Discussion: TDL formalism potential extensions and new applications

DELPH-IN Summit 2013
Sankt Wendel

Overview

- `TDL` formalism is our common language, partially defines what we are and what we do
- Implemented in and interpreted by the processing systems: LKB, PET, ACE, AGREE, . . .

The DELPH-IN reference formalism

- A trimmed down version of TDL in comparison to e.g. (Krieger 1995)
- Clean, easy to implement, a good trade-off between expressiveness and computability
- No value disjunction: encourage generalization in the type inheritance hierarchy, or duplication/enumeration as separate instances (DNF)
- Makes certain implicit assumptions of the processing models

A.1 TDℒ Main Constructors

program → `begin :control.` { *type-def* | *instance-def* | *start* }* `end :control.` |
`begin :declare.` { *declare* | *start* }* `end :declare.` |
`begin :domain` *domain*`.` {*start*}* `end :domain` *domain*`.` |
`begin :instance.` { *instance-def* | *start* }* `end :instance.` |
`begin :lisp.` {*Common-Lisp-Expression*}* `end :lisp.` |
`begin :template.` { *template-def* | *start* }* `end :template.` |
`begin :type.` { *type-def* | *start* }* `end :type.`
domain → `'`*identifier* | `:`*identifier* | `"`*identifier*`"`
identifier → {a–z|A–Z|0–9|_|+|-|*|?}+

A.2 Type Definitions

type-def → *type* { *avm-def* | *subtype-def* } .
type → *identifier*
avm-def → `:=` *body* {`,` *option*}* |
`!=` *nonmonotonic* [`where (` *constraint* {`,` *constraint*}* `)`] {`,` *option*}*
body → *disjunction* [`-->` *list*] [`where (` *constraint* {`,` *constraint*}* `)`]
disjunction → *conjunction* { {`|` | `^`} *conjunction* }*
conjunction → *term* { `&` *term* }*
term → *type* | *atom* | *feature-term* | *diff-list* | *list* | *coreference* |
distributed-disj | *templ-par* | *templ-call* | `~`*term* | `(` *disjunction* `)`
atom → *string* | *integer* | `'`*identifier*
feature-term → `[` [*attr-val* {`,` *attr-val*}*] `]`
attr-val → *attribute* [`:`*restriction*] {`.` *attribute* [`:`*restriction*] [*disjunction*]}*
attribute → *identifier* | *templ-par*
restriction → *conj-restriction* { {`|` | `^`} *conj-restriction* }*
conj-restriction → *basic-restriction* { `&` *basic-restriction* }*
basic-restriction → *type* | `~`*basic-restriction* | *templ-par* | `(` *restriction* `)`
diff-list → `<!` [*disjunction* {`,` *disjunction*}*] `!>` [`:` *type*]
list → `<>` | `<` *nonempty-list* `>` [*list-restriction*]
nonempty-list → [*disjunction* {`,` *disjunction*}* `,`] `...` |
disjunction {`,` *disjunction*}* [`.` *disjunction*]
list-restriction → `:` `(` *restriction* `)` | `:` *type* [`:` (*integer*`,` *integer*) | `:` *integer*]
coreference → `#`*coref-name* | `~#(` *coref-name* {`,` *coref-name*}* `)`
coref-name → *identifier* | *integer*
distributed-disj → `%`*disj-name* `(` *disjunction* {`,` *disjunction*}+ `)`

disj-name → *identifier* | *integer*
templ-call → `@`*templ-name* `(` [*templ-par* {`,` *templ-par*}*] `)`
templ-name → *identifier*
templ-par → `$`*templ-var* [`=` *disjunction*]
templ-var → *identifier* | *integer*
constraint → `#`*coref-name* `=` { *function-call* | *disjunction* }
function-call → *function-name* `(` *disjunction* {`,` *disjunction*}* `)`
function-name → *identifier*
nonmonotonic → *type* `&` [*overwrite-path* {`,` *overwrite-path*}*]
overwrite-path → *identifier* { `.` *identifier* }* *disjunction*
subtype-def → { `:<` *type* }+ {`,` *option*}*
option → `status:` *identifier* | `author:` *string* | `date:` *string* | `doc:` *string* |
`expand-control:` *expand-control*
expand-control → `(` [`(:expand` { `(` {*type* | `(`*type* [*index* [*pred*]]`)`} {*path*}+ `)` }* `)` |
`(:expand-only` { `(` {*type* | `(`*type* [*index* [*pred*]]`)`} {*path*}+ `)` }* `)`] |
[`(:delay` { `(` {*type* | `(`*type* [*pred*]`)`} {*path*}+ `)` }* `)`] |
[`(:maxdepth` *integer* `)`] |
[`(:ask-disj-preference` {`t` | `nil`} `)`] |
[`(:attribute-preference` {*identifier*}* `)`] |
[`(:use-conj-heuristics` {`t` | `nil`} `)`] |
[`(:use-disj-heuristics` {`t` | `nil`} `)`] |
[`(:expand-function` {`depth` | `types`} `-first-expand` `)`] |
[`(:resolved-predicate` {`resolved-p` | `always-false` | `...`} `)`] |
[`(:ignore-global-control` {`t` | `nil`} `)`])
path → {*identifier* | *pattern*} {`.`{*identifier* | *pattern*}}*
pattern → `?` | `*` | `+` | `?`{*identifier*}[`?`|`*`|`+`]
pred → `eq` | `subsumes` | `extends` | ...
integer → {0|1|2|3|4|5|6|7|8|9}+
string → `"`{*any character*}*`"`
index → *integer* | *identifier* | *string*

A.3 Instance Definitions

instance-def → *instance avm-def* .
instance → *identifier*

Case Study I: SProUT

- `TDL` meets finite state techniques
- A grammar consists of pattern/action rules:
  - LHS: a regular expression over TFSs with functional operators and coreferences, representing the recognition pattern
  - RHS: a TFS specification of the output structure
- Small grammars targeting multilingual shallow processing
  - Named entity recognition
  - Information extraction
  - Ontology extraction
  - Opinion mining from text

Case Study II: Deependance

- Extends `TDL` with explicit disjunctions declarations
- Allows distributive/named disjunctions to model co-variation
- Adds *preference* distribution over disjunctive terms
- Needs to combine with best-first/non-exhaustive processing models to work efficiently
- Asserts different processing models (more data-driven and dependency-oriented)

Questions

- What's our general attitude towards variants/extensions of the formalism?
- Deeper integration of stochastic modelling
- Other applications of the `TDL`