A Control Language (Proposal) for Modular Termination Provers

Jörg Endrullis, VU Amsterdam
and Johannes Waldmann, HTWK Leipzig

Current State

- externally (from a user's viewpoint), termination provers are monolithic
- although internally, they certainly consist of components (e.g. Matchbox: simplex solver for additive weights, matrix solver, RFC match bound solver, loop finder).
- components cannot be re-used, this slows down progress (much energy wasted for re-inventing the wheel)

Goals

termination prover should allow

- direct access to, and control of, its components
- re-combination of its components
- combination with other provers' components

A Control Language

- abstract data type
  - leaves: elementary provers
  - branches: combinators
- semantics
- concrete syntax

A termination prover then is an interpreter for this language.
It could be a skeletal interpreter: it just handles the combinators and calls external provers on the leaves.

Semantics Domain—Now

what is the appropriate semantic domain?

- currently: `TRS -> IO (Maybe Bool)` where

  ```
  IO a : computation with result of type a
  data Maybe a = Nothing | Just a
  data Bool = False | True
  ```

- "certified": `TRS -> IO (Maybe (Bool, Proof))`

but this is not modular (such a prover is a "dead-end")

Modular Semantics Domain

```
type Prover =
     TRS -> IO ( Maybe ( [ TRS ], Proof ) )
```

with specification

```
  prover t -> Just ( [ t1, t2, .. tn ] , p )
    <==>  ( SN(t1) && .. && SN(tn) => SN(t) )
```

a successful "leaf" prover returns `Just ( [], p )`

this is naive, and cannot express:

- proofs of non-termination
- disjunction of sub-goals
- relative termination

Combinators for Provers

- `(p ‘orelse‘ q) = \ s ->`
  - execute `p s`, if successful, then this is result
  - else execute `q s`
- `(p ‘andthen‘ q) = \ s ->`
  - execute `p s`, if this fails, then this is result
  - if success `Just ( [s1, s2, ..] , p )`, then combine results of `q s1, q s2, ..`
- other combinators can be built from these:
  - `first = foldr1 orelse`
  - `sequential = foldr1 andthen`

Additional atomic combinators

- `parallel_or :: Prover->Prover->Prover`
  start both, when first result appears,
  kill the other process, return result
- `timed :: Seconds -> Prover -> Prover`
  run for at most the given time

may need more elaborate timer combinators
(e.g. "for half of the remaining time, do . . . ")

Combinator–Example

```
Sequential
  [ DP_Transform
  , Repeatedly ( First
      [ No_Strict_Rules, Simplex
      , Timed 10
        ( Matrix { method = Max_Plus
                 , dimension = 2, bits = 3 } )
      ] )  ]

first = foldr1 orelse
sequential = foldr1 andthen
repeatedly x = x `andthen` repeatedly x
```


Todo

control language developers:

- define the appropriate semantics domain
- define concrete syntax for output language
- identify the atomic combinators (few as possible) and their typical uses
- define concrete syntax for control language
- write skeletal interpreter (stand-alone)

prover authors:

- publish individual, conformant modules
- or allow to call individual modules within “monolithic” prover

Naive Concrete Syntax?

```
Parallel
  [ Sequential
      [ DP_Transform
      , Repeatedly ( First
          [ No_Strict_Rules, Simplex
          , Matrix { method = Max_Plus , dimension = 2, bits
          , Matrix { method = Max_Plus , dimension = 3, bits
          , Matrix { method = Max_Plus , dimension = 4, bits
          , Matrix { method = Max_Plus , dimension = 5, bits
          ] )
      ]
  , Sequential
      [ Reverse_Transform, DP_Transform
      , Repeatedly ( First ...
```


Flexible Concrete Syntax

- abstraction (subprograms, with parameters = lambda expressions)
- repetition (loops = list comprehensions)
- types: Prover, Integer (loop counter), (Time?)

```
matrices dim0 s t = first $ do
  dim <- dim0 : [ 1 .. 5 ]
  method <- [ top_half_strict maxplus, top_strict plustimes ]
  return $ timed ( 2 * dim^2 ) $ method dim s t
dp dim s t = sequential
  [ dp_transform , top dim s t, no_strict_rules ]
dpd dim s t = ( dp dim s t ) `por`
  ( reverse_transform `andthen` dp dim s t )
```


Embedded DSL?

- strategy expression = Haskell expression (strategy language = embedded domain specific language)
- Ideally, yes, but don’t want to deploy a complete Haskell system. Perhaps can use GHC(i) API (ghc as a library).
- “he who does not know (Haskell), is doomed to re-invent it—poorly.”
- e.g. allow some implicit abbreviations, homegrown macro processing or calling `cpp`

Elaboration: Statement type

(notes added after discussion at workshop)
“termination problem” consists of these components:

- signature
- four rule sets: all combinations of top/non-top, strict/non-strict
- Graph: directed graph on (top) rules
- strategy, e.g. “innermost w.r.t. rule set ...” (Aprove calls this Q)
- boolean minimality flag

semantics is termination of the implied rewrite rela-

Elaboration: Statement Semantics

- semantics is termination of the implied rewrite relation (“there are no infinite chains”)
- If a graph is present, extended by “...and the graph is a correct (i.e. over-) approximation of the possible connections of top rules (in infinite derivations)”
- missing components should have sensible defaults.
- not all possible combinations of components have sensible semantics (these are forbidden).

Elaboration: Transformers

(notes added after discussion at workshop)
most general output type of a transformer is a boolean combination of statements $O_1, \ldots$ (of the above form), and information of its relation to input ($\Rightarrow, \Leftarrow, \iff$).
e.g. $I \Leftarrow O_1 \vee O_2$
Remark by Rene (Aprove): this is not enough.
A transformer might produce the information
$(I \iff O_1) \wedge (I \iff O_2)$
which cannot be expressed by a formula with one isolated I and one arrow.