```
\begin{semcalc}
  \begin{semtree}
    tree
  \end{semtree}
  \begin{semrule}
    rules...
  \end{semrule}
  \begin{semlex}
    lexicon...
  \end{semlex}
  \begin{semderi}
    derivation...
  \end{semderi}
\end{semcalc}
```

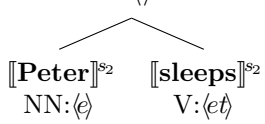
All these sub-environments function exclusively within the *semcalc* or *fgls* environments. A detailed explanation of each sub-environment is provided in the section 5.6.
Example for a full semantic calculation for: $[\![\textbf{Peter sleeps}]\!]^{s_2}$

```
\begin{semcalc}[\stwo]
  \begin{semtreesem}
    \begin{forest}
    [{\e{Peter sleeps}\\S:\typet},name=0
    [{\e{Peter}\\NN:\typee},name=00]
    [{\e{sleeps}\\V:\typeet},name=01]
    ]
    \end{forest}
  \end{semtreesem}
  \begin{semlang}
    \rlang{Peter, sleeps}[NN][VP]
  \end{semlang}
  \begin{semrule}
    \rsrdesbjpred
    \rentrysub{\j{Compositional Determination of the Extension of Subject-Predications
    }\\If \j{S} is a sentence with a predicate \j{P} and a proper name \j{NN} as its
    subject, then for all \j{s\ \ci\ LR} the above holds.}
  \end{semrule}
  \begin{semlex}
    \sldenn{Peter}
```

```
    \sldex{sleeps}
  \end{semlex}
  \begin{semderi}
    \dS{\e{Peter sleeps}}
    \ds[\rr{1}]{\e{sleeps}(\e{Peter})}
    \ds[\rl{1}]{\e{sleeps}(Peter)}
    \ds[\rl{2}]{\lfx{sleeps}(Peter)}
    \ds[\lc]{\n{Peter sleeps in \cs}}
  \end{semderi}
\end{semcalc}
```

The code above will result in the following derivation structure:

(i) *Tree (semantic)*: $[\![\textbf{Peter sleeps}]\!]^{s_2}$



(ii) *Language*:

$$L = L_{NN} \cup L_{VP} = \{\text{Peter, sleeps}\}$$

(iii) *Rules*:

R1: $[\![S]\!]^s = [\![P]\!]^s([\![NN]\!]^s)$

Compositional Determination of the Extension of Subject-Predications
If S is a sentence with a predicate P and a proper name NN as its subject, then for all $s \in LR$ the above holds.

(iv) *Lexicon*:

L1: $[\![\textbf{Peter}]\!]^{s_2} = \text{Peter}$

L2: $[\![\textbf{sleeps}]\!]^{s_2} = \lambda x_{\langle e \rangle}.\vdash x \text{ sleeps in } s_2 \dashv$

(v) *Derivation*:

$[\![\textbf{Peter sleeps}]\!]^{s_2}$

$\overset{2.}{=}$ $[\![\textbf{sleeps}]\!]^{s_2}([\![\textbf{Peter}]\!]^{s_2})$ R1

$\overset{3.}{=}$ $[\![\textbf{sleeps}]\!]^{s_2}(\text{Peter})$ L1

$\overset{4.}{=}$ $\lambda x.\vdash x \text{ sleeps in } s_2 \dashv(\text{Peter})$ L2

$\overset{5.}{=}$ $\vdash \text{Peter sleeps in } s_2 \dashv$ λ-conv.

As shown on the right, each step references the corresponding rule or lexical entry to indicate to the reader which resource was used to reach the next step. These resources can also be referenced outside of the *semderivation* environment using the command `\ref{sem:deri:refNameresCoderesNumber}`, where `refName` refers to the derivation reference name (as explained earlier), `resNumber` is the index of the desired resource (starting from 1), and `resCode` specifies the kind of resource. There are four types of resource codes: `r`, `rs`, `l`, and `la`, which stand for resource, resource-syntax, lexicon, and language, respectively. For further details, see Command 120.
An additional example of a more complex derivation involving indirect interpretation would look like this:

```
\begin{semderivation}
  \dS{\e[g,\!\cs]{\r[S]{\r[DP]{jeden Esel}\r[S]{\lx\r[S]{Hans \r[VP]{\txs f\"{u}tterte
  }}}}}}
  \ds[\fa]{\e{jeden Esel}(\e[g,\!\cs]{\lx\r[S]{Hans \r[VP]{\txs f\"{u}tterte}}})}
  \ds[\abs]{\e{jeden Esel}(\lz\e[g{\asmoD{z}[x]},\!\cs]{Hans \r[VP]{\txs f\"{u}tterte
  }})}
  \ds[\fa]{\e{jeden Esel}(\lz\r{\e[g{\asmoD{z}[x]},\!\cs]{\r[VP]{\txs f\"{u}tterte}}(\e
  [g{\asmoD{z}[x]},\!\cs]{Hans})})}
  \ds[\fa]{\e{jeden Esel}(\lz\r{\e[g{\asmoD{z}[x]},\!\cs]{f\"{u}tterte}(\e[g{\asmoD{z}[
  x]},\cs]{\tx})(\e[g{\asmoD{z}[x]},\!\cs]{Hans})})}
  \ds[SemVar]{\e{jeden Esel}(\lz\r{\e[g{\asmoD{z}[x]},\!\cs]{f\"{u}tterte}(\mz)(\e[g{\
  asmoD{z}[x]},\!\cs]{Hans})})}
```

```
  \ds[\lex]{\e{jeden Esel}(\lz\r{\e[g{\asmoD{z}[x]},\!\cs]{f\"{u}tterte}(\mz)(Hans)})}
  \ds[\lex]{\e{jeden Esel}(\lz\r{\ly\lx\n{\mx\ f\"{u}tterte \my\ in \cs}(\mz)(Hans)})}
  \ds[\lc]{\e{jeden Esel}(\lz\r{\lx\n{\mx\ f\"{u}tterte \mz\ in \cs}(Hans)})}
  \ds[\lc]{\e{jeden Esel}(\lz\n{Hans f\"{u}tterte \mz\ in \cs})}
  \ds[\lex]{\lX\n{\sa\e{Esel}\Cq\sa\mX}(\lz\n{Hans f\"{u}tterte \mz\ in \cs})}
  \ds[\lc]{\n{\sa\e{Esel}\Cq\sA{\lz\n{Hans f\"{u}tterte \mz\ in \cs}}}}
  \ds[\lex]{\n{\sA{\lx\n{\mx\ ist ein Esel in \cs}}\Cq\sA{\lz\n{Hans f\"{u}tterte \mz\
  in \cs}}}}
  \ds[Def. \cd]{\n{\s{\Sv ist ein Esel in \cs}\Cq\s{Hans f\"{u}tterte \Sv in \cs}}}
  \ds[\nc]{\n{\cset{Esel}\Cq\s{Hans f\"{u}tterte \Sv in \cs}}}
  \ds{1 gdw. alle Individuen, die Esel sind in \cs\ auch Individuen sind, die von Hans
  in \cs\ gef\"{u}ttert werden, sonst 0.}
\end{semderivation}
```

which would result in a derivation that looks like this (derivation for *Hans fed every donkey* with quantifier raising):

$[\![[_S\ [_{DP}\ \textbf{jeden Esel}][_S\ \lambda\boldsymbol{x}.[_S\ \textbf{Hans}\ [_{VP}\ t_x\ \textbf{fütterte}]]]]]\!]^{g,s^*}$

2. $=$ $[\![\textbf{jeden Esel}]\!]^{s^*}([\![\lambda\boldsymbol{x}.[_S\ \textbf{Hans}\ [_{VP}\ t_x\ \textbf{fütterte}]]]\!]^{g,s^*})$ App
3. $=$ $[\![\textbf{jeden Esel}]\!]^{s^*}(\lambda z.[\![\textbf{Hans}\ [_{VP}\ t_x\ \textbf{fütterte}]]\!]^{g[x/z],s^*})$ (s)
4. $=$ $[\![\textbf{jeden Esel}]\!]^{s^*}(\lambda z.[[\![_{VP}\ t_x\ \textbf{fütterte}]\!]^{g[x/z],s^*}([\![\textbf{Hans}]\!]^{g[x/z],s^*})])$ App
5. $=$ $[\![\textbf{jeden Esel}]\!]^{s^*}(\lambda z.[[\![\textbf{fütterte}]\!]^{g[x/z],s^*}([\![t_x]\!]^{g[x/z],s^*})([\![\textbf{Hans}]\!]^{g[x/z],s^*})])$ App
6. $=$ $[\![\textbf{jeden Esel}]\!]^{s^*}(\lambda z.[[\![\textbf{fütterte}]\!]^{g[x/z],s^*}(z)([\![\textbf{Hans}]\!]^{g[x/z],s^*})])$ SemVar
7. $=$ $[\![\textbf{jeden Esel}]\!]^{s^*}(\lambda z.[[\![\textbf{fütterte}]\!]^{g[x/z],s^*}(z)(\text{Hans})])$ Lex
8. $=$ $[\![\textbf{jeden Esel}]\!]^{s^*}(\lambda z.[\lambda y.\lambda x.\vdash x\ \text{fütterte}\ y\ \text{in}\ s^*\dashv(z)(\text{Hans})])$ Lex
9. $=$ $[\![\textbf{jeden Esel}]\!]^{s^*}(\lambda z.[\lambda x.\vdash x\ \text{fütterte}\ z\ \text{in}\ s^*\dashv(\text{Hans})])$ λ-conv.
10. $=$ $[\![\textbf{jeden Esel}]\!]^{s^*}(\lambda z.\vdash\text{Hans fütterte}\ z\ \text{in}\ s^*\dashv)$ λ-conv.
11. $=$ $\lambda X.\vdash\downarrow[\![\textbf{Esel}]\!]^{s^*}\subseteq\downarrow X\dashv(\lambda z.\vdash\text{Hans fütterte}\ z\ \text{in}\ s^*\dashv)$ Lex
12. $=$ $\vdash\downarrow[\![\textbf{Esel}]\!]^{s^*}\subseteq\downarrow[\lambda z.\vdash\text{Hans fütterte}\ z\ \text{in}\ s^*\dashv]\dashv$ λ-conv.
13. $=$ $\vdash\downarrow[\lambda x.\vdash x\ \text{ist ein Esel in}\ s^*\dashv]\subseteq\downarrow[\lambda z.\vdash\text{Hans fütterte}\ z\ \text{in}\ s^*\dashv]\dashv$ Lex
14. $=$ $\vdash\{x : x\ \text{ist ein Esel in}\ s^*\}\subseteq\{x : \text{Hans fütterte}\ x\ \text{in}\ s^*\}\dashv$ Def. ↓
15. $=$ $\vdash \mathit{Esel}_{s^*}\subseteq\{x : \text{Hans fütterte}\ x\ \text{in}\ s^*\}\dashv$ not. conv.
16. $=$ 1 gdw. alle Individuen, die Esel sind in s^* auch Individuen sind, die von Hans in s^* gefüttert werden, sonst 0.

Note that the comments on the right should signal the reader which rule or lexical entry or notational convention has been used to get to the next step.

If you want to use all the shortcuts provided by the *fgls* environment for a formulae thats rather short or inside a bigger text, use the *fglsem* wrapper command like `\fglsem{\t{Expression}}` which will result in || without any automatic line breaks or indentation.

3 Usage

To use the ‘fragoli’ package place the ‘fragoli.sty’ file in the same folder as your document and include:

```
\usepackage{fragoli}
```

3.1 Package Options

To combine multiple package options use ‘, ’ like:

```
\usepackage[userainbow=true, language=german]{fragoli}
```

3.1.1 language

```
\usepackage[language=LANGUAGE]{fragoli}
```

This option will effect the default lexicon and rule entries as well as the labels for a semantic derivation. The default ist set to **'english'**, valid options are:

- english (e.g. R1: $|| = \boldsymbol{\lambda Q.(\lambda P.\neg(\exists x_{\langle e \rangle})[Q(x) \wedge P(x)])}$)
- german (e.g. R1: $|| = \boldsymbol{\lambda Q.(\lambda P.\neg(\exists x_{\langle e \rangle})[Q(x) \wedge P(x)])}$)

3.1.2 userainbow

```
\usepackage[userainbow=BOOL]{fragoli}
```

This option will effect the automatic coloring of parentheses levels within a semantic derivation or the *cps* environment. The default is set to **'true'**, valid options are true/false. To change the colors one needs to manually override the color values *fglcpcolor0* to *fglcpcolor5* like this: `\definecolor{fglcpcolor0}{RGB}{000, 000, 000}`.

- true (e.g. (level0(level1((level3(level4(level5(level6))))level2))))
- false (e.g. (level0(level1((level3(level4(level5(level6))))level2))))

3.1.3 usetypes

```
\usepackage[usetypes=BOOL]{fragoli}
```

This option will effect if the default types in lexicon and rule entries are displayed. The default ist set to **'false'**, valid options are true/false.

- true (e.g. R1: $|| = \boldsymbol{\lambda Q_{\langle et \rangle}.(\lambda P_{\langle et \rangle}.\neg(\exists x_{\langle e \rangle})[Q(x) \wedge P(x)])}$)
- false (e.g. R1: $|| = \boldsymbol{\lambda Q.(\lambda P.\neg(\exists x)[Q(x) \wedge P(x)])}$)

3.1.4 typestyle

```
\usepackage[typestyle=STYLE]{fragoli}
```

This option will effect the default parentheses style of all types added with the `\type{#1}` command. The default ist set to **'normal'**, valid options are:

- normal (e.g. $\langle et \rangle$)
- classic (e.g. (et))
- alternative (e.g. $<et>$)

To change/reset the type style mid document to a custom symbol use:

```
\fglsetparleft{symbol}
\fglsetparright{symbol}
\fglresetparleft
\fglresetparright
```

3.1.5 typesspacing

```
\usepackage[typestyle=STYLE]{fragoli}
```

This option will effect the default parentheses spacing of all types added with the `\type{#1}` command. The default ist set to **'narrow'**, valid options are:

- narrow (e.g. $\langle e\langle et\rangle\rangle$) negthinspace and kern-1pt
- mid (e.g. $\langle e\langle et\rangle\rangle$) negthinspace and default space
- wide (e.g. $\langle e\langle et\rangle\rangle$) default space

To change/reset the spacing (any custom value possible) mid document use:

```
\fglsettypespaceinner{spacing}
\fglsettypespaceouter{spacing}
\fglresettypespaceinner
\fglresettypespaceouter
```

3.1.6 typenestingstyle

```
\usepackage[typenestingstyle=STYLE]{fragoli}
```

This option will effect the default parentheses nesting style of all types added with the `\type{#1}` command. The default ist set to `'normal'`, valid options are:

- normal (e.g. $\langle e\langle et\rangle\rangle$, here the first level also has parentheses)
- classic (e.g. $e\langle et\rangle$, here only the nested levels have parentheses)

3.1.7 lambdaheadstyle

```
\usepackage[lambdaheadstyle=STYLE]{fragoli}
```

This option will effect the default type positioning/style of all lambda heads when adding a type as argument. The default ist set to `'normal'`, valid options are:

- normal (e.g. $\lambda \boldsymbol{Q}_{\langle et\rangle}.$)
- domain (e.g. $\lambda \boldsymbol{Q} \in \mathrm{D}_{\langle et\rangle}.$)
- upper (e.g. $\lambda \boldsymbol{Q}^{\langle et\rangle}.$)

3.1.8 lambdaheadspacing

```
\usepackage[lambdaheadspacing=STYLE]{fragoli}
```

This option will effect the default spacing of all lambda heads (effecting the dot and the optional types). The default ist set to `'narrow'`, valid options are:

- narrow (e.g. $\lambda x_{\langle e\rangle}.\lambda y_{\langle e\rangle}.$) negthinspace
- mid (e.g. $\lambda x_{\langle e\rangle}.\lambda y_{\langle e\rangle}.$) kern-1pt
- wide (e.g. $\lambda x_{\langle e\rangle}.\lambda y_{\langle e\rangle}.$) default space

To change/reset the spacing (any custom value possible) mid document use:

```
\fglsetlambdaheadspacing{spacing}
\fglresetlambdaheadspacing
```

3.1.9 usedscount

```
\usepackage[usedscount=BOOL]{fragoli}
```

This option will effect if the row count will be displayed in every semantic derivation step in the *semderivation* and *semderi* environments. The default ist set to `'true'`, valid options are true/false. To change the color one needs to manually override the color value like this:
`\definecolor{fgldeivationrowcountcolor}{RGB}{000, 000, 000}`.

true:

$$\begin{aligned} & \lambda x.[x{+}2](2) \\ \overset{2.}{=}\ & 2{+}2 \\ \overset{3.}{=}\ & 4 \end{aligned}$$

false:

$$\begin{aligned} & \lambda x.[x{+}2](2) \\ =\ & 2{+}2 \\ =\ & 4 \end{aligned}$$

3.1.10 showawnser

```
\usepackage[showawnser=BOOL]{fragoli}
```

This option will effect the question and awnser environments. If set to true everything written within a question environment will be colored light grey and look like this: example text and everything written within an awnser environment will be visible. If set to false every awnser environment will be hidden and the question env. content will be colored black. To change the color one needs to manually override the color value like this: `\definecolor{fglquestioncolor}{RGB}{000, 000, 000}`.

