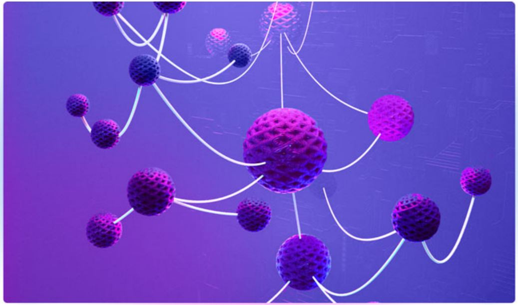
Meta-Reasoning in MeTTa

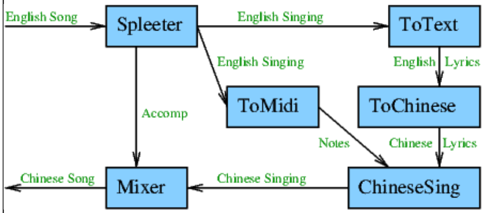
for Inference Control via Provably Pruning the Search Tree

Nil Geisweiller

SingularityNET

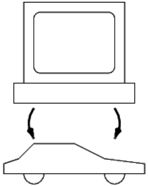
Artificial Intelligence and Theorem Proving 2024 (AITP-24)

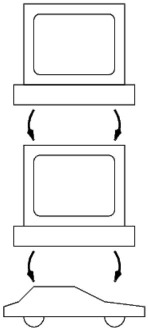


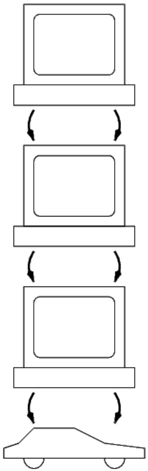


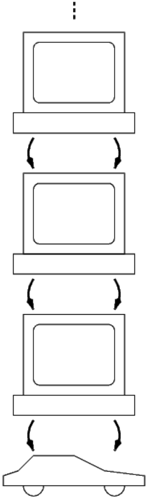

```
(-> (: $s (SongIn "English")) (SongIn "Chinese"))
```

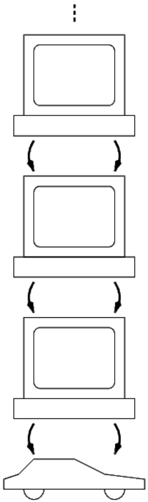
- MeTTa
- Meta-reasoning



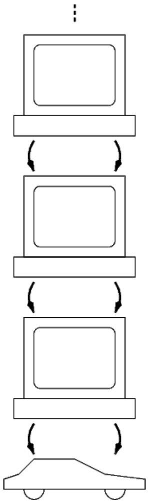






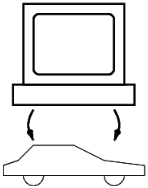


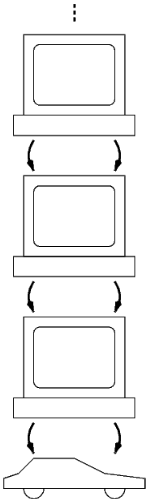
Gödel Machine *(Jürgen Schmidhuber, 2003)*



Gödel Machine *(Jürgen Schmidhuber, 2003)*

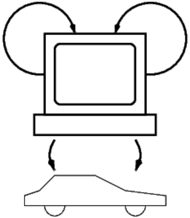
- Merge all machines into one.

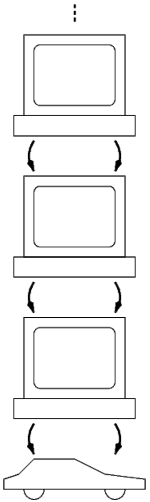




Gödel Machine *(Jürgen Schmidhuber, 2003)*

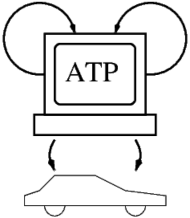
- Merge all machines into one.
- Internal actions to action space.





