Computing and Proving Well-founded Orderings through Finite Abstractions

Rob Sumners

Centaur Technology

ACL2 Workshop 2020

Jumping In!

- **Goal**: prove a relation `(r x y)` is well-founded.
  - In particular:
    - Prove relations arising from state machines are well-founded.
- **Value**: a well-founded relation can admit recursive functions.
  - In particular:
    - Show that state machines make progress to completion.
- **Approach**: find a measure which decreases when `(r x y)`
  - In particular:
    1. Build an abstract model of `(r x y)`
    2. Check for measure existence on abstract model
    3. Build and prove corresponding concrete measure
  - Automate as much as possible!

Well-founded relations in ACL2

- ACL2 axiomatizes ordinals up to epsilon-0.
  - Recognizer: `(o-p x)`, Order: `(o< x y)`
  - Used to define measures to admit recursive functions..
  - ..which in turn provides schemes for proof by induction.

```
(encapsulate (((r * *) => *) ((m *) => *))
     ...
  (defthm m-is-an-ordinal (o-p (m x)))
  (defthm m-is-o<-when-r
    (implies (r x y) (o< (m y) (m x)))))
```

- Any sequence of objects related by `R` must terminate:

```
(defchoose choose-r (y) (x) (r x y))

(defun seq-r (x)
  (declare (xargs :measure (m x)))
  (if (r x (choose-r x)) (seq-r (choose-r x)) x))
```

Example: Bakery Algorithm - introduction

- Concurrent algorithm for solving mutual exclusion
- A number of parallel tasks each executing a number of jobs..
- Where each job goes through the following steps:
  1. **choosing:**
     - pick the next `pos` in line from a shared variable
     - wait to flush previous jobs if `pos` cycles back to 0.
  2. **pos-valid:**
     - atomic compare-and-swap to update shared variable
  3. **wait-in-line:**
     - wait for other tasks which are still choosing
     - wait for other tasks with a lower `pos` in line
  4. **critical-section**

Example: Bakery Algorithm - definition

Algorithm Bakery Task

```
 1: for every job do
 2:     choosing ← 't
 3:     temp ← shared.max
 4:     pos ← temp + 1
 5:     for every task do
 6:         wait if pos = 0 ∧ task.pos-valid
 7:     pos-valid ← 't
 8:     if (shared.max ≤ temp) shared.max ← pos
 9:     choosing ← 'nil
10:     for every task do
11:         wait if pos ≠ 0 ∧ task.choosing
12:         wait if lex<(task.pos, task.id, pos, id)
13:     [critical section], pos-valid ← 'nil
14: done
```

Example: Bakery Algorithm – blocking relation

Algorithm Bakery Task

```
 1: for ... do
 2:     ...
 3:     ...
 4:     ...
 5:     for ... do
 6:         wait if pos = 0 ∧ task.pos-valid
 7:     ...
 8:     ...
 9:     ...
10:     for ... do
11:         wait if pos ≠ 0 ∧ task.choosing
12:         wait if lex<(task.pos, task.id, pos, id)
13:     ...
14: ...
```

Example: Bakery Algorithm – next-state relation

Algorithm Bakery Task

```
 1: for every job do
 2:     choosing ← 't
 3:     temp ← shared.max
 4:     pos ← temp + 1
 5:     for every task do
 6:         ...
 7:     pos-valid ← 't
 8:     if (shared.max ≤ temp) shared.max ← pos
 9:     choosing ← 'nil
10:     for every task do
11:         ...
12:         ...
13:     [critical section], pos-valid ← 'nil
14: done
```

Example: Bakery Algorithm – our goal - 1

- Admit the function `bake-run` which..
  - steps bakery task states until all are done and where..
  - `choose-ready` is constrained to pick a task index which is not done and not blocked

```
(defun bake-run (tasks shared oracle)
  (if (bake-all-tasks-done tasks)
      (mv tasks shared)
    (let* ((n (choose-ready tasks shared oracle))
           (a (nth n tasks)))
      (bake-run (update-nth n (bake-t-next a shared) tasks)
                (bake-sh-next shared a)
                (next-oracle oracle)))))
```