3.1.11 rulesituation

```
\usepackage[rulesituation=SITUATION]{fragoli}
```

This option will effect every predefined semantic rule resource (NOT the lexical entries or the default situation within the fgls env.). By default the situation displayed will be s. Note that every predefined rule can be customized individually again.

3.1.12 situation

```
\usepackage[situation=SITUATION]{fragoli}
```

This option will effect every predefined situation (EXCEPT for all the predefined rules). By default the situation displayed will be s^*. Note that every predefined situation can be customized individually again. This will effect for example every command that contains a situation and also every shortcut in the *fgls* environment, also every predefined lexical entry and all other occurences of situations.

3.1.13 situationspacing

```
\usepackage[situationspacing=STYLE]{fragoli}
```

This option will effect the default spacing of all situations. The default ist set to **'narrow'**, valid options are:

- narrow (e.g. s_1) kern-0.8pt
- mid (e.g. s_1) kern-0.3pt
- wide (e.g. s_1) default space

To change/reset the spacing (any custom value possible) mid document use:

```
\fglsetsituationspacing{spacing}
\fglresetsituationspacing
```

3.1.14 tracespacing

```
\usepackage[tracespacing=STYLE]{fragoli}
```

This option will effect the default spacing of all traces. The default ist set to **'mid'**, valid options are:

- narrow (e.g. t_x) kern-0.8pt
- mid (e.g. t_x) kern-0.3pt
- wide (e.g. t_x) default space

To change/reset the spacing (any custom value possible) mid document use:

```
\fglsettracespacing{spacing}
\fglresettracespacing
```

3.1.15 semanticsuperscriptspacing

```
\usepackage[semanticsuperscriptspacing=STYLE]{fragoli}
```

This option will effect the default spacing superscripts for direct and indirect bracketing. The default ist set to **'narrow'**, valid options are:

- narrow (e.g. $[\![\mathbf{Peter}]\!]^{s^*}$) negthinspace
- mid (e.g. $[\![\mathbf{Peter}]\!]^{s^*}$) kern-1pt
- wide (e.g. $[\![\mathbf{Peter}]\!]^{s^*}$) default space

To change/reset the spacing (any custom value possible) mid document use:

```
\fglsetsuperscriptspacing{spacing}
\fglresetsuperscriptspacing
```

3.1.16 assignmentspacing

```
\usepackage[assignmentspacing=STYLE]{fragoli}
```

This option will effect the default spacing of all assignment functions. The default ist set to **'mid'**, valid options are:

- narrow (e.g. $\|\mathbf{Peter}\|^{g[x/r]}$) negthinspace
- mid (e.g. $\|\mathbf{Peter}\|^{g[x/r]}$) kern-1pt
- wide (e.g. $\|\mathbf{Peter}\|^{g[x/r]}$) default space

To change/reset the spacing (any custom value possible) mid document use:

```
\fglsetassignmentspacing{spacing}
\fglresetassignmentspacing
```

3.1.17 alternativecommands

```
\usepackage[alternativecommands=BOOL]{fragoli}
```

This option enables a set of alternative commands for backwards compatibility within the institute. This is only needed to support legacy code (e.g. Typ, Bsp, Tran, char, WW etc.), debug, and for institute internal usage.

4 Commands

Most commands in this package have two versions, one normal and one bold version. For the bold version just write the last letter in uppercase (except for a few commands).

4.1 Text

This section lists commands modifying the appearance of text.

(1)

```
1 \oo{text}[height]
```

Arguments:

`text`: Text to overline
`height`(def=-2ex): Set the overlining height

Description: Double overlining (that works with line breaks)

Example: $\overline{\overline{\text{Test}}}$

(2)

```
1 \oO{text}
```

Arguments:

`text`: Text to overline

Description: Double overlining (that works with line breaks) with more space (-2.5ex)

Example: $\overline{\overline{\text{Test}}}$

4.1.1 Object Language

Marks text as object language in opposition to meta language.

(3)

```
\obl{text}
```

Arguments:

`text`: Text to format

Description: Marking text as object language

Example: **Test**

(4)

```
\obli{text}
```

Arguments:

`text`: Text to format

Description: Marking text as object language italic

Example: ***Test***

(5)

```
\obla{arg}[arg2][arg3][arg4][arg5][arg6][arg7][arg8][arg9]
```

Arguments:

`arg`: First argument
`argN`: nth argument

Description: Marking text as object language arguments with up to 8 optional arguments in normal parentheses behind the first argument

Example: (**arg**)(**optional1**)(**optional2**)**...**

(6)

```
\oblia{arg}[arg2][arg3][arg4][arg5][arg6][arg7][arg8][arg9]
```

Arguments:

`arg`: First argument
`argN`: nth argument

Description: Marking text as object language italic arguments with up to 8 optional arguments in normal parentheses behind the first argument

Example: **(*arg*)(*optional1*)(*optional2*)...**

4.1.2 Meta Language

Marks text as meta language in opposition to object language. Used to ensure text is not formatted in a chain of commands.

(7)

```
\mel{text}
```

Arguments:

`text`: Text to format

Description: Marking text as meta language

Example: Test

(8)

```
\meli{text}
```

Arguments:

`text`: Text to format

Description: Marking text as meta language italic

Example: *Test*

(9)

```
\mela{arg}[arg2][arg3][arg4][arg5][arg6][arg7][arg8][arg9]
```

Arguments:

`arg`: First argument
`argN`: nth argument

Description: Marking text as meta language arguments with up to 8 optional arguments in normal parentheses behind the first argument
Bold Option: Yes

Example: (arg)(optional1)(optional2)...

(10)

```
\melia{arg}[arg2][arg3][arg4][arg5][arg6][arg7][arg8][arg9]
```

Arguments:

`arg`: First argument
`argN`: nth argument

Description: Marking text as meta language italic arguments with up to 8 optional arguments in normal parentheses behind the first argument
Bold Option: Yes

Example: **(***arg***)(***optional1***)(***optional2***)**...

4.2 Bracketing

This section lists general text bracketing commands.

(11)

```
\set[var]{properties}[name]
```

Arguments:

`var`(def=x): Varibale name and/or domain specifier
`properties`: Set-builder notation
`var`(def=): The name of the set

Description: Creating an inensional build set. Each use of this command will override the `\setvar` command. This command should be used inside the set properties to reference the set variable. The reference will always cut the content of the optional *var* argument before the first space character. Use `\Setvar` to automatically add a space behind the command. To change the style of the set from : to e.g. | replace the `\@fglsetspacer` constant using `\newcommand{\@fglsetspacer}{$|$}`.
Bold Option: Yes

Example: `\set[n \in \mathbb{N}]{\Setvar is even}[A]` will result in: $A = \{n \in \mathbb{N} : n \text{ is even}\}$

(12)

```
\bool{text}
```

Arguments:

`text`: Text inside truth brackets

Description: Creating truth conditional bracketing.
Bold Option: Yes

Example: $\vdash$it rains in $s^* \dashv$

(13)

```
\lambby[subInner][subOuter]{text}
```

Arguments:

> `subInner`: Optional subscript behind opening bracket. Text will be in mathmode. (mostly used in tree like structures)
> `subOuter`: Optional subscript behind closing bracket. Text will be in mathmode (mostly used to identify types)
> `text`: Text inside rectangle brackets

Description: Creating rectangle brackets, mainly used for a lambda function body.
Bold Option: Yes
Example: [P $\wedge$ Q]

Example: $[_{Inner}$ P $\wedge$ Q$]_{Outer}$

(14)

```
\zit{text}
```

Arguments:

> `text`: Text inside double quotation marks

Description: Creating double quotation marks around passed text.
Bold Option: No

Example: “Autounfall”

(15)

```
\her{text}
```

Arguments:

> `text`: Text inside single quotation marks

Description: Creating single quotation marks around passed text.
Bold Option: No

Example: ‘Autounfall’

4.3 Logic

This section covers commands for logic symbols modified for semantic purposes.

(16)

```
\sneg
```

Description: Creating a negation symbol.
Bold Option: Yes

Example: $\neg$

(17)

```
\strue
```

Description: Creating a verum symbol.
Bold Option: Yes
Since: 1.1.1

Example: $\top$

(18)

```
\sfalse
```

Description: Creating a falsum symbol.
Bold Option: Yes
Since: 1.1.1

Example: $\bot$

(19)

```
\snec
```

Description: Creating a necessity symbol.
Bold Option: Yes
Since: 1.1.1

Example: $\Box$

(20)

```
\sposs
```

Description: Creating a possible symbol.
Bold Option: Yes
Since: 1.1.1

Example: $\Diamond$

(21)

```
\simp{arg1}{arg2}
```

Arguments:

- `arg1`: Antecedent.
- `arg2`: Consequent.

Description: Creating a material implication.
Bold Option: Yes

Example: a $\rightarrow$ b

(22)

```
\simpnc{arg1}{arg2}
```

Arguments:

- `arg1`: Antecedent.
- `arg2`: Consequent.

Description: Creating a material implication but without the notational convention.
Bold Option: Yes

Example: $\rightarrow$(a)(b)

(23)

```
\sor{arg1}{arg2}[arg3][arg4][arg5][arg6][arg7][arg8][arg9]
```

Arguments:

- `arg1`: 1st disjunct
- `arg2`: 2nd disjunct
- `argN`: nth disjunct

Description: Creating a disjunctive formula with min. 2 disjuncts.
Bold Option: Yes

Example: a ∨ b ∨ c

(24)

```
\sornc{arg1}{arg2}[arg3][arg4][arg5][arg6][arg7][arg8][arg9]
```

Arguments:

- `arg1`: 1st disjunct
- `arg2`: 2nd disjunct
- `argN`: nth disjunct

Description: Creating a disjunctive formula with min. 2 disjuncts but without the notational convention.
Bold Option: Yes

Example: ∨(a)(b)(c)

(25)

```
\sxor{arg1}{arg2}[arg3][arg4][arg5][arg6][arg7][arg8][arg9]
```

Arguments:

- `arg1`: 1st disjunct
- `arg2`: 2nd disjunct
- `argN`: nth disjunct

Description: Creating an exclusive disjunctive formula with min. 2 disjuncts.
Bold Option: Yes
Since: 1.1.1

Example: a ⊻ b ⊻ c

(26)

```
\sxornc{arg1}{arg2}[arg3][arg4][arg5][arg6][arg7][arg8][arg9]
```

Arguments:

- `arg1`: 1st disjunct
- `arg2`: 2nd disjunct
- `argN`: nth disjunct

Description: Creating an exclusive disjunctive formula with min. 2 disjuncts but without the notational convention.
Bold Option: Yes
Since: 1.1.1

Example: ⊻(a)(b)(c)

(27)

```
\sand{arg1}{arg2}[arg3][arg4][arg5][arg6][arg7][arg8][arg9]
```

Arguments:

- `arg1`: 1st conjunct
- `arg2`: 2nd conjunct
- `argN`: nth conjunct

Description: Creating a conjunctive formula with min. 2 conjuncts.
Bold Option: Yes

Example: a ∧ b ∧ c

(28)

```
\sandnc{arg1}{arg2}[arg3][arg4][arg5][arg6][arg7][arg8][arg9]
```

Arguments:

- `arg1`: 1st conjunct
- `arg2`: 2nd conjunct

`argN`: nth conjunct

Description: Creating a conjunctive formula with min. 2 conjuncts but without the notational convention.
Bold Option: Yes

Example: ∧(a)(b)(c)

(29)

```
\slogic{symbol}{arg1}{arg2}[arg3][arg4][arg5][arg6][arg7][arg8]
```

Arguments:

`symbol`: the junctor symbol (math mode is enforced)
`arg1`: 1st conjunct
`arg2`: 2nd conjunct
`argN`: nth conjunct

Description: Creating a blank formula with min. 2 arguments.
Bold Option: Yes
Since: 1.1.1

Example: `\slogic{\oplus}{a}{b}` will result in: a ⊕ b

(30)

```
\slogicnc{symbol}{arg1}{arg2}[arg3][arg4][arg5][arg6][arg7][arg8]
```

Arguments:

`symbol`: the junctor symbol (math mode is enforced)
`arg1`: 1st conjunct
`arg2`: 2nd conjunct
`argN`: nth conjunct

Description: Creating a blank formula with min. 2 arguments but without the general notational convention.
Bold Option: Yes
Since: 1.1.1

Example: `\slogicnc{\oplus}{a}{b}}` will result in: ⊕(a)(b)

4.4 Lambda

This section covers commands for creating lambda functions.

