Using Equivalence Relations to Capture Define/Use Behaviors

David Greve

11/13/2023

Background

- Motivation
  - Congruence-based rewriting is just so cool
  - Equivalence Relations are restrictive
  - Def/Use with ‘nary’ library
  - Is it possible with equivalence relations?
- Impetus
  - ACL2 help request by Mark Greenstreet

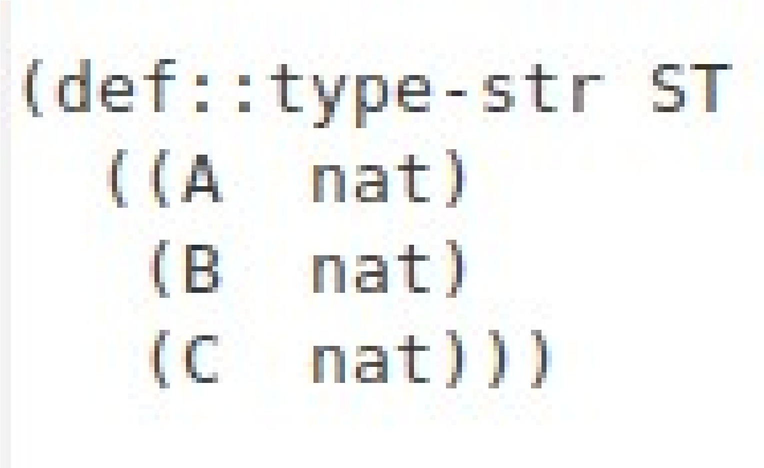
Define/Use

- Consider functions that operate over a “state record”
- Use Set
  - The fields of a record (or inputs) used by a function
- Def Set
  - The fields of a record modified by a function
- Information Flow Specifications
  - Dependencies between record fields
  - A <= {B,C}
  - Live between type specifications and functional specifications

Why do we care?

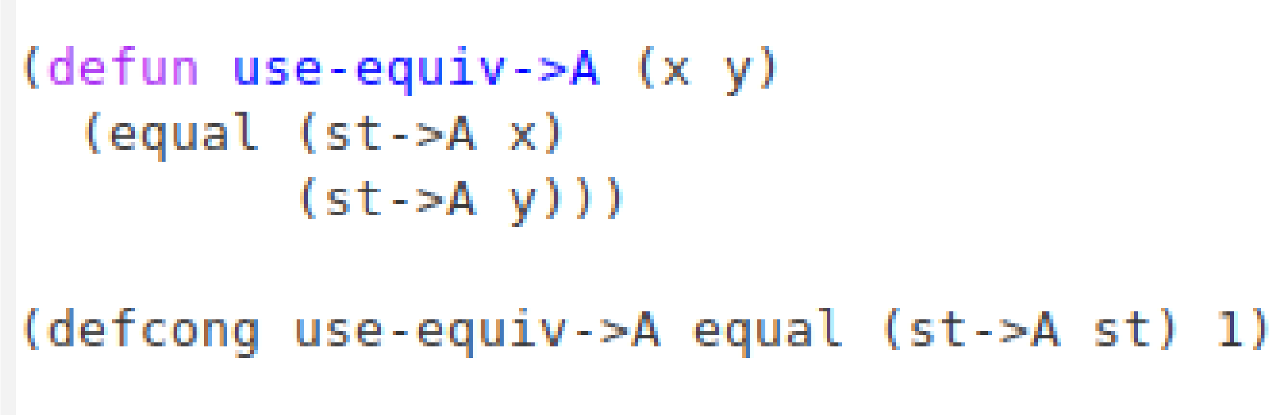
- Non-interference/Frame Conditions
  - Things that don’t change
- **Simplification**
  - Eliminate the things we don’t care about
  - Normalize the things we do care about
  - BTW: This is **why** congruences are so great