Example: Bakery Algorithm – our goal - 2

- In order to admit `bake-run`, we will need to show two relations are well-founded:
  1. `(r x y)` is the bakery next-state relation
     - where `y` is not a done state.
  2. `(r x y)` is the bakery blocking relation.
     - i.e. `x` is waiting for `y`.
- Instances of the measure for the next-state relation are combined to define a measure for admitting `bake-run`
- Measure for the blocking relation is needed to admit a function supporting the witness function for `choose-ready`

Our Approach

- **Approach**: build a measure which decreases when `(r x y)`
  1. Build an abstract model of `(r x y)`
     - User defines a mapping from concrete (finite) domain to abstract values
     - Use bit-blasting and incremental SAT to compute abstract graph of `R`
     - Tag arcs in abstract graph with additional component ordering information
  2. Check for measure existence on abstract model
     - Use graph algorithms on abstract model to either find a descriptor for a measure for `R`..
     - ..or a witness showing that `R` is (possibly) not well-founded.
  3. Build and prove corresponding concrete measure
     - Instantiate a theory to transfer the abstract measure descriptor to a definition of a measure for showing `R` is well-founded.

Build an abstract model - 1

- Given a mapping from bakery task states to abstract values..

```
(define bake-map ((a bake-task-p))
  (b* (((bake-task a) a))
    `((:loc    ,a.loc)
      (:done   ,a.done)
      (:task=0 ,(equal a.task 0))
      (:job=0  ,(equal a.job 0))
      (:inv    ,(and (>= a.task 0) (>= a.job 0))))))
```

- `(bake-map a)` maps bakery task state a to the control-flow components:
  - `:loc` – current program location
  - `:done` – is the task in a done state
  - `:task=0` – is the task counter equal to 0 (ends inner for-loop)
  - `:job=0` – is the job counter equal to 0 (ends outer for-loop)
  - `:inv` – inductive invariant included in abstract value

Build an abstract model - 2

- Given a mapping from bakery task states to abstract values..

```
(define bake-map ((a bake-task-p))
  (b* (((bake-task a) a))
    `((:loc    ,a.loc)
      (:done   ,a.done)
      (:task=0 ,(equal a.task 0))
      (:job=0  ,(equal a.job 0))
      (:inv    ,(and (>= a.task 0) (>= a.job 0))))))
```

- ..use bit-blasting/SAT to compute (reachable) abstract values:

```
(defconsts (*bake-reach* state)
  (comp-map-reach
    :init-hyp `t
    :init-trm `(bake-map (bake-t-init n r))
    :step-hyp `(and (equal (bake-map a) ,*src-var*)
                    (not (bake-task->done a)))
    :step-trm `(bake-map (bake-t-next a sh))))
```

Build an abstract model – 3

```
> (strip-cars *bake-reach*)
(((:LOC 1)  (:DONE NIL) (:TASK=0 T)   (:JOB=0 NIL) (:INV T))
 ((:LOC 2)  (:DONE NIL) (:TASK=0 T)   (:JOB=0 NIL) (:INV T))
 ((:LOC 3)  (:DONE NIL) (:TASK=0 T)   (:JOB=0 NIL) (:INV T))
 ((:LOC 4)  (:DONE NIL) (:TASK=0 T)   (:JOB=0 NIL) (:INV T))
 ((:LOC 5)  (:DONE NIL) (:TASK=0 NIL) (:JOB=0 NIL) (:INV T))
 ((:LOC 6)  (:DONE NIL) (:TASK=0 NIL) (:JOB=0 NIL) (:INV T))
 ((:LOC 6)  (:DONE NIL) (:TASK=0 T)   (:JOB=0 NIL) (:INV T))
 ((:LOC 7)  (:DONE NIL) (:TASK=0 T)   (:JOB=0 NIL) (:INV T))
 ((:LOC 8)  (:DONE NIL) (:TASK=0 T)   (:JOB=0 NIL) (:INV T))
 ((:LOC 9)  (:DONE NIL) (:TASK=0 T)   (:JOB=0 NIL) (:INV T))
 ((:LOC 10) (:DONE NIL) (:TASK=0 NIL) (:JOB=0 NIL) (:INV T))
 ((:LOC 11) (:DONE NIL) (:TASK=0 NIL) (:JOB=0 NIL) (:INV T))
 ((:LOC 12) (:DONE NIL) (:TASK=0 NIL) (:JOB=0 NIL) (:INV T))
 ((:LOC 12) (:DONE NIL) (:TASK=0 T)   (:JOB=0 NIL) (:INV T))
 ((:LOC 13) (:DONE NIL) (:TASK=0 T)   (:JOB=0 NIL) (:INV T))
 ((:LOC 13) (:DONE NIL) (:TASK=0 T)   (:JOB=0 T)   (:INV T))
 ((:LOC 14) (:DONE T)   (:TASK=0 T)   (:JOB=0 T)   (:INV T)))
```