(31)

```
\lambfx{body}[headVar][headType]
```

Arguments:

`body`: lambda function body
`headVar(def=x)`: lambda head variable
`headType`: head variable type

Description: Creats a full lambda function with changeable head variable and optional head type.
Bold Option: Yes

Example: $\lambda x.\vdash x$ sleeps in $s^* \dashv$

(32)

```
\lambfyx{body}[headVar1][headVar2][headType1][headType2]
```

Arguments:

`body`: lambda function body
`headVar1(def=y)`: first lambda head variable
`headVar2(def=x)`: second lambda head variable
`headType1`: first head variable type
`headType2`: second head variable type

Description: Creats a full lambda function with two changeable head variable and optional head type.
Bold Option: Yes

Example: $\lambda y.\lambda x.\vdash x$ kills y in $s^*\dashv$

(33)

```
\lambfzyx{body}[headVar1][headVar2][headVar3][headType1][headType2][headType3]
```

Arguments:

`body`: lambda function body
`headVar1(def=z)`: first lambda head variable
`headVar2(def=y)`: second lambda head variable
`headVar3(def=x)`: third lambda head variable
`headType1`: first head variable type
`headType2`: second head variable type
`headType3`: third head variable type

Description: Creats a full lambda function with three changeable head variable and optional head type.
Bold Option: Yes

Example: $\lambda z.\lambda y.\lambda x.\vdash x$ gives y z in $s^*\dashv$

(34)

```
\lambfsx{body}[headVar][situationVar][headType][situationType]
```

Arguments:

`body`: lambda function body
`headVar(def=x)`: lambda head variable
`situationVar(def=s)`: lambda head variable for the situation
`headType`: head variable type
`situationType`: situation type (should be used if you want to display the type)

Description: Creats a full lambda function with changeable head variable and optional head type.
Bold Option: Yes

Example: $\lambda s.\lambda x.\vdash x$ sleeps in $s\dashv$

(35)

```
\lambfsyx{body}[headVar1][headVar2][situationVar][headType1][headType2][
situationType]
```

Arguments:

`body`: lambda function body
`headVar1(def=y)`: first lambda head variable
`headVar2(def=x)`: second lambda head variable
`situationVar(def=s)`: lambda head variable for the situation
`headType1`: first head variable type
`headType2`: second head variable type
`situationType`: situation type (should be used if you want to display the type)

Description: Creats a full lambda function with two changeable head variable and optional head type.
Bold Option: Yes

Example: $\lambda s.\lambda y.\lambda x.\vdash x$ kills y in $s\dashv$

(36)

```
\lambfszyx{body}[headVar1][headVar2][headVar3][situationVar][headType1][headType
2][headType3][situationType]
```

Arguments:

`body`: lambda function body
`headVar1(def=z)`: first lambda head variable
`headVar2(def=y)`: second lambda head variable
`headVar3(def=x)`: third lambda head variable
`situationVar(def=s)`: lambda head variable for the situation
`headType1`: first head variable type
`headType2`: second head variable type
`headType3`: third head variable type
`situationType`: situation type (should be used if you want to display the type)

Description: Creats a full lambda function with three changeable head variable and optional head type.
Bold Option: Yes

Example: $\lambda s.\lambda z.\lambda y.\lambda x.\vdash x$ gives $y\ z$ in $s\dashv$

(37)

```
\lambh{var}[type][spacing]
```

Arguments:

`arg`: head argument
`type`: argument type
`spacing(def=negthinspace)`: spacing between dots and optional type

Description: Creating a head for an anonymous function with an optional type.
Bold Option: Yes
Example: $\lambda x_{\langle e \rangle}$.

Example: $\boldsymbol{\lambda x}_{\langle e \rangle}$**.**

(38)

```
\lambhe[var][type]
```

Arguments:

`var`: head argument
`type`: argument type

Description: Creating an existential head for an anonymous function with an optional type.
Bold Option: Yes

Example: $(\exists x_{\langle e \rangle})$

(39)

```
\lambhen[var][type]
```

Arguments:

`var`: head argument
`type`: argument type

Description: Creating a negated existential head for an anonymous function with an optional type.
Bold Option: Yes

Example: $\neg(\exists x_{\langle e \rangle})$

(40)

```
\lambhu[var][type]
```

Arguments:

`var`: head argument
`type`: argument type

Description: Creating a uniqueness head for an anonymous function with an optional type.
Bold Option: Yes
Since: 1.1.1

Example: $(\exists! x_{\langle e \rangle})$

(41)

```
\lambhun[var][type]
```

Arguments:

`var`: head argument
`type`: argument type

Description: Creating a negated uniqueness head for an anonymous function with an optional type.
Bold Option: Yes
Since: 1.1.1

Example: $\neg(\exists!x_{\langle e\rangle})$

(42)

```
\lambha[var][type]
```

Arguments:

`var`: head argument
`type`: argument type

Description: Creating a universal head for an anonymous function with an optional type.
Bold Option: Yes

Example: $(\forall x_{\langle e\rangle})$

(43)

```
\lambhan[var][type]
```

Arguments:

`var`: head argument
`type`: argument type

Description: Creating a negated universal head for an anonymous function with an optional type.
Bold Option: Yes

Example: $\neg(\forall x_{\langle e\rangle})$

4.5 Semantics

General semantics commands.

(44)

```
\type[innerSpacing][outerSpacing]{type}[leftPar][rightPar]
```

Arguments:

`innerSpacing(def=negthinspace)`: spacing between type text and the left and right parentheses
`outerSpacing(def=-1pt)`: spacing between the parentheses
`type`: the type (e.g. et)
`leftPar(def=⟨)`: left parentheses
`rightPar(def=⟩)`: rigth parentheses

Description: Creating a type. Should max consist of two letters or nested type functions. Within the command, the abreviated command `\t{type}` can be used instead of `\type{type}` which will shorten the nested code.
Bold Option: No

Example: `\type{e\t{et}}` = $\langle e\langle et\rangle\rangle$

(45)

```
\sarg{index}
```

Arguments:

`index`: the situation index

Description: Creating a situation variable with an index.
Bold Option: Yes (uppercase r results in superscript indexing)

Example: s_5

(46)

```
\trace{variable}[traceSymbol][spacing]
```

Arguments:

`variable`: the variable for which the trace is for
`traceSymbol (def=`t`)`: the trace symbol
`spacing (def=kern-0.3pt)`: the spacing of the subscript

Description: Creates a trace with a given variable.
Bold Option: Yes

Example: t_x

(47)

```
\sarrow{function}
```

Arguments:

`variable`: the characteristic function.

Description: Creates a downarrow and puts the function behind it without space.
Bold Option: Yes (will put everything behind arrow in square brackets)

Example: $\downarrow[\![\mathbf{sleeps}]\!]^{s^*}$

(48)

```
\cset{function}[argument][situation]
```

Arguments:

`variable`: the characteristic set.
`argument(def=)`: optional specifier argument.
`situation(def=`s^*`)`: the situation argument.

Description: Creates a set.
Bold Option: No

Example: `\cset{Kills}[Peter]` will create the set of every person that got killed by Peter in the situation s^*, which will look like this: $Kills_{s^*}^{\text{Peter}}$

(49)

```
\func[name]{domain1}{range1}[domain2][range2][domain3][range3][domain4][range4]
```

Arguments:

`name`: the optional function name.
`domain1`: the first domain.
`range1`: the first range.
`domain2`: the optional second domain.
`range2`: the optional second range.
`domain3`: the optional third domain.
`range3`: the optional third range.
`domain4`: the optional fourth domain.
`range4`: the optional fourth range.

Description: Creates a function in array notation. This command can be nested. It is a 'quick and dirty' version of the *fglfunc* environment and should only be used for small functions or inside text. Problems regarding vertical spacing can arise when creating bigger functions via nesting, in that case use Environment 5.2 and change the spacing parameter.
Since: 1.1.1

Example: `\func[A]{\szero}{B}[\sone][C][\stwo][D]` will create: A $= \begin{bmatrix} s_0 & \mapsto & \text{B} \\ s_1 & \mapsto & \text{C} \\ s_2 & \mapsto & \text{D} \end{bmatrix}$

Example: `\func[B]{a}{1}[b][1][c][\func{x}{1}[y][2][x][3]]` will create: B $= \begin{bmatrix} \text{a} & \mapsto & & & 1 \\ \text{b} & \mapsto & & & 1 \\ \text{c} & \mapsto & \begin{bmatrix} \text{x} & \mapsto & 1 \\ \text{y} & \mapsto & 2 \\ \text{x} & \mapsto & 3 \end{bmatrix} & & \end{bmatrix}$

4.5.1 Direct Interpretation

This section covers commands for direct semantic interpretation.

(50)

```
\sdi{expression}
```

Arguments:

`expression`: the expression to interpret.

Description: Creatin double square bracketing for an object language expression without a situation argument.
Bold Option: Yes

Example: $[\![\textbf{sleeps}]\!]$

(51)

```
\sdim{expression}
```

Arguments:

`expression`: the expression to interpret.

Description: Creatin double square bracketing for a meta language expression without a situation argument.
Bold Option: Yes

Example: $[\![\text{sleeps}]\!]$

(52)

```
\sde[situation][spacing]{expression}
```

Arguments:

`situation(def=`s^*`)`: the situation argument.
`spacing(def=negthinspace)`: the spacing of the superscript to the brackets.
`expression`: the expression to interpret.

Description: Creatin double square bracketing for an object language expression with a situation argument.
Bold Option: Yes

Example: $[\![\textbf{sleeps}]\!]^{s^*}$

(53)

```
\sdem[situation][spacing]{expression}
```

Arguments:

`situation(def=`s^*`)`: the situation argument.
`spacing(def=negthinspace)`: the spacing of the superscript to the brackets.
`expression`: the expression to interpret.

Description: Creatin double square bracketing for a meta language expression with a situation argument.
Bold Option: Yes

Example: $[\![\text{sleeps}]\!]^{s^*}$

4.5.2 Indirect Interpretation

This section covers commands for indirect semantic interpretation.

(54)

```
\sic[situation]{constant}
```

Arguments:

`situation(def=`i`)`: the situation argument.
`constant`: the function abriviation.

Description: Makes constant italic and adds an argument in the subscript.
Bold Option: No (uppercase removes notational convention of subscript)

Example: $\mathbf{Q}_i$

(55)

```
\sicn[situation]{constant}
```

Arguments:

`situation(def=`i`)`: the situation argument.
`constant`: the function abriviation.

Description: Makes negated constant italic and adds an argument in the subscript.
Bold Option: Yes (c uppercase removes notational convention of subscript)

Example: $\neg\mathbf{Q}_i$

(56)

```
\sit[superscript][spacing]{expression}
```

Arguments:

`expression`: the expression to translate.

Description: Creats single bar bracketing for indirect translation of an expression.
Bold Option: Yes

Example: ||

(57)

```
\sitnobf{expression}
```

Arguments:

`expression`: the expression to translate.

Description: Creats single bar bracketing for indirect translation of an expression but without making the expression bold.
Bold Option: Yes

Example: |sleeps|

(58)

```
\sitnobfi{expression}
```

Arguments:

`expression`: the expression to translate.

Description: Creats single bar bracketing for indirect translation of an expression but making it italic and not bold.
Bold Option: Yes

Example: |*sleeps*|

(59)

```
\sid[assignment][spacing]{expression}
```

Arguments:

`assignment`: the assignment function.
`spacing(def=negthinspace)`: the spacing of the superscript to the brackets.
`expression`: the expression to denote.

Description: Creats double bar bracketing for indirect denotation of an expression with optional assignment function.
Bold Option: Yes

Example: $\|\mathbf{sleeps}\|$

(60)

```
\sidg[assignmentReplacement][spacing][spacingAssignment]{expression}
```

Arguments:

`assignmentReplacement`: will be written behind the g.
`spacing(def=negthinspace)`: the spacing of the superscript to the brackets.
`spacingAssignment(def=negthinspace)`: the spacing between the var. in the assigment function.
`expression`: the expression to denote.

Description: Creats double bar bracketing for indirect denotation of an expression with g a assignment function.
Bold Option: Yes

Example: $\|\mathbf{sleeps}\|^{g}$

(61)

```
\asmod{replacement}[original][spacingAssignment]
```

Arguments:

`replacement`: the new variable to replace the original
`original(def=x)`: the variable to be replaced
`spacingAssignment(def=negthinspace)`: the spacing between the var. in the assigment function.

Description: Creates an assignment function with a replacement.
Bold Option: Yes

Example: $[^{x}/_{r}]$

(62)

```
\asf{arg}[replacement1][original1][replacement2][original2][replacement3][original
3][assignment]
```

Arguments:

`arg`: argument to be assigned.
`replacement1`: the new variable to replace the original
`original1(def=x)`: the variable to be replaced
`replacement2`: the new variable to replace the original
`original2(def=y)`: the variable to be replaced
`replacement3`: the new variable to replace the original
`original3(def=z)`: the variable to be replaced
`assignment(def=g)`: the assignment function

Description: Applies an argument to an assignment function and 3 optional replacement spots.
Bold Option: Yes

Example: $g[^{x}/_{r}](\mathrm{p})$

(63)

```
\sidmod{expression}{replacement1}[original1][replacement2][original2][replacement
3][original3][assignment]
```

Arguments:

`expression`: the expression to denote.
`replacement1`: the new variable to replace the original
`original1(def=x)`: the variable to be replaced

`replacement2`: the new variable to replace the original
`original2(def=y)`: the variable to be replaced
`replacement3`: the new variable to replace the original
`original3(def=z)`: the variable to be replaced
`assignment(def=g)`: the assignment function

Description: Creats double bar bracketing for indirect denotation of an expression with g as assignment function and 3 optional replacement spots. More specified versions of this function are

Bold Option: Yes (case of d switches bracketing from bold to normal; case of o switches case of original variable from bold to normal)

Example: $\|\mathbf{test}\|^{g[x/r][u/b][z/h]}$

Example: $\mathbf{\|test\|}^{g[x/r][u/b][z/h]}$

(64)

```
\sidr{expression}{replacement}[original][assignment][spacingAssignment]
```

Arguments:

`expression`: the expression to denote.
`replacement`: the new variable to replace the original
`original(def=x)`: the variable to be replaced
`assignment(def=g)`: the assignment function
`spacingAssignment(def=negthinspace)`: the spacing between the var. in the assigment function.

Description: Creats double bar bracketing for indirect denotation of an expression with g as assignment function and a replacement spot.

Bold Option: Yes (case of d switches bracketing from bold to normal; case of r switches case of original variable from bold to normal)

Example: $\|\mathbf{sleeps}\|^{g[x/r]}$

(65)

```
\sidrr{expression}{replacement1}{replacement2}[original1][original2][assignment][
spacingAssignment]
```

Arguments:

`expression`: the expression to denote.
`replacement1`: the 1st new variable to replace the original
`replacement2`: the 2nd new variable to replace the original
`original1(def=x)`: the 1st variable to be replaced
`original2(def=y)`: the 2nd variable to be replaced
`assignment(def=g)`: the assignment function
`spacingAssignment(def=negthinspace)`: the spacing between the var. in the assigment function.

Description: Creats double bar bracketing for indirect denotation of an expression with g a assignment function and two replacement spots.

Bold Option: Yes (case of d switches bracketing from bold to normal; case of r switches case of original variable from bold to normal)

Example: $\|\mathbf{sleeps}\|^{g[x/r][y/t]}$

(66)

```
\sidrrr{expression}{replacement1}{replacement2}{replacement3}[original1][original
2][original3][assignment][spacingAssignment]
```

Arguments:

`expression`: the expression to denote.
`replacement1`: the 1st new variable to replace the original
`replacement2`: the 2nd new variable to replace the original
`replacement3`: the 3rd new variable to replace the original
`original1(def=x)`: the 1st variable to be replaced
`original2(def=y)`: the 2nd variable to be replaced
`original3(def=z)`: the 3rd variable to be replaced
`assignment(def=g)`: the assignment function
`spacingAssignment(def=negthinspace)`: the spacing between the var. in the assigment function.

Description: Creats double bar bracketing for indirect denotation of an expression with g a assignment function and three replacement spots.

Bold Option: Yes (case of d switches bracketing from bold to normal; case of r switches case of original variable from bold to normal)

Example: $\|\mathbf{sleeps}\|^{g[x/r][y/t][z/w]}$

4.5.3 Rules

This section covers the most general rules for direct and indirect semantic interpretation along with some general rules used for derivation in compositional semantics.