A State Record



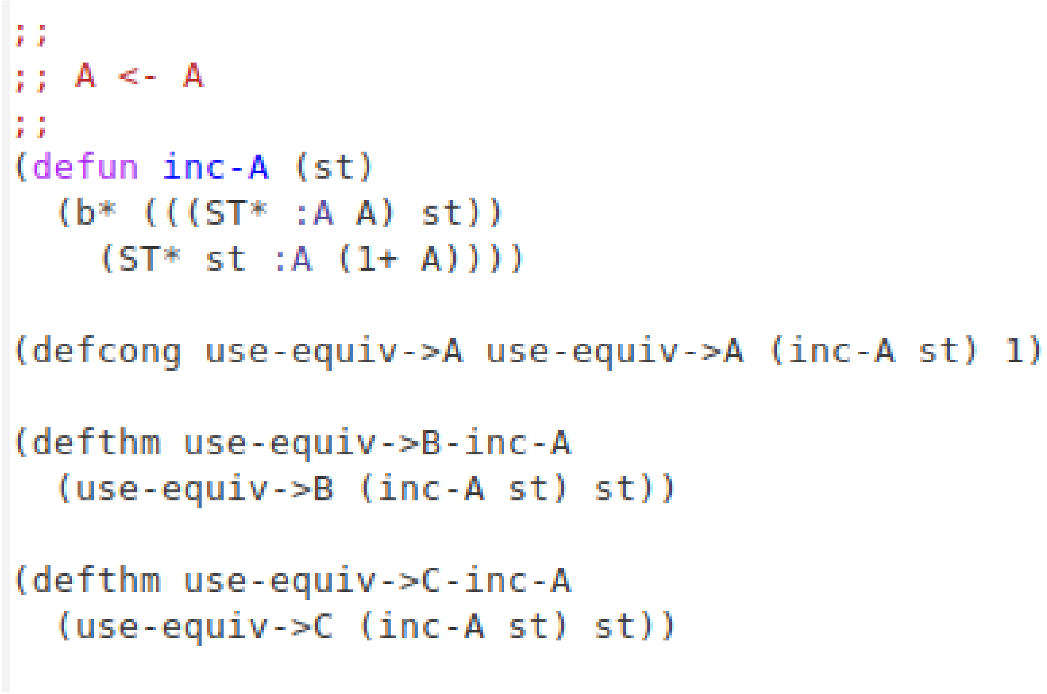
```
(def::type-str ST
  ((A  nat)
   (B  nat)
   (C  nat)))
```

“A-equiv”



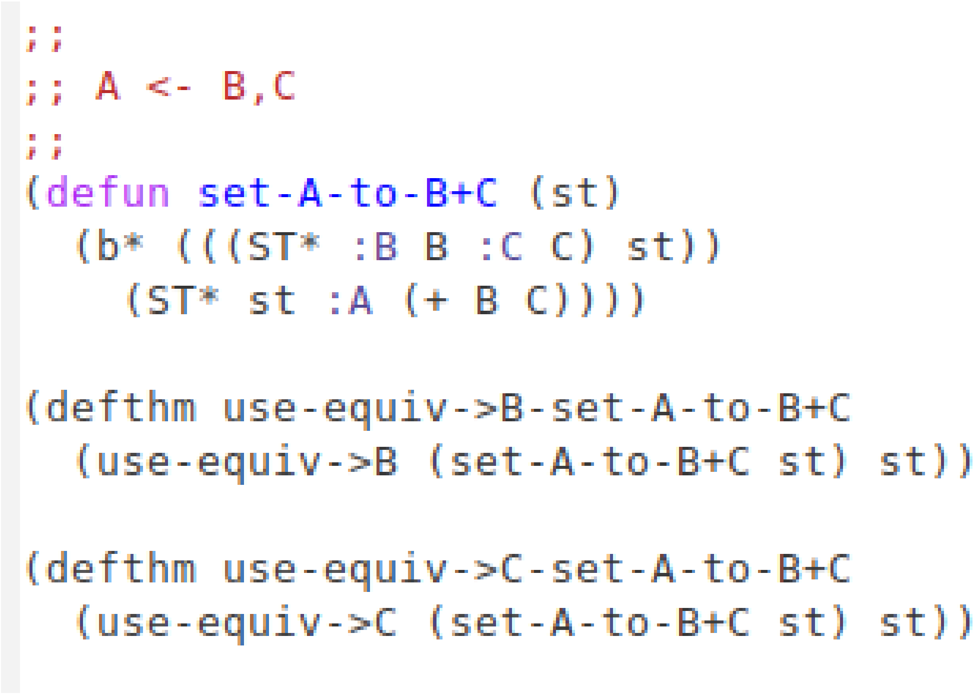
```
(defun use-equiv->A (x y)
  (equal (st->A x)
         (st->A y)))

(defcong use-equiv->A equal (st->A st) 1)
```

