(Diff)List Appends in TDL

Guy Emerson

Delph-in 2017

Diff-list appends are fiddly

```
[ SYNSEM...RELS [ LIST #first,
                  LAST #last ],
  C-CONT.RELS [ LIST #middle2,
                LAST #last ],
  ARGS < [ SYNSEM...RELS [ LIST #first,
                           LAST #middle1 ]],
         [ SYNSEM...RELS [ LIST #middle1,
                           LAST #middle2 ]] >
```


Code like this is hard to maintain.

Diff-list appends are mechanical

```
dl-append := avm & [ APPARG1 [ LIST #first,
                               LAST #between ],
                     APPARG2 [ LIST #between,
                               LAST #last ],
                     RESULT  [ LIST #first,
                               LAST #last ]].
```


This type is in the Grammar Matrix, but a comment explicitly says not to use it. If we can write down a schema like this, why not write a type to implement it?

Nice syntax

```
[ NEW-LIST.APPEND < #1, #2, #3 >,
  LIST1 #1,
  LIST2 #2,
  LIST3 #3 ].
```


The aim is to write types so that this syntax is possible.

Nice syntax

```
[ SYNSEM...RELS.APPEND < #1, #2, #3 >,
  C-CONT.RELS #3,
  ARGS < [ SYNSEM...RELS #1 ],
         [ SYNSEM...RELS #2 ] >
```


The first example with nice syntax.

Types

```
diff-list-append := diff-list &
  [ LIST #start,
    LAST #end,
    APPEND list-of-dlists & [ START #start,
                              END #end ] ].

list-of-dlists := list &
  [ START list,
    END list ].
```


`diff-list-`append contains a list of diff-lists. The idea is that this list will automatically link up the diff-lists, put the result in START and END, and then this result is re-entrant with LIST and LAST.

START and END are effectively creating a diff-list, but I think it would be a bad idea to have a type inheriting from both `list` and `diff-list`, because it would make it harder to spot a bug resulting from accidentally unifying a `list` and a `diff-list`.

Types

```
cons-of-dlists := list-of-dlists & cons &
  [ FIRST diff-list & [ LIST #start,
                        LAST #middle ],
    REST list-of-dlists & [ START #middle,
                            END #end ],
    START #start,
    END #end ].
```


`list-of-dlists` on REST propagates all the constraints through the whole list.

`#middle` links up the diff-lists.

`#start` and #end expose the result.

Types

```
cons-of-dlists := list-of-dlists & cons &
  [ FIRST diff-list & [ LIST #start,
                        LAST #middle ],
    REST list-of-dlists & [ START #middle,
                            END #end ],
    START #start,
    END #end ].

null-of-dlists := list-of-dlists & null &
  [ START #null,
    END #null ].
```


At the end of the list of diff-lists, we need to say there’s nothing left to add.

Normal list appends

- Possible:
  - Convert lists to diff-lists
  - Append diff-lists
- Nice syntax not possible
  (without changing representation of lists)

Because there is no gravity in the type system, specifying a type on a feature path cannot change types higher up in the feature structure. In particular, we need the copy operation to behave differently for a `cons` and a `null`, but this can only trigger changes in the features of `cons` or `null`.

Dodgy syntax

```
[ OUTPUT-LIST #new,
  FIRST-LIST [ COPY #new,
               NEXT #second ],
  SECOND-LIST #second,
```


This syntax is possible, although it’s not as clean as for the diff-lists. Also note that each list can only be copied once.

Types

```
list-copy := list &
  [ COPY list,
    NEXT list ].
```


COPY stores the new open-ended list.

NEXT stores the end of the list.

This is essentially a diff-list, but as before, I didn’t want to directly inherit from both `list` and `diff-list`, to avoid bugs.

Types

```
cons-copy := list-copy & cons &
  [ FIRST #first,
    REST list-copy & [ COPY #rest,
                       NEXT #next ],
    COPY [ FIRST #first,
           REST #rest ],
    NEXT #next ].
```


`list-copy` on REST propagates the constraints.

`#first` makes sure that the new list and the old list have the same elements.

`#rest` makes sure that all the new list nodes combine into one long list (see last slide)

`#next` makes sure that we keep pass up the last node to the start of the list.

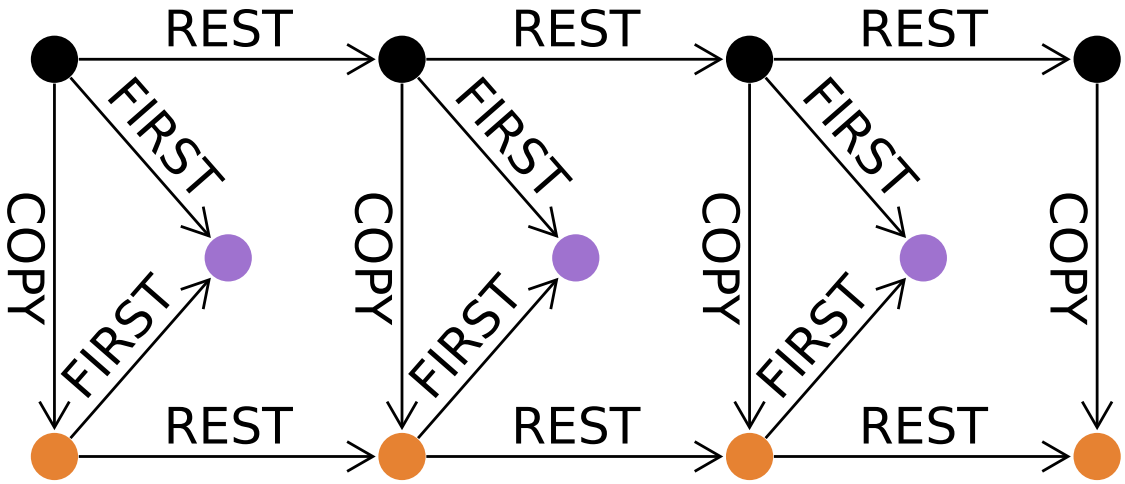
Types

```
cons-copy := list-copy & cons &
  [ FIRST #first,
    REST list-copy & [ COPY #rest,
                       NEXT #next ],
    COPY [ FIRST #first,
           REST #rest ],
    NEXT #next ].

null-copy := list-copy & null &
  [ COPY #next,
    NEXT #next ].
```


At the end of the list, #next needs to point to the new empty list.

cons-copy



The old list nodes are on top in black, the new list nodes are at the bottom in orange, and the elements of the list are in the middle in purple.

The elements are shared (`#first` constraint).

The lists line up (`#rest` constraint) – following REST.COPY will always be the same as following COPY.REST.