4.6 Constants

This section will list all pre-defined ‘fragoli’ commands acting as constants for common types or general symbols in semantics.

4.6.1 Situations

‘fragoli’ offers the most common situtation variables for fast usage. The following are available: blank, *** and from *0* to *10*. Examples:

```
\sblank   s
\sblanK   s
\sstar    s*
\sstaR    s*
\szero    s0
\szer0    s0
\sone     s1
\sonE     s1
```

4.6.2 Lambda Heads

‘fragoli’ offers a wide range of pre-defined lambda heads for fast usage. All heads are available in a lower- and uppercase version as well as a normal and bold version. E.g. `\lmds` will create “$\lambda s.$”. The last letter defines the lamba head variable; If the last letter is uppercase, the head variable will also be uppercase. The case of “d” defines if the head is bold (uppercase) or normal (lowercase). The following letters are available: “a-z”. Example for all p versions:

```
\lmdp     λp.
\lmDp     λp.
\lmdP     λP.
\lmDP     λP.
```

4.6.3 Types

‘fragoli’ offers a set of predefined types - covering the most common ones. These constants will not be affected by the package option *usetypes* and will also be displayed if *usetypes=false*. If you want them to only be displayed if *usetypes=true*, add an @ after the backslash (e.g. `\@typet`). The parentheses style will be affected by the package option *typestyle*. The syntax for these types is defined by the letter case of every letter following *type*. Every adjacent letter with matching uppercase will be on the same level. E.g. `\typeeET` vs. `\typeEEt` will result in $\langle e\langle et\rangle\rangle$ vs. $\langle\langle ee\rangle t\rangle$. For types with a depth greater than 3 the syntax will also contain l & r as left and right parentheses. `\typeletETrt` vs. `\typeetlETtr` will result in $\langle\langle et\rangle\langle et\rangle t\rangle$ vs. $\langle\langle et\rangle\langle et\rangle t\rangle$. Examples:

```
\typee      ⟨e⟩
\types      ⟨s⟩
\typed      ⟨t⟩
\typet      ⟨d⟩
\typeet     ⟨et⟩
\typett     ⟨tt⟩
\typeSEt    ⟨⟨se⟩t⟩
\typeeET    ⟨e⟨et⟩⟩
\typeslEetr ⟨s⟨e⟨et⟩⟩⟩
\typeretETlt ⟨⟨et⟩⟨et⟩t⟩
\typeetrETtl ⟨⟨et⟩⟨et⟩t⟩
```

4.6.4 Traces

'fragoli' also offers commands for the most common traces for fast usage. The following are available (also as bold option):

```
\tracex   t_x
\tracey   t_y
\tracez   t_z
```

For an uppercase version, write the last character in uppercase. For a bold version write the *e* in *trace* in uppercase.

5 Environments

This section provides an overview of all newly introduced environments. Most environments include a set of specialized commands that are only accessible within the respective environment. Some environments are restricted, meaning they can only be used within a specific parent environment.

5.1 exams

The *question* and *answer* environments are designed to be used together. The intended purpose is to create a single LaTeX file for homework assignments or exams that contains both the questions and their solutions—useful for tutorial sessions or sample solutions. By simply adjusting the package argument, one can toggle between displaying only the questions or including the solutions as well.

(67)

```
\begin{question}
  content...
\end{question}
```

Arguments: None

Description: Creates an environment that represents a question. If the *showanswer* package argument is set to *true*, the content within this environment will be displayed in light gray; otherwise, it will appear in black. example text

New Commands: No

Restricted: No

Example: `\begin{question}How does the knights move?\end{question}`

(68)

```
\begin{awnser}
  content...
\end{awnser}
```

Arguments: None

Description: Creates an environment that represents an answer. If the *showanswer* package argument is set to *true*, the content within this environment will be displayed; otherwise, it will be hidden.

New Commands: No

Restricted: No

Example:

```
\begin{awnser}It generally moves like an L and sometimes in blitz games it can be very unpredictable.\end{awnser}
```

5.2 fglfunc

The *fglfunc* environment should be used to create large functions in array notaion. This env. supports nesting. The *fgls* and *cps* environments are not enabled. When nesting multiple times it is recommended to change the vertical spacing to at least 14. Be carefull with linebreaks within the env. as this can result in an additional blank row or compiling errors!

(69)

```
\begin{fglfunc}[functionName][verticalSpacing][domainMode]
  content...
\end{fglfunc}
```

Arguments:

`functionName`: The main function name.
`verticalSpacing(def=10)`: The vertical spacing between each row.
`domainMode(def=n)`: The text formatting mode of the domains.

Description: Creates an environment for fast and easy function creation. Possible domain modes are *n/m/b* which stand for *normal*, *mathmode*, and *bold*. They will effect the text formatting of each domain.
New Commands: Yes
Restricted: No
Since: 1.1.1

Example: See below.

Example for a short nested function K:

```
\begin{fglfunc}[\sde{kisses}][15]
    \f{Peter}{
    \b{
      \f{Peter}{0}
      \f{Maria}{1}
      \f{Frida}{0}}\d}
    \f{Maria}{
    \b{
      \f{Peter}{0}
      \f{Maria}{0}
      \f{Frida}{0}}\d}
    \d
\end{fglfunc}
```

The code above will result in the following array structure:

$$[\![\textbf{kisses}]\!]^{s^*} = \begin{bmatrix} \text{Peter} & \mapsto & \begin{bmatrix} \text{Peter} & \mapsto & 0 \\ \text{Maria} & \mapsto & 1 \\ \text{Frida} & \mapsto & 0 \\ \vdots & & \end{bmatrix} \\ \text{Maria} & \mapsto & \begin{bmatrix} \text{Peter} & \mapsto & 1 \\ \text{Maria} & \mapsto & 0 \\ \text{Frida} & \mapsto & 0 \\ \vdots & & \end{bmatrix} \\ \vdots & & \end{bmatrix}$$

5.2.1 Commands

A few short commands are provided for fast and easy to read code.

(70)

```
\f{domain}{range}
```

Arguments:

`domain`: The domain.
`range`: the range.

Description: Will create a new row in the array representing an assignment.

(71)

```
\b{function}[functionName]
```

Arguments:

`function`: A nested function.
`functionName`: The nested function name.

Description: Will create a box with left and right brackets for a new nested function. To manually add left or right brackets you can use `\l` and `\r`. Do not add line breaks inside the box command.

(72)

```
\d[domain][symbol][range]
```

Arguments:

`domain(def=.)`: An optional domain.
`symbol`: The function symbol.
`range`: An optional range.

Description: Will create a new row with optional parameters. Should be used to indicate infinity or to add comments.

5.3 cps

The colored parentheses environment is designed to improve readability in semantic derivations. Most modern integrated development environments feature a similiar feature/plugin - called rainbow brackets. Parentheses on the same nesting level will receive the same color. Currently only "()" are supported. After five levels of nesting the colors will repeat. Currently the feature is limited to parentheses directly written in the environment, those inserted by commands will not be affected (work in progress).

(73)

```
\begin{cps}
  scontent...
\end{cps}
```

Arguments: None
Description: Parentheses on the same nesting level will receive the same color.
New Commands: No
Restricted: No

Example: (1(2((4((6(7))5))3)))

5.4 fgls

The fragoli-semantics environment is designed to improve speed and readability in semantic derivations by adding shortcuts and specialised versions of general commands.

(74)

```
\begin{fgls}[situation]
  content...
\end{fgls}
```

Arguments:

`situation(def=`s^*`)`: The situation which will be displayed on all commands that make use of a situation (e.g. `\e{}`).

Description: Adds shortcuts and specialised versions of general commands.
New Commands: Yes

Restricted: No

(75)

```
\fglsem[situation]{formulae}[maxWidth][alignment]
```

Arguments:

`situation(def=`s^*`)`: The situation which will be displayed on all commands that make use of a situation (e.g. `\e{}`).
`formulae`: The semantic formulae.
`maxWidth(def=linewidth)`: The maximum width of the varwidth box.
`alignment(def=same as varwidth default`:The vertical alignment of the varwidth box.

Description: Command wrapper for the *fgls* environment. Should be used for short formulae or formulae within text. Will remove initial indent and line breaks. Will put everything in an varwidth box.

Since: 1.1.1

5.4.1 Constants

This section will list all pre-defined commands acting as constants for common types or general symbols in the *fgls* environment.

1. Object Language
   `\or` is currently not available.

`\ox`	$\boldsymbol{x}$
`\oy`	$\boldsymbol{y}$
`\oz`	$\boldsymbol{z}$
`\oi`	$\boldsymbol{i}$
`\oj`	$\boldsymbol{j}$
`\os`	$\boldsymbol{s}$
`\ot`	$\boldsymbol{t}$
`\og`	$\boldsymbol{g}$
`\od`	$\boldsymbol{d}$
`\oR`	$\boldsymbol{R}$
`\oq`	$\boldsymbol{q}$
`\op`	$\boldsymbol{p}$
`\x`	$\boldsymbol{x}$
`\y`	$\boldsymbol{y}$
`\z`	$\boldsymbol{z}$

   Each object language constant exists in two forms: lowercase and uppercase. The uppercase version is represented by capitalizing the second letter of the command. Additionally, each constant has a secondary mode that automatically adds a space following the constant. To activate this mode, capitalize the first letter of the command. Both forms can be used simultaneously. The letters x, y, and z have special single-character commands, as they are the most commonly used. As a trade-off, these letters do not have a secondary mode that automatically appends a space.

2. Meta Language

```
\mx
\my
\mz
\mi
\mj
\ms
\mt
\mg
\md
\mr
\mp
\mq
\mu
\mv
```

x
y
z
i
j
s
t
g
d
r
p
q
u
v

Each meta language constant exists in two forms: lowercase and uppercase. The uppercase version is represented by capitalizing the second letter of the command. Additionally, each constant has a secondary mode that automatically adds a space following the constant. To activate this mode, capitalize the first letter of the command. Both forms can be used simultaneously. To get the Greek letter μ use `\greekmu`. For advanced existential quantifiers see Section 5.4.2.

3. Logic

```
\qe
\qu
\qa
\jn
\jc
\jd
\ji
\jj
\jx
\jb
\jp
\jt
\jf
\je
\jo
```

$\exists$
$\exists!$
$\forall$
$\neg$
$\wedge$
$\vee$
$\rightarrow$
$\leftrightarrow$
$\veebar$
$\Box$
$\Diamond$
$\top$
$\bot$
$\equiv$
$\oplus$

Every logic constant exists in two versions, normal and bold. For the bold version just write the second letter in uppercase. All junctor constants have a second mode that automatically adds a space behind them (for the arrows it adds a space on both sides). To use this mode just write the first letter in uppercase. Both versions can be combined.

4. Set theory

```
\ci
\cd
\ce
\cb
\cq
\ca
\cu
```

$\in$
$\downarrow$
$\emptyset$
$\subset$
$\subseteq$
$\cap$
$\cup$

Every set notation exists in two versions, normal and bold. For the bold version just write the second letter in uppercase. All operators have a second mode that automatically adds a space behind them. To use this mode just write the first letter in uppercase. Both versions can be combined.

5. Relations

```
\rn
\rg
\rs
\rp
\rpe
\rq
\rqe
```

$\not=$
$\not>$
$\not<$
$\not\geq$
$\not\leq$
$\not\succ$
$\not\prec$

Every relation operator exists in two versions, normal and bold. For the bold version just write the last letter in uppercase. All operators have a second mode that automatically adds a space before and behind them. To use this mode just write the first letter in uppercase. Both versions can be combined.

6. Arguments Variable

```
\ax
\aX
\ay
\aY
\az
\aZ
\ai
\aI
\as
\aS
\ad
\aD
\ar
\aR
\ap
\aP
\aq
\aQ
\ae
\aE
```

$(\boldsymbol{x})$
$(\boldsymbol{X})$
$(\boldsymbol{y})$
$(\boldsymbol{Y})$
$(\boldsymbol{z})$
$(\boldsymbol{Z})$
$(\boldsymbol{i})$
$(\boldsymbol{I})$
$(\boldsymbol{s})$
$(\boldsymbol{S})$
$(\boldsymbol{d})$
$(\boldsymbol{D})$
$(\boldsymbol{r})$
$(\boldsymbol{R})$
$(\boldsymbol{p})$
$(\boldsymbol{P})$
$(\boldsymbol{q})$
$(\boldsymbol{Q})$
$(\boldsymbol{e})$
$(\boldsymbol{E})$

```
\Ax
\AX
\Ay
\AY
\Az
\AZ
\Ai
\AI
\As
\AS
\Ad
\AD
\Ar
\AR
\Ap
\AP
\Aq
\AQ
\Ae
\AE
```

$(\mathbf{x})$
$(\mathbf{X})$
$(\mathbf{y})$
$(\mathbf{Y})$
$(\mathbf{z})$
$(\mathbf{Z})$
$(\mathbf{i})$
$(\mathbf{I})$
$(\mathbf{s})$
$(\mathbf{S})$
$(\mathbf{d})$
$(\mathbf{D})$
$(\mathbf{r})$
$(\mathbf{R})$
$(\mathbf{p})$
$(\mathbf{P})$
$(\mathbf{q})$
$(\mathbf{Q})$
$(\mathbf{e})$
$(\mathbf{E})$

```
\bx
\bX
\by
\bY
\bz
\bZ
\bi
\bI
\bs
\bS
\bd
\bD
\br
\bR
\bp
\bP
\bq
\bQ
\be
\bE
```
(x)
(X)
(y)
(Y)
(z)
(Z)
(i)
(I)
(s)
(S)
(d)
(D)
(r)
(R)
(p)
(P)
(q)
(Q)
(e)
(E)

7. Situations
   `\cs` will always return the current situation set in the current fgls env. (def=s^*)

```
\ss
\sz
\so
\st
\si
```
s^*
s_0
s_1
s_2
i

   Each situation constant exists in two forms: normal and bold. The bold version is represented by capitalizing the second letter of the command. Additionally, each situation constant has a secondary mode, which automatically adds a space following the constant. To activate this mode, capitalize the first letter of the command. Both forms can be used simultaneously. It is important to note that the symbol i has a special rule: when the first letter is lowercase, the constant is always represented in subscript, while the version with an uppercase first letter is treated as a normal variable with an added space at the end.

8. Traces

```
\tx
\ty
\tz
```
t_x
t_y
t_z

   Each trace constant exists in multiple forms: normal, bold, uppercase, and with an additional space (each combination is possible). The bold version is represented by capitalizing the first letter of the command. The uppercase version is represented by capitalizing the second letter of the command. Additionally, each situation constant has a secondary mode, which automatically adds a space following the constant. To activate this mode, append an additional s to the command (e.g. `\txs`).

9. Types

```
\te
\tet
\teet
\ts
\tst
\tse
\tt
\ttt
```
$\langle e \rangle$
$\langle et \rangle$
$\langle e\langle et \rangle\rangle$
$\langle s \rangle$
$\langle st \rangle$
$\langle se \rangle$
$\langle t \rangle$
$\langle tt \rangle$

Shortcuts for the most common types. Since the orgininal command is already short, only the most common ones got even shorter commands.

5.4.2 Commands

Text-Commands:

(76)

```
\o{text}
```

Arguments:

`text`: Text to format.