Gödel Machine *(Jürgen Schmidhuber, 2003)*

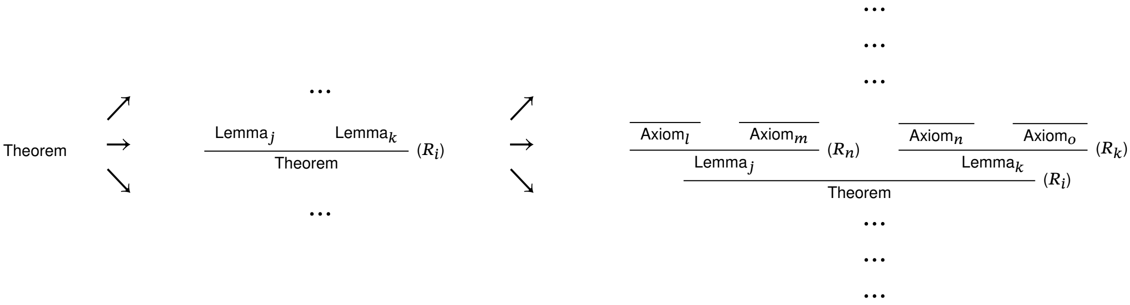
- Merge all machines into one.
- Internal actions to action space.
- Mathematical proof ⇒ trigger action.



Theorem

$$\text{Theorem} \quad \longrightarrow \quad \frac{\text{Lemma}_j \qquad \text{Lemma}_k}{\text{Theorem}} \; (R_i)$$

$$\text{Theorem} \quad \longrightarrow \quad \dfrac{\text{Lemma}_j \qquad \text{Lemma}_k}{\text{Theorem}}\,(R_i) \quad \longrightarrow \quad \dfrac{\dfrac{\overline{\text{Axiom}_l} \qquad \overline{\text{Axiom}_m}}{\text{Lemma}_j}\,(R_n) \qquad \dfrac{\overline{\text{Axiom}_n} \qquad \overline{\text{Axiom}_o}}{\text{Lemma}_k}\,(R_k)}{\text{Theorem}}\,(R_i)$$


Theorem
Lemma$_j$
Lemma$_k$
Theorem
(R_i)
Axiom$_l$
Axiom$_m$
(R_n)
Axiom$_n$
Axiom$_o$
(R_k)
Lemma$_j$
Lemma$_k$
(R_i)
Theorem

$$\text{Theorem} \quad \begin{matrix} \nearrow R_1 \\ \\ \searrow R_2 \end{matrix}$$

$$\frac{\text{Lemma}_j \qquad \text{Lemma}_k}{\text{Theorem}} \ (R_1)$$

$$\frac{\text{Lemma}_l \qquad \text{Lemma}_m}{\text{Theorem}} \ (R_2)$$

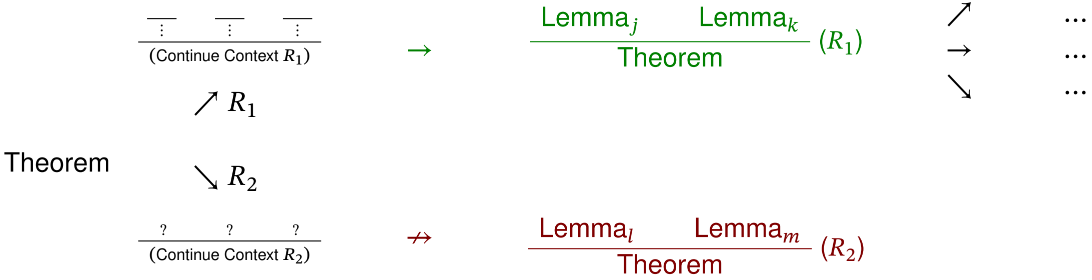
$$\frac{\overline{}\ \vdots \quad \overline{}\ \vdots \quad \overline{}\ \vdots}{(\text{Continue Context } R_1)} \rightarrow \frac{\text{Lemma}_j \qquad \text{Lemma}_k}{\text{Theorem}}\ (R_1)$$

$$\nearrow R_1$$

Theorem

$$\searrow R_2$$

$$\frac{? \qquad ? \qquad ?}{(\text{Continue Context } R_2)} \not\rightarrow \frac{\text{Lemma}_l \qquad \text{Lemma}_m}{\text{Theorem}}\ (R_2)$$


(Continue Context R_1)
R_1
Theorem
R_2
? ? ?
(Continue Context R_2)
Lemma$_j$ Lemma$_k$
Theorem
(R_1)
...
...
...
Lemma$_l$ Lemma$_m$
Theorem
(R_2)

MeTTa: Meta Type Talk

- Functional and logic programming
- Non-determinism (like Curry)
- Unification (like Prolog)
- Gradual typing
- Self-modifiable
- Concurrency
- Scalable

```
;; Bit strings
(= (bits Z) Nil)
(= (bits (S $k)) (Cons 0 (bits $k)))
(= (bits (S $k)) (Cons 1 (bits $k)))

;; Generate all 3-bit strings
!(bits (S (S (S Z))))
```

