fff-description.grm

```
// FILE. . . . . d:/hak/hlt/src/hlt/fot/fuz/syntax/sources/fff-description.grm
// EDIT BY . . . Hassan Ait-Kaci
// ON MACHINE. . Hak-Laptop
// STARTED ON. . Tue Aug 21 05:03:21 2018

// Last modified on Mon Sep 24 09:28:54 2018 by hak
```

FFF Language Description

In **FFF**, one can specify similar pairs of functors and compute the least similarity on this signature containing these pairs. On a complete and consistent similarity on a signature, one then can compute fuzzy lattice operations, where the *infimum* operation (∧) is unification and the *supremum* operation (∨) is generalization, modulo the specified signature similarity. One can also specify partial argument-position aligment maps for some pairs of similar functors, and ask **FFF** to verify their consistency, by introducing new functors if necessary to obtain a consistent total argument-position maps for all similarity classes, or establish that such a completion is not possible by pointing out existing inconsistencies among the specified maps. **N.B.**: at the time of this documentation's writing, these last two features (the pragmas "`#map`" and "`#comp`") have not been implemented yet. But everything else that has been published is implemented and **FFF** runs as predicted there - see LOPSTR 2017:

- presentation slides;
- proceedings paper.

To illustrate **FFF** code and runs, here are a few examples.

Declarations: In **FFF**, one may declare and verify properties of various components of a first-order term signature and a similarity on it by using a *pragma*. A pragma is a command that has a specific effect. It has a name starting with the character ('`#`') and a pragma's name may be possibly followed by arguments. The number, order, nature, and syntax of such arguments depend on the pragma (see below). An **FFF** pragma is executed upon reading its ending semi-colon ('`;`') followed by a carriage-return on the current input stream (either the standard input or a file).

The pragmas that are currently supported are `#help`, `#fun`, `#sig`, `#sim`, `#close`, `#show`, `#funclass`, `#termclass`, `#funrep`, `#termrep`, `#map`, `#comp`, `#load`, `#trace`, and `#reset`, and are defined below. One enters:

- `#help #pragma;`

to print a description of pragma `#pragma`; if invoked with no argument (i.e., `#help;`), to print the list of defined pragmas;

- **`#fun`** $f_1/n_1 \ \ldots \ f_k/n_k$**`;`**.

  to declare signature functor symbols, where f_i is an identifier denoting a functor's name being declared, and n_i is a natural number specifying its arity, for each $i = 1,\ldots,k$, then print the current signature; the slashes "`/`" are optional and can be blank space; *e.g.*, `#fun` $f_1\ n_1\ \ldots\ f_k\ n_k$`;`;

- **`#sig;`**.

  to print the current signature (same as "`#fun;`");

- **`#sim`** $f_1\ g_1\ \alpha_1\ \ldots\ f_k\ g_k\ \alpha_k$**`;`**.

  to declare similar pairs of functors, where f_i and g_i denote (necessarily distinct and similar) functors' names and each α_i is a value in the interval `(0.0,1.0]` defining their similariy degree, for each $i = 1,\ldots,k$; then print the set of all declared similar pairs;

- **`#close;`**

  to compute the similarity closure of the set of fuzzy pairs that have been declared for the current signature;

- **`#show;`**

  to display the currently declared similar pairs on the current signature as a square matrix on the signature, of values in `[0.0,1.0]`; it also displays its corresponding set of similarity degrees, and lists the partitions per similarity degrees;

- **`#funclass`** $f\ \alpha$**`;`**

  if the similarity has been closed, to display the similarity class of functor f at similarity degree α;

- **`#termclass`** $t\ \alpha$**`;`**

  if the similarity has been closed, to display the similarity class of FOT t at similarity degree α;

- **`#funrep`** $f\ \alpha$**`;`**

  if the similarity has been closed, to show the similarity class representative of functor f at similarity degree α;

- **`#termrep`** $t\ \alpha$**`;`**

  if the similarity has been closed, to show the similarity class representative of FOT t at similarity degree α;

- **`#map`** $f\ g\ \alpha\ i_1{:}j_1\ \ldots\ i_k{:}j_k$**`;`**.

  to specify an argument-position map where:

- f and g are (necessarily distinct and similar) functors;
- α is a value in the interval `(0.0,1.0]` that belongs to the (finite) set of known approximation degrees in the current similarity on the current functor signature (if it is not, it will be set *de facto* to the greatest known positive degree in the similary that has a lesser value if one exists, otherwise this will generate an error);
- k is a natural number such that $0 \leq k \leq$ `Math.min(`f`.arity(),`g`.arity())`;
- each i`:`j is a pair of non-zero natural numbers such that $1 \leq i \leq$ f`.arity()` and $1 \leq j \leq$ g`.arity()` that indicates which argument positions of f and g are in (necessarily injective) mutual correspondence at approximation level α.