Description: Shortcut for `\obl{text}`
Bold Option: No

Example: **text**

(77)

```
\v{text}
```

Arguments:

`text`: Text to format.

Description: Shortcut for `\obli{text}`
Bold Option: No

Example: ***text***

(78)

```
\m{text}
```

Arguments:

`text`: Text to format.

Description: Shortcut for `\mel{text}`
Bold Option: No

Example: text

(79)

```
\j{text}
```

Arguments:

`text`: Text to format.

Description: Shortcut for `\meli{text}`
Bold Option: No

Example: *text*

(80)

```
\sub{text}
```

Arguments:

`text`: Text for italic subscript.

Description: Shortcut for `\ensuremath{_{text}}`
Bold Option: Yes

Example: $_{text}$

(81)

```
\sup{text}
```

Arguments:

`text`: Text for italic superscript.

Description: Shortcut for `\ensuremath{^{text}}`
Bold Option: Yes

Example: ^{text}

Argument-Commands:

(82)

```
\a{arg}
```

Arguments:

`arg`: object language variable as argument.

Description: Shortcut for `\obl{(}\obli{arg}\obl{)}`
Bold Option: No

Example: (a)

(83)

```
\A{arg}
```

Arguments:

`arg`: object language constant as argument.

Description: Shortcut for `\obl{(arg)}`
Bold Option: No

Example: $(\mathbf{a})$

(84)

```
\ab{arg}
```

Arguments:

`arg`: object language variable as argument with bold brackets.

Description: Shortcut for `\B{\obli{arg}}`
Bold Option: Yes (makes art non italic)

Example: $(\boldsymbol{a})$

(85)

```
\sa{function}
```

Arguments:

`function`: the characteristic function.

Description: Shortcut for `\sarrow{function}`
Bold Option: Yes

Example: $\downarrow[\![\mathbf{Peter}]\!]^{s^*}$

Lambda-Commands:

(86)

```
\lfx{word}[lambdaBody][lambdaHead][situation][headType]
```

Arguments:

`word`: the word to create the function for
`lambdaBody`: the lambda function body, if empty the body will be "`lambdaHead word` in s^*".
`lambdaHead(def=x)`: the lambda head variable
`situation(def=`s^*`)`: the situation inside the lambda body.
`headType`: the lambda head type

Description: Specialized version of `\lambfx{lambdaBody}[lambdaHead][headType]`
Bold Option: Yes

Example: $\lambda x.\vdash x$ sleeps in $s^* \dashv$

(87)

```
\lfy{word}[lambdaBody][lambdaHead1][lambdaHead2][situation][headType1][headType2]
```

Arguments:

`word`: the word to create the function for
`lambdaBody`: the lambda function body, if empty the body will be "`lambdaHead1 word lambdaHead2` in s^*".
`lambdaHead1(def=y)`: the first lambda head variable
`lambdaHead2(def=x)`: the second lambda head variable
`situation(def=`s^*`)`: the situation inside the lambda body
`headType1`: the first lambda head type
`headType2`: the second lambda head type

Description: Specialized version of `\lambfyx{lambdaBody}[lambdaHead1][lambdaHead2][headType1][headType2]`
Bold Option: Yes

Example: $\lambda y.\lambda x.\vdash x$ kills y in $s^* \dashv$

(88)

```
\lfz{word}[lambdaBody][lambdaHead1][lambdaHead2][lambdaHead3][situation][headType
1][headType2][headType3]
```

Arguments:

`word`: the word to create the function for
`lambdaBody`: the lambda function body, if empty the body will be "*lambdaHead1* `word lambdaHead2` *lambdaHead3* in s^*".
`lambdaHead1(def=z)`: the first lambda head variable
`lambdaHead2(def=y)`: the second lambda head variable
`lambdaHead3(def=x)`: the third lambda head variable
`situation(def=`s^*`)`: the situation inside the lambda body
`headType1`: the first lambda head type
`headType2`: the second lambda head type
`headType3`: the third lambda head type

Description: Specialized version of `\lambfzyx{lambBody}[lambHead1][lambHead2][lambHead3][headType1][headType2][headType3]`
Bold Option: Yes

Example: $\lambda z.\lambda y.\lambda x.\vdash x$ gives y z in $s^* \dashv$

(89)

```
\lfsx{word}[lambdaBody][situationHead][lambdaHead2][situationType][headType2]
```

Arguments:

`word`: the word to create the function for
`lambdaBody`: the lambda function body, if empty the body will be "`lambdaHead2 word` in `situationHead`".
`situationHead(def=s)`: the situation lambda head variable
`lambdaHead2(def=x)`: the second lambda head variable
`headType1`: the situation lambda head type (should be used if type should be displayed)
`headType2`: the second lambda head type

Description: Specialized version of `\lambfsx{lambdaBody}[lambdaHead1][lambdaHead2][headType1][headType2]`
Bold Option: Yes

Example: $\lambda s.\lambda x.\vdash x$ sleeps in $s \dashv$

(90)

```
\lfsy{word}[lambdaBody][situationHead][lambdaHead2][lambdaHead3][situationType][headType2][headType3]
```

Arguments:

`word`: the word to create the function for
`lambdaBody`: the lambda function body, if empty the body will be "`lambdaHead3 word lambdaHead2 in situationHead`".
`situationHead(def=s)`: the situation lambda head variable
`lambdaHead2(def=y)`: the second lambda head variable
`lambdaHead3(def=x)`: the third lambda head variable
`headType1`: the situation lambda head type (should be used if type should be displayed)
`headType3`: the second lambda head type
`headType3`: the third lambda head type

Description: Specialized version of
`\lambfsyx{lambdaBody}[lambdaHead1][lambdaHead2][lambdaHead3][headType1][headType2][headType3]`
Bold Option: Yes

Example: $\lambda s.\lambda y.\lambda x.\vdash x$ kills y in $s\dashv$

(91)

```
\lfsz{word}[lambdaBody][situationHead][lambdaHead2][lambdaHead3][lambdaHead4][situationHead][headType2][headType3]
```

Arguments:

`word`: the word to create the function for
`lambdaBody`: the lambda function body, if empty the body will be
"`lambdaHead4 word lambdaHead3 lambdaHead2 in situationHead`".
`situationHead(def=s)`: the situation lambda head variable
`lambdaHead2(def=z)`: the second lambda head variable
`lambdaHead3(def=y)`: the third lambda head variable
`lambdaHead4(def=x)`: the fourth lambda head variable
`headType1`: the situation lambda head type (should be used if type should be displayed)
`headType2`: the second lambda head type
`headType3`: the third lambda head type (fourth lambda head will get the same type due to the 9 args limit in LaTeX)

Description: Specialized version of
`\lambfszyx{lambdaBody}[lambdaHead1][lambdaHead2][lambdaHead3][lambdaHead4][headType1][headType2][headType3][headType4]`
Bold Option: Yes

Example: $\lambda s.\lambda z.\lambda y.\lambda x.\vdash x$ gives y z in $s\dashv$

There are pre-defined lambda heads for the most common variables and Quantors. Each one has four different versions which can been seen below for the example of x. The variables are: x,y,z,p,q,r,s,t,d,i,u,v.

(92)

```
\lx[type]
\lX[type]
\Lx[type]
\LX[type]
```

Arguments:

`type`: type of the lambda head variable

Description: Shortcut for `\lambh{var}[type]`
Bold Option: Yes (first letter defines if the lambda head is bold, second letter defines the lambda head variable)

Example: $\lambda x.$, $\lambda X.$, $\boldsymbol{\lambda x}.$, $\boldsymbol{\lambda X}.$

(93)

```
\lha[variable][type]
\lhan[variable][type]
\lhe[variable][type]
\lhen[variable][type]
```

Arguments:

`variable`: the variable bound by the quantor
`type`: type of the lambda head variable

Description: Shortcut for various lambda heads with quantifiers.
Bold Option: Yes

Example: $(\forall x_{\langle e \rangle})$, $(\forall \boldsymbol{x}_{\langle e \rangle})$, $\neg(\forall x_{\langle e \rangle})$, $\neg(\forall \boldsymbol{x}_{\langle e \rangle})$, $(\exists x_{\langle e \rangle})$, $(\exists \boldsymbol{x}_{\langle e \rangle})$, $\neg(\exists x_{\langle e \rangle})$, $\neg(\exists \boldsymbol{x}_{\langle e \rangle})$

Derivation-Commands:

(94)

```
\r[subInner][subOuter]{arg}
```

Arguments:

`subInner`: Optional subscript behind opening bracket. Text will be in mathmode. (mostly used in tree like structures)
`subOuter`: Optional subscript behind closing bracket. Text will be in mathmode (mostly used to identify types)
`arg`: term inside rectangular brackets.

Description: Shortcut for `\lambby{arg}`
Bold Option: Yes

Example: [arg]

(95)

```
\b{text}
```

Arguments:

`text`: text inside normal brackets.

Description: Shortcut for `(text)`. Should be used mostly for bold mode or within a *cps* env. for brackets that one does not want to be colored.
Bold Option: Yes

Example: (text)

(96)

```
\n{arg}
```

Arguments:

`arg`: term inside truth value brackets.

Description: Shortcut for `\bool{arg}`
Bold Option: Yes

Example: ⊢arg⊣

(97)

```
\s{arg}[var]
```

Arguments:

`arg`: Set-builder notation
`var(def=x)`: Varibale name and/or domain specifier

Description: Shortcut for `\set[var]{arg}`. To write the set variable use `\sv` or `\Sv` for the version with an extra space appended.
Bold Option: Yes

Example: `\s{\Sv is a dog}` will result in: $\{x : x \text{ is a dog}\}$

(98)

```
\i{expression}
```

Arguments:

`expression`: the expression to interpret.

Description: Shortcut for `\sdi{arg}`

Bold Option: Yes

Example: $[\![\mathbf{Dog}]\!]$

(99)

```
\e[situation]{expression}
```

Arguments:

`situation(def=`s^*`)`: the situation.
`expression`: the expression to interpret.

Description: Shortcut for `\sde[situation]{arg}`
Bold Option: Yes

Example: $[\![\mathbf{Dog}]\!]^{s^*}$

(100)

```
\c[situation]{constant}
```

Arguments:

`situation(def=`i`)`: the situation.
`constant`: the constant.

Description: Shortcut for `\sic[situation]{constant}`
Bold Option: No (Uppercase version removes notational convention)

Example: $\mathbf{S}_i$

(101)

```
\t{expression}
```

Arguments:

`expression`: the expression to translate.

Description: Shortcut for `\sit{expression}`
Bold Option: Yes

Example: ||

(102)

```
\ti{expression}
```

Arguments:

`expression`: the expression to translate.

Description: Shortcut for `\sit{\textit{expression}}`
Bold Option: Yes

Example: ||

(103)

```
\tb{expression}
```

Arguments:

`expression`: the expression to translate.

Description: Shortcut for `\sitnobf{expression}`
Bold Option: Yes

Example: |Dog|

(104)

```
\tbi{expression}
```

Arguments:

`expression`: the expression to translate.

Description: Shortcut for `\sitnobfi{expression}`
Bold Option: Yes

Example: $|Dog|$

(105)

```
\h{type}
```

Arguments:

`type`: the Ty2 type.

Description: Shortcut for `\type{type}`
Bold Option: Yes

Example: $\langle et \rangle$

(106)

```
\d[assignment]{expression}
```

Arguments:

`assignment(def=g)`: the assignment function.
`expression`: the expression to denote.

Description: Shortcut for `\sid[assignment]{expression}`
Bold Option: Yes

Example: $\|\mathbf{Dog}\|^{g}$

(107)

```
\dr{expression}[replacement][assignment][original]
```

Arguments:

`expression`: the expression to denote.
`replacement(def=r)`: the variable to replace the original with.
`assignment(def=g)`: the assignment function.
`original(def=x)`: the original argument to be replaced

Description: Shortcut for `\sidr{expression}{replacement}[original][assignment]`
Bold Option: No (Uppercase makes original variable bold)

Example: $\|\mathbf{Dog}\|^{g[x/r]}$

(108)

```
\drr{expression}[replacement1][replacement2][assignment][original1][original2]
```

Arguments:

`expression`: the expression to denote.
`replacement1(def=r)`: the first variable to replace the original with.
`replacement2(def=d)`: the second variable to replace the original with.
`assignment(def=g)`: the assignment function.
`original1(def=x)`: the fist original argument to be replaced
`original2(def=y)`: the second original argument to be replaced

Description: Shortcut for `\sidrr{expression}{replacement1}{replacement2}[original1][original2][assignment]`
Bold Option: No (Uppercase makes original variable bold)

Example: $\|\mathbf{Dog}\|^{g[x/r][y/d]}$

(109)

```
\drrr{expression}[replacement1][replacement2][replacement3][assignment][original1][original2][original3]
```

Arguments:

`expression`: the expression to denote.
`replacement1(def=r)`: the first variable to replace the original with.
`replacement2(def=d)`: the second variable to replace the original with.
`replacement3(def=t)`: the third variable to replace the original with.
`assignment(def=g)`: the assignment function.
`original1(def=x)`: the fist original argument to be replaced
`original2(def=y)`: the second original argument to be replaced
`original3(def=z)`: the third original argument to be replaced

Description: Shortcut for
`\sidrrr{expression}{replacement1}{replacement2}{replacement3}[original1][original2][original3][assignment]`
Bold Option: No (Uppercase makes original variable bold)

Example: $\|\mathbf{Dog}\|^{g[x/r][y/d][z/t]}$

Quantifier-Commands:

(110)

```
\qe[relation][value]
```

Arguments:

`relation`: the mathematical relation (mathmode already enabled)
`value`: the numerical value

Description: Creates an existential quantifier with additional parameters.
Bold Option: Yes
Space Option: Yes
Since: 1.1.1
Example:

$\exists^{>1}$

(111)

```
\qeg[value]
```

Arguments:

`value(def=2)`: the numerical value

Description: Creates an existential quantifier with a greater than or equal to paramter.
Bold Option: Yes
Space Option: Yes
Since: 1.1.1
Example:

$\exists^{\geq 2}$

(112)

```
\qes[value]
```

Arguments:

`value(def=2)`: the numerical value

Description: Creates an existential quantifier with a smaller than or equal to paramter.
Bold Option: Yes
Space Option: Yes
Since: 1.1.1
Example:

$\exists^{\leq 2}$

(113)

```
\qee[value]
```

Arguments:

`value(def=2)`: the numerical value

Description: Creates an existential quantifier with an equal to paramter.
Bold Option: Yes
Space Option: Yes
Since: 1.1.1
Example:

$\exists^{=2}$

5.5 semderivation

The *semantic-derivation* environment is the heart of the FraGoLi package and is used for fast and easy derivation of a direct or indirect semantic expressions. The *fgls* and *cps* env. are enabled automatically.

(114)

```
\begin{semderivation}[label]
content...
\end{semderivation}
```

Arguments:

`label`: The item label, if blank the label will be the current position of this env. inside the *semcalc*. If you add something, make sure to add `\item` infront.

Description: Env. for easy derivation of direct or indirect semantic expressions.
New Commands: Yes

Restricted: No

5.5.1 Constants

This section lists all predefined commands that serve as constants for common derivation steps within the *semderivation* environment. These commands are intended to be used as sidenotes. If the language is changed via the package options, the constants will be automatically translated and abbreviated. Some of these are interchangeable.

