Tuning for Performance

- Outline of talk:
  - Classifier Performance
  - Recognizer Performance
  - Performance Tips
  - CLOS Instances and the Backchainer
In some systems, slow performance is due to poorly-tuned code.

In Loom, slow performance can result from the enormous amount of inferencing that occurs under the hood.
Classifier Performance

- Classifier Phases

  1. normalization (compute closure of ~100 inference rules)
  2. classification (compute subsumption links — very fast)
  3. completion (normalize constraints)
  4. sealing (compile access functions)
Classifier Performance

- Classifier Phases
  1. normalization
  2. classification
  3. completion
  4. sealing

- Bulk of time is spent in phases (1) and (3), normalizing features:
  1. start with local features (:at-most, :at-least, :all, ...);
  2. inherit features from parent concepts;
  3. compute larger set of features (deductive closure);
  4. keep only the most specific features;
  5. classify the remaining features.
Speeding Up Normalization

- Each constraint in Loom represents a rule of inference (not just a type check).
- The overhead of normalization depends on the number of features per concept (it’s estimated to be quadratic in the number of features).
- So, a simple way to speed up an application is to specify fewer constraints :-).
Loom permits you to lobotomize the classifier

- “(power-level :medium)” causes Loom to ignore a few of the most expensive normalization rules.
- “(power-level :low)” causes Loom to make a single pass over the normalization rules (rather than computing their closure).
Load-Time vs. Run-Time Classification

Most applications perform the bulk of classification at load time; for them, speed of classification may not be critical.

- Normally, run-time production of new system-generated descriptions will quiesce (no more “.”s and “+”s);
An explicit call by an application (e.g., (tellm)) triggers reclassification of updated instances.

Recognition strategy:

- For each instance on the queue
  - (1) normalize asserted and inherited features;
  - (2) classify the instance;
  - (3) install dependency bombs (TMS monitors);
  - (4) test for incoherence;
  - (5) propagate forward constraints.

Steps 1-5 are applied to each instance at least two times (once each in strict and default mode).
Classifying Instances

During the recognition process, each feature in a concept definition represents a miniature query.

Examples:

(:at-least k R)
Retrieve fillers of the role R;
Succeed if the number of fillers is at least k.

(:at-most k R)
If role R is closed, retrieve fillers of the role R;
Succeed if the number of fillers is at most k.

(:all R A)
If role R is closed, retrieve fillers of the role R;
Succeed if each of the fillers satisfies the concept A.

The bulk of recognition time consists of computing feature satisfaction and truth maintaining the results.
Testing for Closed Roles

- Probing features such as (:all R A) or (:at-most k R) usually entails proving that the role R is closed.
- This test is fast if
  - R has the :closed-world property, or
  - R is :single-valued and a role filler exists.

Tip: Always specify the :single-valued and :closed-world properties on relations whenever they are valid for your application domain.
Subtlety in the semantics of role closure:

(defconcept A
  :implies (:at-least 1 R))
(defrelation R
  :characteristics (:closed-world))
tell (Thing Joe)
  (A Fred))

• The role "(R of Joe)" is closed, but the role "(R of Fred)" is not closed.
Domain and Range Constraints

- **Tip**: Always specify domain and range constraints for a relation (unless they are inherited from a parent relation).

(defrelation R :domain A :range B)
(tellm (R Fred Joe))

→ Loom infers that Fred satisfies A and that Joe satisfies B.

(defconcept A :implies (:exactly 1 R))
(defrelation R :domain A)

→ Loom infers that R is :single-valued.
Performance Warnings

- A "no generator found" performance warning indicates that a query will exhibit abysmal performance.

  - Slower (sometimes):
    `(retrieve (?x ?y) (R ?x ?y))`

  - Faster (sometimes):
    `(retrieve (?x ?y)
      (and (A ?x) (R ?x ?y)))`

- If no domain is specified for R, the slower query will scan the entire kb to generate bindings for ?x.
Performance Tips:

- **Tip**: Always rephrase definitions or queries to eliminate performance warnings.

- **Tip**: Never wrap an eval around an `ask` or `retrieve` unless you are single, childless, and have no desire to graduate, e.g.,
  
  ```lisp
  (eval `(retrieve (?y) (and (R ,foo ?y) (A ?y)))
  ```

- **Tip**: To programmatically compose a query on the fly, use “query” or bind variables:
  
  ```lisp
  (query `(?y) `(and (R ,foo ?y) (A ?y)))
  ```

  ```lisp
  (let ((?x foo))
    (retrieve (?y) (and (R ?x ?y) (A ?y))))
  ```

Better!
**perfect relations**

- Marking a concept or relation :perfect tells Loom that facts about it cannot change.

  - Tip: Use of the :perfect properties reduces match overhead.

  - Tip: Computed relations are prime candidates for the :perfect attribute.

```lisp
(defunrelation <>
  :domain Number :range Number
  :characteristics (:symmetric :perfect)
  :predicate /= )
```
How to Get No Recognition

- The overhead of instance classification (recognition) is eliminated if you specify as a creation policy :clos-instance or :lite-instance.
- Deduction over CLOS instances and LITE instances is backward chained, with no caching.
- However (there is always a catch) inference without instance classification is strictly weaker than inference with it.
Deduction with CLOS and LITE Instances

- With creation policy set to :clos-instance or :lite-instance inference is performed using backward chaining.

- The backchainer recognizes rules of the form

  \[(\text{implies } A \ B) \quad \text{and} \quad (\text{implies } \text{<description>} \ B)\]

  but ignores rules of the form

  \[(\text{implies } A \ \text{<descriptions>})\]
Backward chaining and type restrictions

- The design decision not to chain backwards across value restrictions was a judgment call.

(defconcept A
   :implies (:all R B))
(tell (A Fred) (R Fred Joe))
(ask (B Joe))  -->  ???

- The recognizer will prove that Joe satisfies B; the backchainer won't.