⇓

```
[(Cons 0 (Cons 0 (Cons 0 Nil)))
 (Cons 0 (Cons 0 (Cons 1 Nil)))
 (Cons 0 (Cons 1 (Cons 0 Nil)))
 (Cons 0 (Cons 1 (Cons 1 Nil)))
 (Cons 1 (Cons 0 (Cons 0 Nil)))
 (Cons 1 (Cons 0 (Cons 1 Nil)))
 (Cons 1 (Cons 1 (Cons 0 Nil)))
 (Cons 1 (Cons 1 (Cons 1 Nil)))]
```

```
;; Backward chainer
(= (bc $kb $_ (: $prf $ccln)) (match $kb (: $prf $ccln) (: $prf $ccln)))
(= (bc $kb (S $k) (: ($prfabs $prfarg) $ccln))
   (let* (((: $prfabs (-> $prms $ccln)) (bc $kb $k (: $prfabs (-> $prms $ccln))))
          ((: $prfarg $prms) (bc $kb $k (: $prfarg $prms))))
     (: ($prfabs $prfarg) $ccln)))
```

```
;; Backward chainer
(= (bc $kb $_ (: $prf $ccln)) (match $kb (: $prf $ccln) (: $prf $ccln)))
(= (bc $kb (S $k) (: ($prfabs $prfarg) $ccln))
   (let* (((: $prfabs (-> $prms $ccln)) (bc $kb $k (: $prfabs (-> $prms $ccln))))
          ((: $prfarg $prms) (bc $kb $k (: $prfarg $prms))))
     (: ($prfabs $prfarg) $ccln)))

;; Knowledge base
!(bind! &kb (new-space))
!(add-atom &kb (: AK (-> $a (-> $b $a))))
!(add-atom &kb (: AS (-> (-> $a (-> $b $c)) (-> (-> $a $b) (-> $a $c)))))
```

```
;; Backward chainer
(= (bc $kb $_ (: $prf $ccln)) (match $kb (: $prf $ccln) (: $prf $ccln)))
(= (bc $kb (S $k) (: ($prfabs $prfarg) $ccln))
   (let* (((: $prfabs (-> $prms $ccln)) (bc $kb $k (: $prfabs (-> $prms $ccln))))
          ((: $prfarg $prms) (bc $kb $k (: $prfarg $prms))))
     (: ($prfabs $prfarg) $ccln)))

;; Knowledge base
!(bind! &kb (new-space))
!(add-atom &kb (: AK (-> $a (-> $b $a))))
!(add-atom &kb (: AS (-> (-> $a (-> $b $c)) (-> (-> $a $b) (-> $a $c)))))

;; Query
!(bc &kb (S (S Z)) (: $prf (-> $a $a)))
```

```
;; Backward chainer
(= (bc $kb $_ (: $prf $ccln)) (match $kb (: $prf $ccln) (: $prf $ccln)))
(= (bc $kb (S $k) (: ($prfabs $prfarg) $ccln))
   (let* (((: $prfabs (-> $prms $ccln)) (bc $kb $k (: $prfabs (-> $prms $ccln))))
          ((: $prfarg $prms) (bc $kb $k (: $prfarg $prms))))
     (: ($prfabs $prfarg) $ccln)))

;; Knowledge base
!(bind! &kb (new-space))
!(add-atom &kb (: AK (-> $a (-> $b $a))))
!(add-atom &kb (: AS (-> (-> $a (-> $b $c)) (-> (-> $a $b) (-> $a $c)))))

;; Query
!(bc &kb (S (S Z)) (: $prf (-> $a $a)))

;; Results
[(: ((AS AK) AK) (-> $a $a))
 ...]
```

```
;; Backward chainer with dependent types and lambda abstraction

;;;;;;;;;;;;;;;;
;; Base cases ;;
;;;;;;;;;;;;;;;;

;; Match the knowledge base
(= (bc $kb $env $idx $_ (: $prf $thrm))
   (match $kb (: $prf $thrm) (: $prf $thrm)))
;; Match the environment
(= (bc $kb $env $idx $_ (: $prf $thrm))
   (match' $env (: $prf $thrm) (: $prf $thrm)))

;;;;;;;;;;;;;;;;;;;;;
;; Recursive steps ;;
;;;;;;;;;;;;;;;;;;;;;

;; Proof application
(= (bc $kb $env $idx (S $k) (: ($prfabs (: $prfarg $prms)) $thrm))
   (let* (((: $prfabs (-> (: $prfarg $prms) $thrm))
            (bc $kb $env $idx $k (: $prfabs (-> (: $prfarg $prms) $thrm))))
           ((: $prfarg $prms)
            (bc $kb $env $idx $k (: $prfarg $prms))))
     (: ($prfabs (: $prfarg $prms)) $thrm)))
;; Proof abstraction
(= (bc $kb $env $idx (S $k) (: (\ $idx $prfbdy) (-> (: $idx $prms) $thrm)))
   (let (: $prfbdy $thrm)
     (bc $kb (Cons (: $idx $prms) $env) (s $idx) $k (: $prfbdy $thrm))
     (: ( $idx $prfbdy) (-> (: $idx $prms) $thrm))))
```

```
;; Equality is transitive
!(add-atom &kb (: Trans (-> (: $prf1 (=== $x $y))      ; Premise 1
                        (-> (: $prf2 (=== $y $z)) ; Premise 2
                            (=== $x $z)))))       ; Conclusion

;; Equality is symmetric
!(add-atom &kb (: Sym (-> (: $prf (=== $x $y)) ; Premise
                      (=== $y $x)))))         ; Conclusion

;; Equality respects function application
!(add-atom &kb (: Cong (-> (: $f (-> (: $_ $a) $b))            ; Premise 1
                       (-> (: $x $a)                         ; Premise 2
                           (-> (: $x' $a)                    ; Premise 3
                               (-> (: $prf (=== $x $x'))     ; Premise 4
                                   (=== ($f $x) ($f $x')))))))) ; Conclusion

;; Rule of replacement
!(add-atom &kb (: Replace (-> (: $prf1 (=== $x $x')) ; Premise 1
                          (-> (: $prf2 $x)         ; Premise 2
                              $x')))))             ; Conclusion