A <= {A}



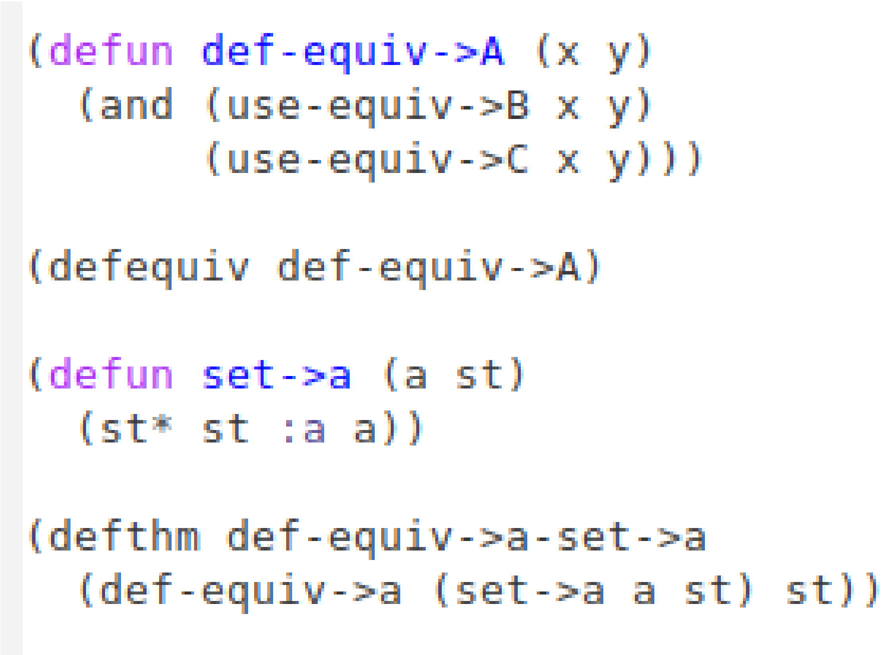
```
;;
;; A <- A
;;
(defun inc-A (st)
  (b* (((ST* :A A) st))
    (ST* st :A (1+ A))))

(defcong use-equiv->A use-equiv->A (inc-A st) 1)

(defthm use-equiv->B-inc-A
  (use-equiv->B (inc-A st) st))

(defthm use-equiv->C-inc-A
  (use-equiv->C (inc-A st) st))
```