Build an abstract model – 4

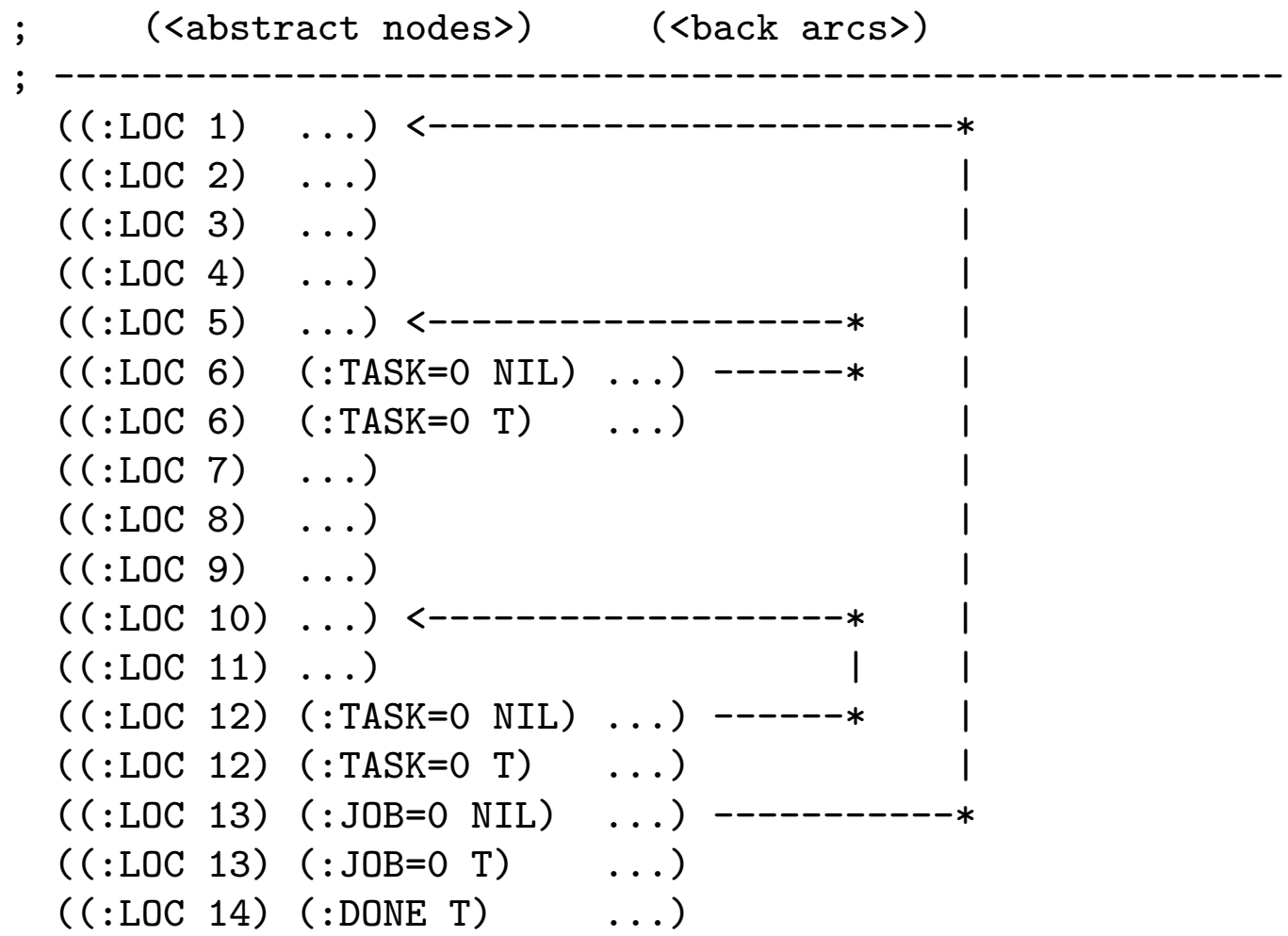
- To avoid enumerating `job` and `task` counter values within the abstract graph, we need to add these counters as **submeasures**:
  - Tag the arcs in the graph based on whether or not these submeasures strictly decrease or possibly increase
- For our example abstract model, we need two submeasures – the `job` and `task` loop counters:
  - The `job` counter never increases and decreases at the end of the outer loop
  - The `task` counter increases before going into the inner loops and decreases at the end of the inner loops.