;; Define double
!(add-atom &kb (: double (-> (: $k N) N)))
!(add-atom &kb (: double_base (=== (double (: Z N)) Z)))
!(add-atom &kb (: double_rec (-> (: $k N)
                          (=== (double (: (S (: $k N)) N)) (S (: (S (: (double (: $k N)) N)) N))))))

...
```

```
;; Query: find proof that for any natural k, (double k) is even
!(bc &kb Nil z (fromNumber 11) (: $prf (-> (: $k N) (Even (double (: $k N))))))

;; Results
[(: ((SIN ((Replace ((((Cong Even) Z) (double Z)) (Sym double_base)))))
     (\ z (\ (s z) ((Replace (Sym ((((Cong Even) (double (S z))) (S (S (double z)))) (double_rec z))))
                   (MkEvenSS (s z))))))
    (-> (: $k N) (Even (double (: $k N)))))
 ...]
```

```
                                                                                                                           -(z)
                                                                                                                           N
                                                                                                               -(z)        -(double)
                                                                                                               N           N
                                                                                                               -(S)        -(S)                  -(z)
                                                                                                               N           N                     N
                                                                                     ------------------(Even)  -(double)   -(S)   ----------------------------------------(double_rec)
                                                                                     (-> (: $_ N) Type)        N           N      (=== (double (S z)) (S (S (double z))))
                                                                                     ------------------------------------------------------------------------------------(Cong)      -----------------(s z)
                                                                                                                 (=== (Even (double (S $k))) (Even (S (S (double $k)))))             (Even (double z))
                         -(Z)        ------------------(double_base)                                             ---------------------------------------------------------(Sym)  -------------------------(MkEvenSS)
                         N           (=== (double Z) Z)                                                          (=== (Even (S (S (double $k)))) (Even (double (S $k))))         (Even (S (S (double z))))
----------(Even)   -(Z)  -(double)   ------------------(Sym)                            -----------------(s z)   -----------------------------------------------------------------------------------------(Replace)
(-> N Type)        N     N           (=== Z (double Z))                                 (Even (double z))                  (Even (double (S z)))
-------------------------------------------------------(Cong)  -------(MkEvenZ)   -(z)  --------------------------------------------------------(\)
           (=== (Even Z) (Even (double Z)))                    (Even Z)           N     (-> (: (s z) (Even (double z))) (Even (double (S z))))
           ------------------------------------------------------------(Replace)  ----------------------------------------------------------------------(\)
                     (Even (double Z))                                            (-> (: z N) (-> (: (s z) (Even (double z))) (Even (double (S z)))))
                     -----------------------------------------------------------------------------------------------------------------------------------(SIN)
                                                               (-> (: $k N) (Even (double $k)))
```

```
;; Backward chainer
(= (bc $kb $_ (: $prf $ccln)) (match $kb (: $prf $ccln) (: $prf $ccln)))
(= (bc $kb (S $k) (: ($prfabs $prfarg) $ccln))
   (let* (((: $prfabs (-> $prms $ccln)) (bc $kb $k (: $prfabs (-> $prms $ccln))))
          ((: $prfarg $prms) (bc $kb $k (: $prfarg $prms))))
     (: ($prfabs $prfarg) $ccln)))
```