`\alp`	α-conversion
`\bta`	β-reduction
`\fa`	Functional application
`\abs`	Abstraction
`\mod`	Modification
`\var`	Variable assignment
`\con`	Con
`\id`	Id
`\lc`	λ-conversion
`\nc`	Notation convention
`\lex`	lexical trees

5.5.2 Commands

(115)

```
\ds[sidenote]{derivationRow}[itemLabel]
```

Arguments:

`sidenote`: Sidenote text.
`derivationRow`: The derivational step performed.
`itemLabel`: The itemize item label for the current step, if blank the label will be an equal sign with the current row number on top.

Description: Adds a new step to the derivation without the equals and row counter, should theirfore be used as first row

Bold Option: No (Uppercase removes equals and row counter, should be used as first step to indicate the beginning)

(116)

```
\cdr
```

Arguments: none

Description: Returns the current derivation reference string. For example sem:deri:3 if used in the third derivation env. or sem:deri:derithree if used in a semcalc env with derithree as refName argument.

5.6 semcalc

The *semantic-calculation* environment is designed to improve speed and to provide a clear structure for a full semantic derivation. If not specified otherwise via a package option, *cps* is enabled in the *semderi* environment. The *fgls* shortcuts are enabled.

(117)

```
\begin{semcalc}[situation][refName]
content...
\end{semcalc}
```

Arguments:

`situation(def=`s^*`)`: The situation which will be displayed on all extensions. The default value is set by the situation package parameter.
`refName(def=@fglsderivationcounter)`: The reference name used in the automatic labels that will be created for each resource entry (e.g. rules, lexicon, etc.) and derivation step. By default fragoli will start at 1 and increase eachtime a new semantic derivation is created (including semderivation and semderi env.).

Description: Adds shortcuts and specialised versions of general commands.
New Commands: Yes

Restricted: No

5.6.1 Commands

Both *resource-entry* commands should be used inside the *semrule* and *semlex* environment to add a rule or lexicon entry to the derivation. The sidenote information should referenz the origin of a rule or lexicon entry (e.g. from a script or paper) or to referenz the rule or lexicon entry used in a derivational step.

(118)

```
\rentry[sidenote]{resourceEntry}[label]
```

Arguments:

`sidenote`: Sidenote text.
`resourceEntry`: The resource entry (either a rule or a lexicon entry)
`label`: The item label, if blank the label will be R/L + the current rule/lexicon index.

Description: Add either a rule or a lexicon entry.

(119)

```
\rentrysub[sidenote]{resourceSubEntry}
```

Arguments:

`sidenote`: Sidenote text.
`resourceSubEntry`: Detailed information for a resource entry.

Description: Give additional information about a rule or a lexicon entry.

(120)

```
\rr{resourceCounter}
```

Arguments:

`resourceCounter`: The number of the rule to be referenced

Description: Creates a reference to the rule specified. Its the short version of `\ref{\cdr:r:number}`. For a lexicon entry use `\rl{resourceCounter}`, for a language entry use `\rla{resourceCounter}`, for a syntactic rule entry use `\rrs{resourceCounter}`

(121)

```
\cdr
```

Arguments: none

Description: Returns the current derivation reference string. For example sem:deri:3 if used in the third derivation env. or sem:deri:derithree if used in a semcalc env with derithree as refName argument.

5.7 semlang

The *semantic-lang* environment is an optional environment to define the lnguage for any derivation of a direct or indirect semantic expressions. So far no custom commands are implemented. Each language gets automatically label (if used via `\rentry` or predefined language command) with the following pattern: "sem:deri:refName:la:rulecounter". So the first language in the second semcalc env. will have the following label: "sem:deri:2:la:1" assuming that no custom refName has been set.

(122)

```
\begin{semlang}[label]
content...
\end{semlang}
```

Arguments:

`label`: The item label, if blank the label will be the current position of this env. inside the *semcalc*. If you add something, make sure to add `\item` infront.

Description: Env. for defining the language your woking in for this current derivation.
New Commands: Yes

Restricted: Yes (semcalc)

5.7.1 Commands

The FraGoLi package offers a quick builder to create a language based of sets of lexical entries.

(123)

```
\rlang{entires}[setOne][setTwo][setThree][setFour][setFive][setSix][setSeven][
setEight][symbol]
```

Arguments:

`entires`: Every element of the language, separated by a comma.
`setOne`: An optional set of entries to group similar ones.
`setTwo`: An optional set of entries to group similar ones.
`setThree`: An optional set of entries to group similar ones.
`setFour`: An optional set of entries to group similar ones.
`setFive`: An optional set of entries to group similar ones.
`setSix`: An optional set of entries to group similar ones.
`setSeve`: An optional set of entries to group similar ones.
`setEigth`: An optional set of entries to group similar ones.
`symbol(def=L)`: The symbol for the main language set.

Description: Builds a set that defines the language. If any optional set is added, they will all be unified.

Example: `\rlang{Peter, Maria, liebt}[NN][VP]` Result: $L = L_{NN} \cup L_{VP} = \{Peter, Maria, liebt\}$

5.8 semtree

The *semantic-tree* environment is an optional environment to add general tree structures to any derivation of a direct or indirect semantic expressions. So far no custom tree generating commands are implemented, all tree packages should work but *forest* package is tested best and recommended.

(124)

```
\begin{semtree}[label]
content...
\end{semtree}
```

Arguments:

> `label`: The item label, if blank the label will be the current position of this env. inside the *semcalc*. If you add something, make sure to add `\item` infront.

Description: Env. for additional trees of semantic expressions.
New Commands: No

Restricted: Yes (semcalc)

5.9 semtreesem

The *semantic-tree-semantic* environment is a specialized version of the *semantic-tree* env. and should be used for semantic trees. So far no custom tree generating commands are implemented, all tree packages should work but *forest* package is tested best and recommended.

(125)

(125)

```
\begin{semtreesem}[label]
content...
\end{semtreesem}
```

Arguments:

> `label`: The item label, if blank the label will be the current position of this env. inside the *semcalc*. If you add something, make sure to add `\item` infront.

Description: Env. for additional semantic trees of semantic expressions.
New Commands: No

Restricted: Yes (semcalc)

5.10 semtreesyn

The *semantic-tree-syntactic* environment is a specialized version of the *semantic-tree* env. and should be used for syntactic trees. So far no custom tree generating commands are implemented, all tree packages should work but *forest* package is tested best and recommended.

(126)

(126)

```
\begin{semtreesyn}[label]
content...
\end{semtreesyn}
```

Arguments:

> `label`: The item label, if blank the label will be the current position of this env. inside the *semcalc*. If you add something, make sure to add `\item` infront.

Description: Env. for additional syntactic trees of semantic expressions.
New Commands: No

Restricted: Yes (semcalc)

5.11 semrule

The *semantic-rules* environment is used to list all rules used for a derivation of a direct or indirect semantic expressions. Each rule gets automatically label (if used via `\rentry` or predefined rule) with the following pattern: "sem:deri:refNamer:rulecounter". So the first rule in the second semcalc env. will have the following label: "sem:deri:2:r:1" assuming that no custom refName has been set.

(127)

```
\begin{semrule}[label][situation]
content...
\end{semrule}
```

Arguments:

`label`: The item label, if blank the label will be the current position of this env. inside the *semcalc*. If you add something, make sure to add `\item` infront.
`situation(def=s)`: The situation which will be displayed on all extensions. The default value is set by the rulesituation package parameter.

Description: Env. for easy listing of semantic derivation rules.
New Commands: Yes

Restricted: Yes (semcalc)

5.11.1 Commands

The FraGoLi package offers the most common semantic rules for both direct and indirect interpretation based on the accompanying material to an introductory course to linguistic semantics by *Thomas Ede Zimmermann*. All pre-defined rules can be custemized with aspect to variables and lambda heads. General commands for fast rule creation are also featured. All commands start with rsr (resource semantic rule) followed by a categorization into general/direct/indirect e.g. `\rsrgarrow` for `resource-semantic-rule-general-arrow`. All pre-defined rules have the option to add a sidenote. If the package otpion usetypes is enabled all pre-defined rules will show the type on each lambda head. All rules are based on their basic versions just wrapped inside a resource entry command. Currently the command catagorie abriviations are:

`rsrg`	resource-semantic-rule-general
`rsrde`	resource-semantic-rule-direct-exenstion
`rsrdeqr`	resource-semantic-rule-direct-extension-quantifier-raising
`rsrdi`	resource-semantic-rule-direct-intension
`rsrit`	resource-semantic-rule-indirect-translation
`rsrid`	resource-semantic-rule-indirect-denotation

General Rules

(128)

```
\rsrgarrow[sidenote]{var}{function}
```

Arguments:

`sidenote`: Sidenote text.
`var(def=x)`: Variable.
`function(def=f)`: Function.

Description: Give additional information about a rule or a lexicon entry.
Example: `\rsrgarrow`

R1: $\downarrow f = \{x : f(x) = 1\}$

Direct Interpretation Rules

(129)

```
\rsrde{arg1}{arg2}[sidenote][situation][label]
```

Arguments:

`arg1`: left argument.
`arg2`: right argument.
`sidenote`: Sidenote text.
`situation(def=`s^*`)`: The situtaion.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation extension rule.
Example: `\rsrde{Left}{Right}`

R1: $[\![\textbf{Left}]\!]^s$ = Right

(130)

```
\rsrdi{arg1}{arg2}[sidenote][label]
```

Arguments:

`arg1`: left argument.
`arg2`: right argument.
`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrdi{Left}{Right}`

R1: $[\![\textbf{Left}]\!]$ = Right

(131)

```
\rsrdesbjpred[sidenote][situation][label]
```

Arguments:

`sidenote`: Sidenote text.
`situation(def=`s^*`)`: The situtaion.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrdesbjpred`

R1: $[\![S]\!]^s = [\![P]\!]^s([\![NN]\!]^s)$

(132)

```
\rsrdesbjquant[sidenote][situation][label]
```

Arguments:

`sidenote`: Sidenote text.
`situation(def=`s^*`)`: The situtaion.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrdesbjquant`

R1: $[\![S]\!]^s = [\![QN]\!]^s([\![P]\!]^s)$

(133)

```
\rsrdeobjpred[sidenote][situation][label]
```

Arguments:

`sidenote`: Sidenote text.
`situation(def=`s^*`)`: The situtaion.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrdeobjpred`

R1: $[\![P]\!]^s = [\![V]\!]^s([\![NN]\!]^s)$

(134)

```
\rsrdeobjquant[sidenote][arg1][arg2][situation][type1][type2][label]
```

Arguments:

`sidenote`: Sidenote text.
`arg1(def=x)`: The first argument.
`arg2(def=y)`: The second argument.
`situation(def=`s^*`)`: The situation.
`type1(def=`$\langle e \rangle$`)`: The type of the first argument
`type2(def=`$\langle e \rangle$`)`: The type of the second argument
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrdeobjquant`

R1: $[\![P]\!]^s = \lambda x_{\langle e \rangle}.[\![QN]\!]^s(\lambda y_{\langle e \rangle}.[\![V]\!]^s(y)(x))$

(135)

```
\rsrdequant[sidenote][situation][label]
```

Arguments:

`sidenote`: Sidenote text.
`situation(def=`s^*`)`: The situtaion.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrdequant`

(i) *Rules*:
R1: $[\![QN]\!]^s = [\![D]\!]^s([\![N]\!]^s)$

(136)

```
\rsrdecoordination[sidenote][situation][label]
```

Arguments:

`sidenote`: Sidenote text.
`situation(def=`s^*`)`: The situtaion.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrdecoordination`

R1: $[\![\boldsymbol{S}\ \boldsymbol{K}\ \hat{\boldsymbol{S}}]\!] = [\![\boldsymbol{K}]\!][\![\hat{\boldsymbol{S}}]\!]([\![\boldsymbol{S}]\!])$

(137)

```
\rsrdiconva[sidenote][situation][label]
```

Arguments:

`sidenote`: Sidenote text.
`situation(def=s)`: The situtaion.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrdiconva`

R1: $[\![\boldsymbol{A}]\!] = \lambda s_{\langle s \rangle}.[\![\boldsymbol{A}]\!]^s$

(138)

```
\rsrdiconvb[sidenote][situation][label]
```

Arguments:

`sidenote`: Sidenote text.
`situation(def=s)`: The situtaion.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrdiconvb`

R1: $[\![\boldsymbol{A}]\!](s) = [\![\boldsymbol{A}]\!]^{s}$

(139)

```
\rsrdeattitude[sidenote][situation][label]
```

Arguments:

`sidenote`: Sidenote text.
`situation(def=`s^*`)`: The situtaion.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrdeattitude`

R1: $[\![\boldsymbol{P}]\!]^{s^*} = [\![\boldsymbol{V}]\!]^{s^*}([\![\boldsymbol{S}]\!])$

Indirect Interpretation Translation Rules

(140)

```
\rsrit{left}{right}[sidenote][label]
```

Arguments:

`left`: Left argument.
`right`: Right argument
`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrit{Left}{Right}`

R1: || = Right

(141)

```
\rsriti{left}{right}[sidenote][label]
```

Arguments:

`left`: Left argument.
`right`: Right argument
`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsriti{Left}{Right}`

R1: || = Right

(142)

```
\rsritibasic{left}{right}{rightArg}[sidenote][label]
```

Arguments:

`left`: Left argument.
`right`: Right argument
`rightArg`: Argument on the right side.
`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsritibasic{Left}{Right}{Arg}`

R1: || = ||(||)

(143)

```
\rsritcoordination[sidenote][label]
```

Arguments:

`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsritcoordination`

R1: || = ||(||)(||)

(144)

```
\rsritpredication[sidenote][label]
```

Arguments:

`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsritpredication`

R1: || = ||(||)

(145)

```
\rsritnamedirectobject[sidenote][label]
```

Arguments:

`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsritnamedirectobject`

R1: || = ||(||)

(146)

```
\rsritquantification[sidenote][label]
```

Arguments:

`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsritquantification`

R1: || = ||(||)

(147)

```
\rsritquantificational[sidenote][label]
```

Arguments:

`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.