Checking for a measure – 1

- Given the (tagged) abstract graph, we search for a measure descriptor using the following graph algorithm:
  - if the current subgraph is an SCC:
    1. search for submeasure that never increases and decreases once.
    2. if no such submeasure, then find witness cycle and fail.
    3. otherwise, remove arcs where submeasure decreases and recur.
    4. `cons` submeasure onto measure descriptors from recursion.
  - otherwise:
    1. partition the graph into SCCs using a standard algorithm.
    2. recursively build measure descriptors for each of the SCCs.
    3. build the DAG of SCCs and enumerate the SCCs in the DAG.
    4. `cons` enumeration onto measure descriptors from recursion.

Checking for a measure – 2

- Abstract graph with backwards arcs for inner and outer loops:



Checking for a measure – 3

- Computed measure descriptor on (tagged) abstract graph:

```
;                (<abstract node>) . (<measure-descriptor>)
; ------------------------------------------------------
  ((:LOC 1)  ...)                   . (3 JOB 11)
  ((:LOC 2)  ...)                   . (3 JOB 10)
  ((:LOC 3)  ...)                   . (3 JOB 9)
  ((:LOC 4)  ...)                   . (3 JOB 8)
  ((:LOC 5)  ...)                   . (3 JOB 7 TASK 1)
  ((:LOC 6)  (:TASK=0 NIL) ...)     . (3 JOB 7 TASK 2)
  ((:LOC 6)  (:TASK=0 T)   ...)     . (3 JOB 6)
  ((:LOC 7)  ...)                   . (3 JOB 5)
  ((:LOC 8)  ...)                   . (3 JOB 4)
  ((:LOC 9)  ...)                   . (3 JOB 3)
  ((:LOC 10) ...)                   . (3 JOB 2 TASK 2)
  ((:LOC 11) ...)                   . (3 JOB 2 TASK 1)
  ((:LOC 12) (:TASK=0 NIL) ...)     . (3 JOB 2 TASK 3)
  ((:LOC 12) (:TASK=0 T)   ...)     . (3 JOB 1)
  ((:LOC 13) (:JOB=0 NIL)  ...)     . (3 JOB 12)
  ((:LOC 13) (:JOB=0 T)    ...)     . (2)
  ((:LOC 14) (:DONE T)     ...)     . (1)
```

Building the concrete measure

- Given the computed measure descriptor, define a measure demonstrating a well-founded relation.
- We prove a "theory" in which:
  - we assume the properties of the abstract graph construction and computed measure and..
  - we prove that a derived function is a measure showing the relation is well-founded.
- Simply "instantiate" this theory to get the desired result of a sufficient measure
- NOTE – we use a book of *bounded ordinals* which afford better composition of ordinals.
  - Importantly.. allows measure of one relation to be submeasure of another relation..

Pulling it together!

- Goal: admit the function `bake-run`..

```
(defun bake-run (tsks shrd orcl)
  (if (bake-all-done tsks)
      (mv tsks shrd)
    (let* ((n (choose-ready tsks shrd orcl))
           (a (nth n tsks)))
      (bake-run (update-nth n (bake-t-next a shrd) tsks)
                (bake-sh-next shrd a)
                (next-oracle orcl)))))
```

- Computed measure for bakery next-state relation is used to define a measure for `bake-run`
- Computed measure for bakery blocking relation is used in support of the witness for `choose-ready`
- Given user specification of abstract values and submeasures, process is (essentially) automatic

Related Work, Ongoing Work, Concluding..

- Closest related work are *CCGs*[Manolios, Vroon, 2006]
  - Used for proving function termination in `ACL2s`.
  - Termination is ensured by showing no infinite paths through calling context graphs pulled from function definition.
  - Similar notions of submeasures and some concepts but different constructions and application domains.
- Ongoing work:
  - General methodology for generating abstract value maps and submeasures (hierarchically) from "type" declarations and static analysis.
- Thanks.. and Questions?