A <= {B,C}



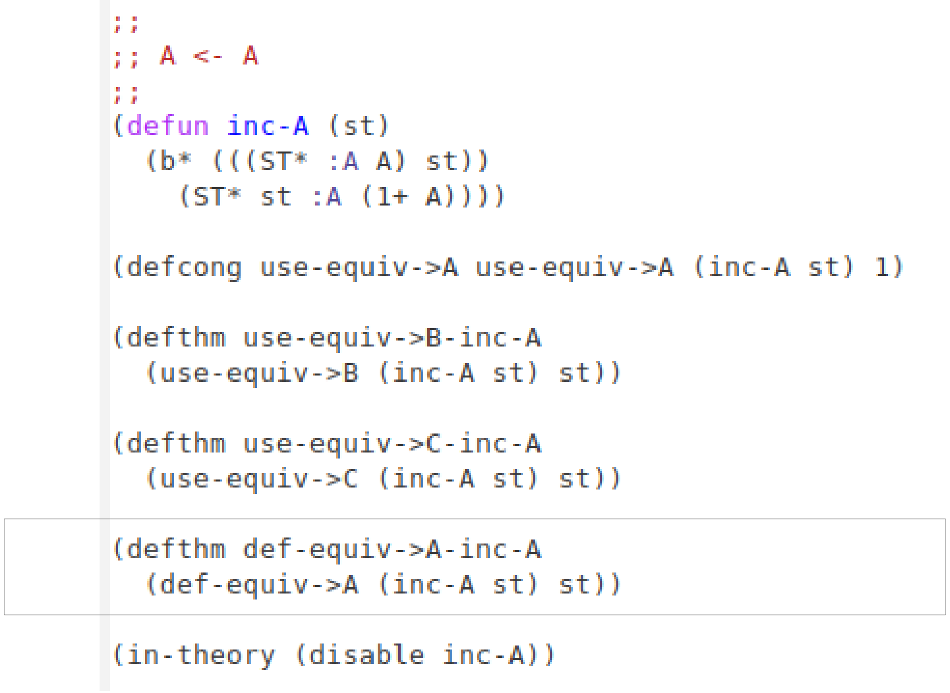
```
;;
;; A <- B,C
;;
(defun set-A-to-B+C (st)
  (b* (((ST* :B B :C C) st))
    (ST* st :A (+ B C))))

(defthm use-equiv->B-set-A-to-B+C
  (use-equiv->B (set-A-to-B+C st) st))

(defthm use-equiv->C-set-A-to-B+C
  (use-equiv->C (set-A-to-B+C st) st))
```

Dual Equivalences (def-equiv)



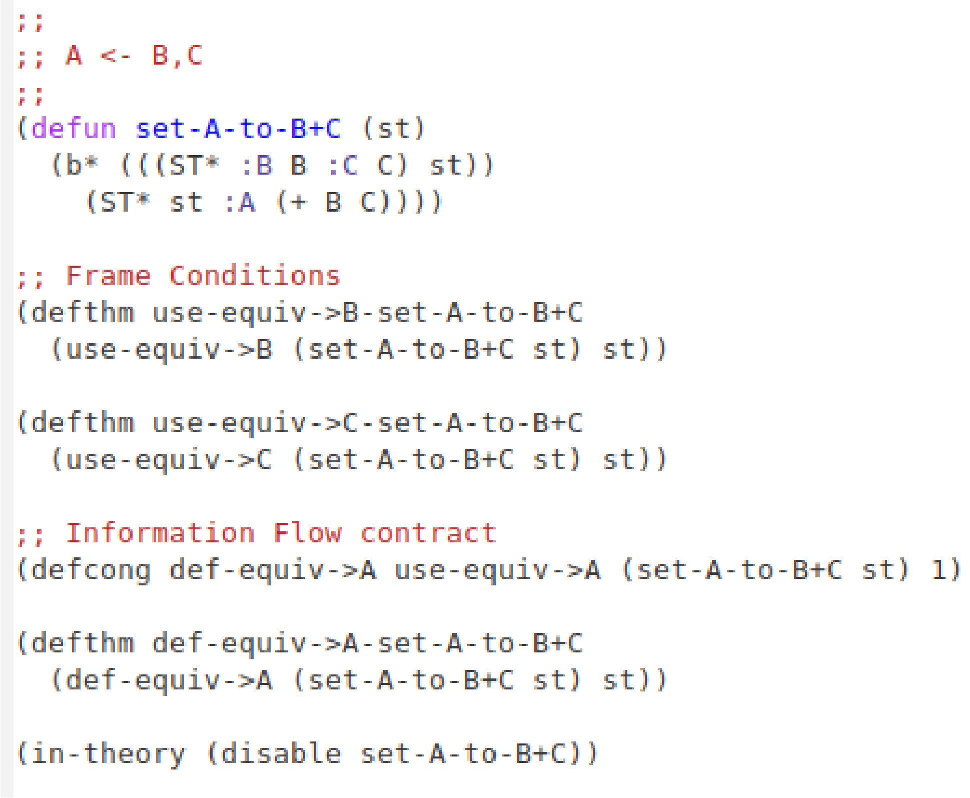
```
(defun def-equiv->A (x y)
  (and (use-equiv->B x y)
       (use-equiv->C x y)))

(defequiv def-equiv->A)

(defun set->a (a st)
  (st* st :a a))

(defthm def-equiv->a-set->a
  (def-equiv->a (set->a a st) st))
```