Example: `\rsritquantificational`

R1: $||$ = $||(||)$

(148)

```
\rsritquantificationasobj[sidenote][arg1][arg2][type1][type2][label]
```

Arguments:

`sidenote`: Sidenote text.
`arg1(def=x)`: First argument.
`arg1(def=y)`: Second argument.
`type1(def=⟨et⟩)`: Type of the first argument
`type2(def=⟨et⟩)`: Type of the second argument
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsritquantificationasobj`

R1: $||$ = $\boldsymbol{\lambda x_{\langle et\rangle}.||(\lambda y_{\langle et\rangle}.||(y)(x))}$

(149)

```
\rsritattitude[sidenote][situation][situationType][label]
```

Arguments:

`sidenote`: Sidenote text.
`situation(def=i)`: The situation.
`situationType(def=⟨s⟩)`: The situation type (should be used to always display type).
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsritattitude`

R1: $||$ = $\boldsymbol{||(\lambda i_{\langle s\rangle}.||)}$

(150)

```
\rsritraisedsubj[sidenote][situation][situationType][label]
```

Arguments:

`sidenote`: Sidenote text.
`situation(def=i)`: The situation.
`situationType(def=⟨s⟩)`: The situation type (should be used to always display type).
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsritraisedsubj`

R1: $||$ = $||(\boldsymbol{\lambda i_{\langle s\rangle}.||})$

(151)

```
\rsritcontrolverbs[sidenote][arg1][arg2][type1][type2][label]
```

Arguments:

`sidenote`: Sidenote text.
`arg1(def=x)`: First argument.
`arg1(def=i)`: Situation argument.
`type1(def=⟨et⟩)`: Type of the first argument
`type2(def=⟨s⟩)`: Type of the situation argument
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsritcontrolverbs`

R1: $||$ = $\boldsymbol{\lambda x_{\langle e\rangle}.||(\lambda i_{\langle s\rangle}.||(x))(x)}$

Indirect Interpretation Denotation Rules

(152)

```
\rsrid{right}{left}[sidenote][assignment][label]
```

Arguments:

`right`: Right argument.
`left`: Left argument.
`sidenote`: Sidenote text.
`assignment(def=g)`: The assignment function var.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsrid{Right}{Left}`

R1: $\|\mathbf{Right}\|^g = \text{Left}$

(153)

```
\rsridinterpretvar[sidenote][assignment][arg][label]
```

Arguments:

`sidenote`: Sidenote text.
`assignment(def=g)`: The assignment function var.
`arg(def=x)`: The argument variable.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsridinterpretvar`

R1: $\|\boldsymbol{x}\|^g = g(\boldsymbol{x})$

(154)

```
\rsridapp[sidenote][assignment][arg1][arg2][label]
```

Arguments:

`sidenote`: Sidenote text.
`assignment(def=g)`: The assignment function var.
`arg1(def=`$\boldsymbol{\alpha}$`)`: The first argument.
`arg2(def=`$\boldsymbol{\beta}$`)`: The second argument.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsridapp`

R1: $\|\boldsymbol{\alpha}(\boldsymbol{\beta})\|^g = \|\boldsymbol{\alpha}\|^g(\|\boldsymbol{\beta}\|^g)$

(155)

```
\rsridabs[sidenote][assignment][arg][var][replacement][label]
```

Arguments:

`sidenote`: Sidenote text.
`assignment(def=`g`)`: The assignment function var.
`arg(def=`$\boldsymbol{\alpha}$`)`: The argument.
`var(def=`$\boldsymbol{x}$`)`: The variable to be replaced.
`replacement(def=`u`)`: The variable replacement.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation intension rule.
Example: `\rsridabs`

R1: $\|\boldsymbol{\lambda x.\alpha}\|^g = \lambda u.\|\boldsymbol{\alpha}\|^{g[x/u]}$

Quantifier Raising Rules

(156)

```
1 \rsrdeqrsemvar[sidenote][leftTerm][var][situation][assignment][label]
```

Arguments:

`sidenote`: Sidenote text.
`leftTerm(def=`ϕ`)`: The var. for the left term.
`var(def=`x`)`: The variable to be replaced.
`situation(def=`s`)`: The situation.
`assignment(def=`g`)`: The assignment function var.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation rule in the quantifier raising framework.
Example: `\rsrdeqrsemvar`

R1: $[\![\phi]\!]^{g,s} = g(x)$

(157)

```
1 \rsrdeqrabs[sidenote][rightTerm][replacement][situation][assignment][function][
  original][label]
```

Arguments:

`sidenote`: Sidenote text.
`rightTerm(def=`ϕ`)`: The var. for the left term.
`replacement(def=`x`)`: The variable to be replaced.
`situation(def=`s`)`: The situation.
`assignment(def=`g`)`: The assignment function var.
`function(def=`f`)`: The assignment function var.
`original(def=`ξ`)`: The assignment function var.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation rule in the quantifier raising framework.
Example: `\rsrdeqrabsr`

R1: $f(x) = [\![\phi]\!]^{g[\delta/x],s}$

(158)

```
1 \rsrdeqrlex[sidenote][leftTerm][rightTerm][replacement][situation][assignment][
  label]
```

Arguments:

`sidenote`: Sidenote text.
`leftTerm(def=`α`)`: The left term.
`rightTerm(def=`β`)`: The right term.
`situation(def=`s`)`: The situation.
`assignment(def=`g`)`: The assignment function var.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation rule in the quantifier raising framework.
Example: `\rsrdeqrlex`

R1: $[\![\alpha]\!]^{g,s} = [\![\beta]\!]^{s}$

(159)

```
1 \rsrdeqrfa[sidenote][leftTerm][rightTerm][rightArgTerm][situation][assignment][
  label]
```

Arguments:

`sidenote`: Sidenote text.
`leftTerm(def=`α`)`: The left term.
`rightTerm(def=`β`)`: The right term.
`rightArgTerm(def=`γ`)`: The right argument term.
`situation(def=`s`)`: The situation.
`assignment(def=`g`)`: The assignment function var.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation rule in the quantifier raising framework.
Example: `\rsrdeqrfa`

R1: $[\![\alpha]\!]^{g,s} = [\![\beta]\!]^{g,s}([\![\gamma]\!]^{g,s})$

(160)

```
\rsrdeqrtree[sidenote][lambda][leftTerm][rightTerm][replacement][situation][assignment][type][label]
```

Arguments:

`sidenote`: Sidenote text.
`lambda(def=x)`: The lambda head and replacement.
`leftTerm(def=ψ)`: The left term.
`rightTerm(def=ϕ)`: The right term.
`replacement(def=ξ)`: The assignment function modification var.
`situation(def=s)`: The situation.
`assignment(def=g)`: The assignment function var.
`type(def=⟨e⟩)`: The type of the lambda head.
`label`: The item label, if blank the label will be R + the current rule index.

Description: Create an entry for a direct interpretation rule in the quantifier raising framework.
Example: `\rsrdeqrtree`

R1: $[\![\psi]\!]^{g,s} = \lambda x_{\langle e \rangle}.[\![\phi]\!]^{g[\delta/x],s}$

5.12 semrulesyn

The *semantic-rule-syn* is an optional environment and is used to list all syntactic rules used for a derivation of a direct or indirect semantic expressions. Each syntactic rule gets automatically label (if used via `\rentry` or predefined syntactical rule) with the following pattern: "sem:deri:refNamers:rulecounter". So the first syntactical rule in the second semcalc env. will have the following label: "sem:deri:2:rs:1" assuming that no custom refName has been set.

(161)

```
\begin{semrulesyn}[label]
content...
\end{semrulesyn}
```

Arguments:

`label`: The item label, if blank the label will be the current position of this env. inside the *semcalc*. If you add something, make sure to add `\item` infront.

Description: Env. for syntactic rules.
New Commands: No

Restricted: Yes (semcalc)

5.13 semlex

The *semantic-lexicon* environment is used to list all lexicon entries used for a derivation of a direct or indirect semantic expressions. Each lexicon entry gets automatically label (if used via `\rentry` or predefined lexicon entry) with the following pattern: "sem:deri:refName:l:lexcounter". So the first lexicon entry in the second semcalc env. will have the following label: "sem:deri:2:l:1" assuming that no custom refName has been set.

(162)

```
\begin{semlex}[label]
content...
\end{semlex}
```

Arguments:

`label`: The item label, if blank the label will be the current position of this env. inside the *semcalc*. If you add something, make sure to add `\item` infront.

Description: Env. for easy listing of semantic lexicon entries.
New Commands: Yes

Restricted: Yes (semcalc)

5.13.1 Commands

The FraGoLi package offers the most common semantic lexicon entries for both direct and indirect interpretation based on the accompanying material to an introductory course to linguistic semantics by *Thomas Ede Zimmermann*. All pre-defined lexicon entries can be custemized with aspect to variables and lambda heads. General commands for fast rule lexicon creation are also featured. All commands start with sl (semantic lexicon) followed by a categorization into direct (plus intension/extension) or indirect (plus translation/denotation) e.g. `\sldenn` for `semantic-lexicon-direct-extension-properName`. All pre-defined lexicon entries have the option to add a sidenote. If the package otpion `usetypes` is enabled all pre-defined rules will show the type on each lambda head.
Direct Interpretation Lexicon Extension

(163)

```
1 slde{left}{right}[sidenote][situation][label]
```

Arguments:

`left`: The object language element
`rightv`: The meta language translation
`sidenote`: Sidenote text.
`situation(def=`s^*`)`: The situation
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for direct translaion of extensions.
Example: `\slde{ab}{\e{a} $\cdot$ 10 + \e{b}}`

L1: $[\![\mathbf{ab}]\!]^{s^*} = [\![\mathbf{a}]\!]^{s^*} \cdot 10 + [\![\mathbf{b}]\!]^{s^*}$

(164)

```
1 sldei{element}[sidenote][situation][label]
```

Arguments:

`left`: The object language element
`sidenote`: Sidenote text.
`situation(def=`s^*`)`: The situation
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for direct translaion of extensions with identical graphemes.
Example: `\sldei{10}`

L1: $[\![\mathbf{10}]\!]^{s^*} = 10$

(165)

```
1 \sldex{word}[lambdaBody][sidenote][var][situation][type][label]
```

Arguments:

`word`: The word to create the lexicon entry for
`lambdaBody`: The lambda body, if empty the text will be "*var* `word` in s^*"
`sidenote`: Sidenote text.
`var(def=x)`: Lambda head variable.
`situation(def=`s^*`)`: The situation
`type(def=`$\langle e \rangle$`)`: The type of the lambda head variable.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry function for intransitive verbs or adjectives.
Example: `\sldex{sleeps}`

L1: $[\![\mathbf{sleeps}]\!]^{s^*} = \lambda x_{\langle e \rangle}.\vdash x$ sleeps in $s^* \dashv$

(166)

```
1 \sldey{word}[lambdaBody][sidenote][var1][var2][situation][type1][type2][label]
```

Arguments:

`word`: The word to create the lexicon entry for
`lambdaBody`: The lambda body, if empty the text will be "*var1* `word` *var2* in s^*"
`sidenote`: Sidenote text.
`var1(def=x)`: Inner lambda head variable.
`var2(def=y)`: Outer lambda head variable.
`situation(def=`s^*`)`: The situation
`type1(def=`$\langle e \rangle$`)`: The type of the inner lambda head variable.
`type2(def=`$\langle e \rangle$`)`: The type of the outer lambda head variable.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry function for transitive verbs.
Example: `\sldey{kills}`

L1: $[\![\textbf{kills}]\!]^{s^*} = \lambda y_{\langle e \rangle}.\lambda x_{\langle e \rangle}.\vdash x \text{ kills } y \text{ in } s^* \dashv$

(167)

```
\sldez{word}[lambdaBody][sidenote][var1][var2][var3][situation][type1][type2]
```

Arguments:

`word`: The word to create the lexicon entry for.
`lambdaBody`: The lambda body, if empty the text will be "*var1* `word` *var2* in s^*".
`sidenote`: Sidenote text.
`var1(def=x)`: Inner lambda head variable.
`var2(def=y)`: Middle lambda head variable.
`var3(def=z)`: Outer lambda head variable.
`situation(def=`s^*`)`: The situation.
`type1(def=`$\langle e \rangle$`)`: The type of the inner lambda head variable.
`type2(def=`$\langle e \rangle$`)`: The type of the middle and outer lambda head variable.

Description: General lexicon entry function for ditransitive verbs.
Example: `\sldez{shows}`

L1: $[\![\textbf{shows}]\!]^{s^*} = \lambda z_{\langle e \rangle}.\lambda y_{\langle e \rangle}.\lambda x_{\langle e \rangle}.\vdash x \text{ shows } y \text{ } z \text{ in } s^* \dashv$

(168)

```
\sldenn{name}[sidenote][situation][label]
```

Arguments:

`name`: The proper name to create the lexicon entry for.
`sidenote`: Sidenote text.
`situation(def=`s^*`)`: The situation
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry function for proper names.
Example: `\sldenn{Peter}`

L1: $[\![\textbf{Peter}]\!]^{s^*} = \text{Peter}$

A little set of common semantic names is pre-defined like `\sldennpeter`. Those names are "Peter, Maria, Anna, Paul, Eike, Ida, Olaf, Michael, Alina, Marta". All pre-defined names can have sidenotes, custom situations and labels. The command syntax would be `\sldennpeter[sidenote][situation][label]`.

(169)

```
\slden{nominal}[article][sidenote][var][situation][type][label]
```

Arguments:

`nominal`: The nominal to create the lexicon entry for.
`article(def=a)`: The article in the lambda boy (will be switched to *ein* if language is set to german). If empty this will also be the case.
`sidenote`: Sidenote text.
`var(def=x)`: The lambda head variable
`situation(def=`s^*`)`: The situation
`type(def=`$\langle e \rangle$`)`: The lambda head var type.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry function for proper names.
Example: `\slden{Cat}`

L1: $[\![\textbf{Cat}]\!]^{s^*} = \lambda x_{\langle e \rangle}.\vdash x \text{ is a cat in } s^* \dashv$

A little set of common semantic nominals is pre-defined like `\sldenman`. Those will fully translate if language is set to *german*. All pre-defined nominals can have sidenotes, custom lambda heads, situations, types and labels. The command syntax would be `\sldenman[sidenote][var][situation][type][label]`.

```
\sldenman
\sldenwoman
\sldenchild
\sldendog
\sldencat
\sldenanimal
\sldenhouse
\sldendonkey
```

Mann
Woman
Child
Dog
Cat
Animal
House
Donkey

Direct Interpretation Lexicon Intension

(170)

```
sldi{left}{right}[sidenote][label]
```

Arguments:

`left`: The object language element
`rightv`: The meta language translation
`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for direct translaion of intensions.
Example: `\sldi{ab}{\i{a} $\cdot$ 10 + \i{b}}`

L1: $[\![\mathbf{ab}]\!] = [\![\mathbf{a}]\!] \cdot 10 + [\![\mathbf{b}]\!]$

(171)

```
\sldii{element}[sidenote][label]
```

Arguments:

`left`: The object language element
`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for direct translaion of intension with identical graphemes.
Example: `\sldii{10}`

L1: $[\![\mathbf{10}]\!] = 10$

(172)

```
\sldix{word}[lambdaBody][sidenote][var][situation][type1][type2][label]
```

Arguments:

`word`: The word to create the lexicon entry for
`lambdaBody`: The lambda body, if empty the text will be "*var* `word` in `situation`"
`sidenote`: Sidenote text.
`var(def=x)`: Lambda head variable.
`situation(def=s)`: The situation
`type1(def=⟨e⟩)`: The type of the lambda head variable.
`type2(def=⟨s⟩)`: The type of the situation lambda head variable (should be used if you want to display the type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry function for intransitive verbs or adjectives.
Example: `\sldix{sleeps}`

L1: $[\![\mathbf{sleeps}]\!] = \lambda s_{\langle s \rangle}.\lambda x_{\langle e \rangle}.\vdash x$ sleeps in $s\dashv$

(173)

```
\sldiy{word}[lambdaBody][sidenote][var1][var2][situation][type1][type2][type3]
```

Arguments:

`word`: The word to create the lexicon entry for
`lambdaBody`: The lambda body, if empty the text will be "*`var`* `word` in `situation`"
`sidenote`: Sidenote text.
`var(def=x)`: Lambda head variable.
`var2(def=y)`: Lambda head variable.
`situation(def=s)`: The situation
`type1(def=⟨e⟩)`: The type of the inner lambda head variable.
`type2(def=⟨e⟩)`: The type of the middle lambda head variable.
`type3(def=⟨s⟩)`: The type of the situation lambda head variable (should be used if you want to display the type).