↓

```
;; Backward chainer with control (conditionals + context updaters)
(= (bc $kb (MkControl $absupd $argupd $bcont $rcont $mcont) $ctx (: $prf $ccln))
   (if ($bcont (: $prf $ccln) $ctx)
       (match $kb (: $prf $ccln) (if ($mcont (: $prf $ccln) $ctx) (: $prf $ccln) (empty)))
       (empty)))
(= (bc $kb (MkControl $absupd $argupd $bcont $rcont $mcont) $ctx (: ($prfabs $prfarg) $ccln))
   (if ($rcont (: ($prfabs $prfarg) $ccln) $ctx)
       (let* (((: $prfabs (-> $prms $ccln))
                (bc $kb (MkControl $absupd $argupd $bcont $rcont $mcont)
                    ($absupd (: ($prfabs $prfarg) $ccln) $ctx) (: $prfabs (-> $prms $ccln))))
               ((: $prfarg $prms)
                (bc $kb (MkControl $absupd $argupd $bcont $rcont $mcont)
                    ($argupd (: ($prfabs $prfarg) $ccln) $ctx) (: $prfarg $prms))))
         (: ($prfabs $prfarg) $ccln))
       (empty)))
```

```
;; January precedes February, which precedes Mars, etc.
!(add-atom &kb (: JF (<= Jan Feb)))
!(add-atom &kb (: FM (<= Feb Mar)))
!(add-atom &kb (: MA (<= Mar Apr)))
!(add-atom &kb (: AM (<= Apr May)))
!(add-atom &kb (: MJ (<= May Jun)))
!(add-atom &kb (: JJ (<= Jun Jul)))
!(add-atom &kb (: JA (<= Jul Aug)))
!(add-atom &kb (: AS (<= Aug Sep)))
!(add-atom &kb (: SO (<= Sep Oct)))
!(add-atom &kb (: ON (<= Oct Nov)))
!(add-atom &kb (: ND (<= Nov Dec)))

;; Precedence is non strict, i.e. reflexive
!(add-atom &kb (: Refl (<= $x $x)))

;; Precedence is transitive
!(add-atom &kb (: Trans (-> (<= $x $y)
                            (-> (<= $y $z)
                                (<= $x $z)))))

;; Shortcut rule: January precedes all months
!(add-atom &kb (: JPA (<= Jan $x)))
```

```
;; 1st observation: if
;; - the target theorem is (<= x x)
;; - the current proof is Refl
;; then continue.
!(add-atom &ctl-kb (: RS (Continue (: Refl $r) (MkTD (<= $x $x) $k))))

;; 2nd observation: if
;; - the target theorem is (<= Jan x)
;; - the current proof is JPA
;; then continue.
!(add-atom &ctl-kb (: JS (Continue (: JPA $r) (MkTD (<= Jan $x) $k))))

;; 3rd observation: if
;; - the target theorem is (<= $x $y) such that $x != Jan and $x != $y
;; - the current proof is Trans or FM to ND
;; then continue.
!(let $rn (superpose (Trans FM MA AM MJ JJ JA AS SO ON ND))
   (add-atom &ctl-kb (: TS (-> (!= Jan $x)
                            (-> (!= $x $y)
                                (Continue (: $rn $rc)
                                          (MkTD (<= $x $y) $k)))))))

;; Backward chainer as continuation condition.  Return True iff a
;; proof of continuation is found.
(: td-continuor (-> $a $ct Bool))
(= (td-continuor $query $ctx)
   (let $results (collapse (bc &ctl-kb &ctl (S (S 2)) (: $prf (Continue $query $ctx)))
     (not (== () $results))))
```

- Discover statistical patterns via reasoning.
- Formalize learning to reason about it.
- Use more universal control theory.
- Unify problem and control theory.

metta-lang.dev
github.com/trueagi-io/chaining