This is therefore equivalent to **`#map`** g f α j_1`:`i_1 `...` j_k`:`i_k.

Note that when k`=0` (*i.e.*, just entering `#map` f g α`;`), the similarity between any two term structures whose root symbols are these respective functors will involve none of their subterms at approximation degree α or less, regardless of whether either of their respective arities is non-zero.

N.B.: a `#map` pragma may be executed several times on the same two functors f and g for a same or different α. Upon invoking the pragma `#comp`, the compound effects on the same pair will result, in the order specified if these position maps are verified to be consistent, later ones overriding the effects of any earlier ones executed. Also, for any pair of functors $\langle f,g \rangle$ for which no `#map` is specified, the default is the identity on `{1, ...,` `Math.min(`f`.arity(),`g`.arity())}`.

Executing a `#map` defines how argument positions of similar functors correspond to one another. Therefore, it refers the similarity that has been defined. If no similarity is defined, it will define one as necessary; that is, the effect of `#map` f g α `...;` may trigger an automatic `#sim` f g α`;` if needed.

- **`#comp;`**

  to verify all the necessary *consistency conditions for partial non-aligned similar functor arguments*. That is, that all the declared argument-position maps between any similar pair of functors for the current signature's similarity, as the approximation degree decreases:

  1. have monotonically decreasing domains and ranges;
  2. are one-to-one from domain to range (and vice-versa); and,
  3. are consistent under argument-position map composition, completing the signature and its similarity if necessary and if a consistent completion is possible, or indicating that such a completion is not possible due to a mapping specification inconsistency and what causes it.

  (Click here for details.)

  Because a `#map` may implicitly trigger a `#sim` if needed, executing a `#comp;` may trigger a `#close;` as needed.

- **`#load "`***`file`***`";`**

  to load the clauses contained in the specified file and process them in reading order;

- **`#trace `***`n`***`;`**

  where n is a natural number, to turn on tracing mode at a verbosity detail level of n (the greater, the more verbose). To turn off tracing mode, one uses `#trace 0;` (or simply

`#trace;`). When in tracing mode, `FFF` gives details of what it does (useful for debugging or if one wishes to see this sort of information);

- **`#reset;`**

  to erase all declared functors, resetting the signature and the similarity.

Evaluations: **`FFF`** can evaluate term similarity and lattice operations on two well-formed first-order terms. These operations it performs are fuzzy unless no similarity is defined, in which case they are crisp. Such an expression has one of the following forms and is evaluated upon a carriage-return. Enter:

- **`lhs ~ rhs;`**

  where `lhs` and `rhs` are first-order terms, and **`FFF`** will tell you how similar these two terms are with an approximation degree $\alpha \in$ `[0.0,1.0]` (if no similarity is defined, this can only be equality: *i.e.*, either `0.0` or `1.0`);

- **`lhs /\ rhs;`**

  where `lhs` and `rhs` are first-order terms possibly sharing variables, and **`FFF`** will try to fuzzy-unify them. If they are unifiable (*i.e.*, if the resulting fuzzy degree α is greater than `0`), **`FFF`** will print the resulting term that is their fuzzy-greatest common instance `glb`, along with the most general unifying substitution σ and fuzzy degree $\alpha \in$ `(0.0,1.0]` such that `glb` $\approx_\alpha \sigma$`(lhs)` $\approx_\alpha \sigma$`(rhs)`. If not (*i.e.*, if the resulting fuzzy degree α is equal to `0.0`), **`FFF`** will print a message indicating that the terms are not unifiable.

- **`lhs \/ rhs;`**

  where `lhs` and `rhs` are first-order terms possibly sharing variables, and **`FFF`** will fuzzy-generalize them and print their fuzzy-least upper bound term `lub`, along with the two most fuzzy general generalizing substitutions σ_l and σ_r, and the greatest fuzzy degree α such that `lhs` $\approx_\alpha \sigma_l$`(lub)` and `rhs` $\approx_\alpha \sigma_r$`(lub)`. Note that there cannot be any failure in generalizing because it is always possible to generalize dissimilar terms with a new variable at any fuzzy degree $\alpha \in$ `(0.0,1.0]`.

Again, here are a few examples.

Technical Background: For more formal details on, and justifications of, **`FFF`**'s constructs and computations, please refer to this preliminary abstract spec until further evolution of either this software, or more details on the specs side.

```
/* ********************************************************************** */
```

This file was generated on Thu Sep 27 10:27:39 CEST 2018 from file `fff-description.grm`
by the `hlt.language.tools.Hilite` Java tool written by Hassan Aït-Kaci