Description: General lexicon entry function for intransitive verbs or adjectives.
Example: `\sldiy{kills}`

L1: $[\![\textbf{kills}]\!] = \lambda s_{\langle s\rangle}.\lambda y_{\langle e\rangle}.\lambda x_{\langle e\rangle}.\vdash x$ kills y in $s\dashv$

(174)

```
\sldiz{word}[lambdaBody][sidenote][var1][var2][var3][situation][type1][type2]
```

Arguments:

`word`: The word to create the lexicon entry for
`lambdaBody`: The lambda body, if empty the text will be "*`var`* `word` in `situation`"
`sidenote`: Sidenote text.
`var(def=x)`: Lambda head variable.
`var2(def=y)`: Lambda head variable.
`var3(def=z)`: Lambda head variable.
`situation(def=s)`: The situation
`type1(def=⟨e⟩)`: The type of the inner middle and second middle lambda head variable.
`type2(def=⟨s⟩)`: The type of the situation lambda head variable (should be used if you want to display the type).

Description: General lexicon entry function for intransitive verbs or adjectives.
Example: `\sldiz{shows}`

L1: $[\![\textbf{shows}]\!] = \lambda s_{\langle s\rangle}.\lambda z_{\langle e\rangle}.\lambda y_{\langle e\rangle}.\lambda x_{\langle e\rangle}.\vdash x$ shows y z in $s\dashv$

(175)

```
\sldinn{name}[sidenote][situation][type][label]
```

Arguments:

`name`: The proper name to create the lexicon entry for.
`sidenote`: Sidenote text.
`situation(def=s)`: The situation
`type(def=⟨s⟩)`: The situation type (should be used to display the type)
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry function for proper names.
Example: `\sldinn{Peter}`

L1: $[\![\textbf{Peter}]\!] = \lambda s_{\langle s\rangle}.$Peter

Indirect Interpretation Lexicon Translation

(176)

```
\slit{left}{right}[sidenote][label]
```

Arguments:

`left`: Left argument
`right`: Right argument.
`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect translation.
Example: `\slit{Left}{Right}`

L1: || = Right

(177)

```
\slitnn{name}[sidenote][label]
```

Arguments:

`name`: The proper name.
`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect proper name translation.
Example: `\slitnn{Peter}`

L1: $||= \mathbf{p}$

A little set of common semantic names is pre-defined like `\slitnnpeter`. Those names are "Peter, Maria, Anna, Alina, Marta, Paul, Eike, Ida, Olaf, Michael". All pre-defined names can have sidenotes, custom situations and labels. The command syntax would be `\slitnnpeter[sidenote][situation][label]`.

(178)

```
\slitf{name}[situation][sidenote][label]
```

Arguments:

`name`: The proper name.
`situation(def=i)`: The situation var.
`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect function translation.
Example: `\slitf{sleeps}`

L1: $||= \mathbf{S}_{i}$

(179)

```
\slitland[sidenote][label]
```

Arguments:

`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect translation.
Example: `\slitland`

L1: $||= \wedge$

(180)

```
\slitlor[sidenote][label]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=q)`: First lambda head variable.
`var2(def=p)`: Second lambda head variable.
`type1`: First lambda head variable type (should be used to display type).
`type2`: Secon lambda head variable type (should be used to display type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect translation.
Example: `\slitlor`

L1: $||= \lambda q_{\langle t \rangle}.\lambda p_{\langle t \rangle}.\neg[\neg p \wedge \neg q]$

(181)

```
\slitlno[sidenote][var1][var2][var3][type1][type2][type3][label]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=Q)`: First lambda head variable.
`var2(def=P)`: Second lambda head variable.
`var2(def=x)`: Bound variable (existential)
`type1(def=⟨et⟩)`: First lambda head variable type (should be used to display type).
`type2(def=⟨et⟩)`: Second lambda head variable type (should be used to display type).
`type3(def=⟨e⟩)`: Bound variable type (should be used to display type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect translation of quantifier no.
Example: `\slitlno`

L1: $||= \boldsymbol{\lambda Q}_{\langle et\rangle}.(\boldsymbol{\lambda P}_{\langle et\rangle}.\neg(\exists \boldsymbol{x}_{\langle e\rangle})[\boldsymbol{Q(x)} \wedge \boldsymbol{P(x)}])$

(182)

```
\slitloneindef[sidenote][var1][var2][var3][type1][type2][type3][label]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=Q)`: First lambda head variable.
`var2(def=P)`: Second lambda head variable.
`var3(def=x)`: Bound variable (existential)
`type1(def=⟨et⟩)`: First lambda head variable type (should be used to display type).
`type2(def=⟨et⟩)`: Second lambda head variable type (should be used to display type).
`type3(def=⟨e⟩)`: Bound variable type (should be used to display type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect translation of quantifier one (indefinite).
Example: `\slitloneindef`

L1: $||= \boldsymbol{\lambda Q}_{\langle et\rangle}.(\boldsymbol{\lambda P}_{\langle et\rangle}.(\exists \boldsymbol{x}_{\langle e\rangle})[\boldsymbol{Q(x)} \wedge \boldsymbol{P(x)}])$

(183)

```
\slitlonenum[sidenote][var1][var2][var3][var4][type1][type2][type3][type4]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=Q)`: First lambda head variable.
`var2(def=P)`: Second lambda head variable.
`var3(def=x)`: Bound variable outer (existential)
`var4(def=y)`: Bound variable inner (existential)
`type1(def=⟨et⟩)`: First lambda head variable type (should be used to display type).
`type2(def=⟨et⟩)`: Second lambda head variable type (should be used to display type).
`type3(def=⟨e⟩)`: Bound variable type (should be used to display type).
`type4(def=⟨e⟩)`: Bound variable type (should be used to display type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect translation of quantifier one (numerical).
Example: `\slitlonenum`

L1: $||= \boldsymbol{\lambda Q}_{\langle et\rangle}.(\boldsymbol{\lambda P}_{\langle et\rangle}.(\exists \boldsymbol{x}_{\langle e\rangle})[\ \boldsymbol{Q(x)} \wedge \boldsymbol{P(x)} \wedge \neg(\exists \boldsymbol{y}_{\langle e\rangle})[\neg(\boldsymbol{x} = \boldsymbol{y})\ \wedge \boldsymbol{Q(y)} \wedge \boldsymbol{P(y)}]])$

(184)

```
\slitlevery[sidenote][var1][var2][var3][type1][type2][type3][label]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=Q)`: First lambda head variable.
`var2(def=P)`: Second lambda head variable.
`var3(def=x)`: Bound variable (existential)
`type1(def=⟨et⟩)`: First lambda head variable type (should be used to display type).
`type2(def=⟨et⟩)`: Second lambda head variable type (should be used to display type).
`type3(def=⟨e⟩)`: Bound variable type (should be used to display type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect translation of quantifier every.
Example: `\slitlevery`

L1: $||= \boldsymbol{\lambda Q}_{\langle et\rangle}.(\boldsymbol{\lambda P}_{\langle et\rangle}.\neg(\exists \boldsymbol{x}_{\langle e\rangle})[\boldsymbol{Q(x)} \wedge \neg\boldsymbol{P(x)}])$

(185)

```
\slitlmost[sidenote][label]
```

Arguments:

`sidenote`: Sidenote text.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect translation of quantifier most.
Example: `\slitlmost`

L1: $||$ = **MOST**

(186)

```
\slitldefarticle[sidenote][var1][var2][var3][var4][type1][type2][type3][type4]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=Q)`: First lambda head variable.
`var2(def=P)`: Second lambda head variable.
`var3(def=x)`: Bound variable outer (existential)
`var4(def=y)`: Bound variable inner (existential)
`type1(def=⟨et⟩)`: First lambda head variable type (should be used to display type).
`type2(def=⟨et⟩)`: Second lambda head variable type (should be used to display type).
`type3(def=⟨e⟩)`: Bound variable type (should be used to display type).
`type4(def=⟨e⟩)`: Bound variable type (should be used to display type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect translation of definite article (Russell style).
Example: `\slitldefarticle`

L1: $||$ = $\boldsymbol{\lambda Q_{\langle et\rangle}.(\lambda P_{\langle et\rangle}.(\exists x_{\langle e\rangle})[Q(x) \wedge \neg(\exists y_{\langle e\rangle})[\neg(x = y) \wedge Q(y)] \wedge P(x)])}$

(187)

```
\slitadox[sidenote][var1][var2][var3][var4][type1][type2][type3][label]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=p)`: First lambda head variable.
`var2(def=x)`: Second lambda head variable.
`var3(def=j)`: Bound variable outer (existential)
`var4(def=i)`: Bound variable inner (existential)
`type1(def=⟨st⟩)`: First lambda head variable type (should be used to display type).
`type2(def=⟨e⟩)`: Second lambda head variable type (should be used to display type).
`type3(def=⟨s⟩)`: Bound variable type (should be used to display type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect translation of hintikka semantics. For other perspectives use `\slitaepi` or `\slitabou`.
Example: `\slitadox`

L1: $||$ = $\boldsymbol{\lambda p_{\langle st\rangle}.(\lambda x_{\langle e\rangle}.(\forall j_{\langle s\rangle})[\mathbf{DOX}(x)(i)(j) \rightarrow p_i])}$

Indirect Interpretation Lexicon Denotation

(188)

```
\slid{left}{right}[sidenote][assignment][label]
```

Arguments:

`left`: Left argument
`right`: Right argument.
`sidenote`: Sidenote text.
`assignment`: The assignment function.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General lexicon entry for indirect translation.
Example: `\slid{Left}{Right}[][g]`

L1: $\|\mathbf{Left}\|^g$ = Right

(189)

```
\slidnn{name}[sidenote][assignment][label]
```

Arguments:

`name`: The proper name.
`sidenote`: Sidenote text.
`assignment`: The assignment function.
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General interpretation of a proper name constant.
Example: `\slidnn{Peter}`

L1: $\|\mathbf{p}\| = \text{Peter}$

A little set of common semantic names is pre-defined like `\slidnnpeter`. Those names are "Peter, Maria, Anna, Alina, Marta, Paul, Eike, Ida, Olaf, Michael". All pre-defined names can have sidenotes, custom assignment functions and labels. The command syntax would be `\slidnnpeter[sidenote][assignment][label]`.

(190)

```
\slidland[sidenote][var1][var2][assignment][type1][type2][label]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=v)`: First lambda head variable.
`var2(def=u)`: Second lambda head variable.
`assignment`: The assignment function.
`type1(def=⟨t⟩)`: First lambda head variable type (should be used to display type).
`type2(def=⟨t⟩)`: Second lambda head variable type (should be used to display type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General interpretation of the logical constant and.
Example: `\slidland`

L1: $\|\wedge\| = \lambda v_{\langle t\rangle}.\lambda u_{\langle t\rangle}.[v \cdot u]$

(191)

```
\slidlexists[sidenote][var1][assignment][type1][label]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=P)`: First lambda head variable.
`assignment`: The assignment function.
`type1(def=⟨t⟩)`: First lambda head variable type (should be used to display type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General interpretation of the existential quantifier.
Example: `\slidlexists`

L1: $\|\exists\| = \lambda P_{\langle et\rangle}.\vdash\downarrow P \neq \emptyset\dashv$

(192)

```
\slidlneg[sidenote][var1][assignment][type1][label]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=u)`: First lambda head variable.
`assignment`: The assignment function.
`type1(def=⟨t⟩)`: First lambda head variable type (should be used to display type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General interpretation of the negation.
Example: `\slidlneg`

L1: $\|\neg\| = \lambda u_{\langle t\rangle}.[1 - u]$

(193)

```
\slidlequal[sidenote][var1][assignment][type1][label]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=y)`: First lambda head variable.
`var2(def=x)`: Second lambda head variable.
`assignment`: The assignment function.
`type1(def=⟨t⟩)`: First lambda head variable type (should be used to display type).
`type2(def=⟨t⟩)`: Second lambda head variable type (should be used to display type).
`label`: The item label, if blank the label will be L + the current lexicon index.

Description: General interpretation of the negation.
Example: `\slidlequal`

L1: $\|=\| = \lambda y_{\langle t \rangle}.\lambda x_{\langle t \rangle}.\vdash y = x \dashv$

(194)

```
\slidadox[sidenote][var1][assignment][type1][label]
```

Arguments:

`sidenote`: Sidenote text.
`var1(def=x)`: First lambda head variable.
`var2(def=$s_0$)`: Second lambda head variable.
`var2(def=$s_1$)`: Third lambda head variable.
`type1(def=⟨e⟩)`: First lambda head variable type (should be used to display type).
`type2(def=⟨s⟩)`: Second lambda head variable type (should be used to display type).
`type3(def=⟨s⟩)`: Third lambda head variable type (should be used to display type).
`assignment`: The assignment function.

Description: General indirect interpretation of hintikka semantics. For other perspectives use `\slidaepi` or `\slidabou`. Text gets automatically translated.
Example: `\slidadox`

L1: $\|\mathbf{DOX}\| = \lambda x_{\langle e \rangle}.\lambda s_{0\,\langle s \rangle}.\lambda s_{1\,\langle s \rangle}.\vdash s_1$ is a doxastic alternative of x in $s_0 \dashv$

5.14 semderi

The *semantic-derivation* environment is a specialized version of the *semderivation* env. and is tweaked to work inside a *semcalc* env.

(195)

```
\begin{semderi}[label]
content...
\end{semderi}
```

Arguments:

`label`: The item label, if blank the label will be the current position of this env. inside the *semcalc*. If you add something, make sure to add `\item` infront.

Description: Env. for easy derivation of direct or indirect semantic expressions.
New Commands: Yes

Restricted: Yes (semcalc)

5.14.1 Constants

This env. has the same new constants as the *semderivation* env.

5.14.2 Commands

This env. has the same new commands as the *semderivation* env.

6 Changelog

1.2.2
- Add degree semantics support.
- Add traces with spaces.
- Add new commands for semantic lexicon entries.
- Add typenestingstyle (credits to *Cécile Meier*)
- Add new pre defined lambda heads.
- Add commands to change package settings mid document.
- Add basic brackets command for *cps* prevention and bold mode.
- Add type shortcuts for fgls env.
- Add formatting options for types, situations and direct/indirect superscripts.
- Add lambda headers with quantifiers for fgls env.
- Fix fglsem command with regard to line breaks. Adding varwidth dependency for that.
- Fix some pre-defined lambda heads not beeing bold in bold mode.
- Fix spacing on lambda heads with visible types in upper mode
- Fix spacing on derivation steps

1.1.1
- Fix missing package dependency.
- Fix text overline overriding uuline internal length.
- Fix parentheses on lambda quantifier heads being bold in non bold mode.
- Add lambdaheadstyle package option.
- Add new fuction framework.
- Add new logic commands.
- Add new logic constants.
- Add new relation constants.
- Add new quantifier commands.
- Extend set command.

1.0.0
- Initial release.

0.2.2
- Add option to set default the default situation in the fgls, semrule, semcalc and semderivation env.
- Add current situation command and current derivation ref command.
- Automatic labels for all semcalc resources and for all derivation steps.
- Add rules from the Quantifier Raising Script by Dr. Cécile Meier.
- Add sarrow command.
- Add semrulesyn env.
- Add semtreesyn and semtreesem env.
- Add semlang env.
- Add language builder command to semlang env.
- Add traces

0.1.2
- Add semtree environment.
- Fixed derivation step counter.
- Fixed various documentation bugs.

0.0.1
- First build.