Extended inc-A contract



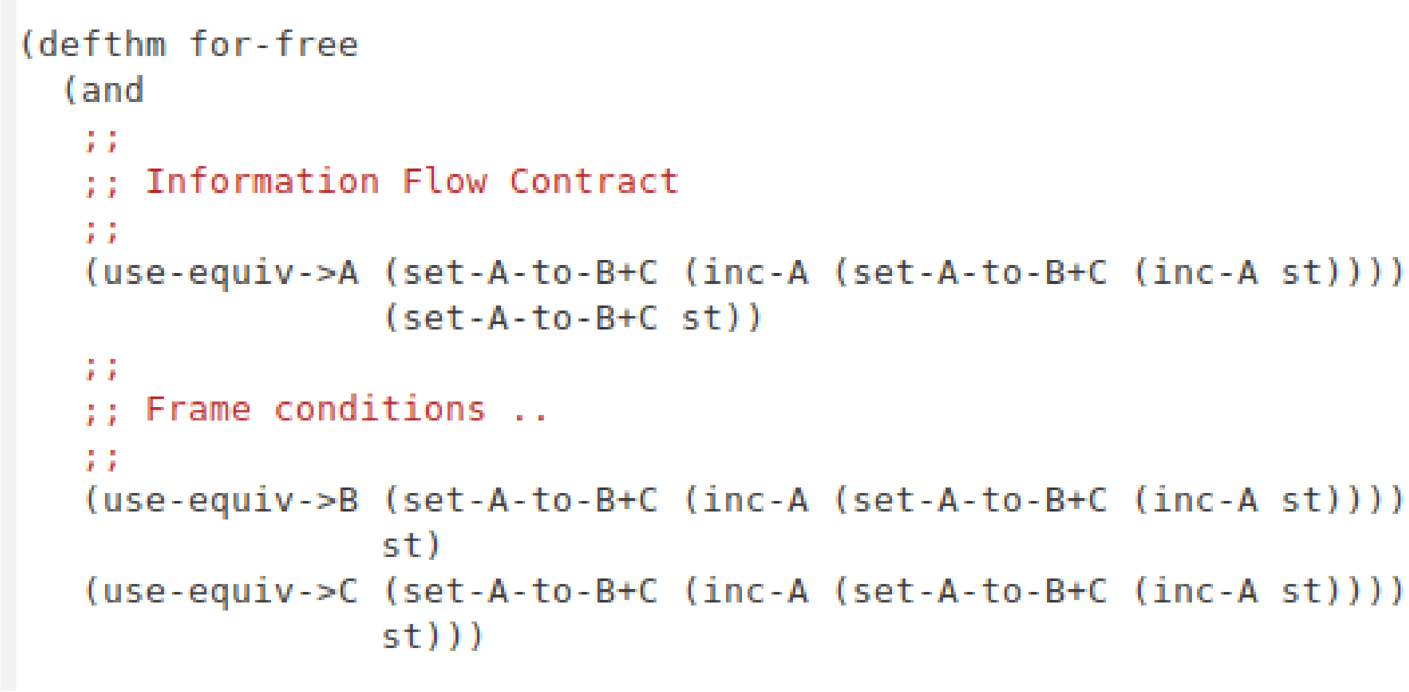
```
;;
;; A <- A
;;
(defun inc-A (st)
  (b* (((ST* :A A) st))
    (ST* st :A (1+ A))))

(defcong use-equiv->A use-equiv->A (inc-A st) 1)

(defthm use-equiv->B-inc-A
  (use-equiv->B (inc-A st) st))

(defthm use-equiv->C-inc-A
  (use-equiv->C (inc-A st) st))

(defthm def-equiv->A-inc-A
  (def-equiv->A (inc-A st) st))

(in-theory (disable inc-A))
```

Extended A=B+C Contract



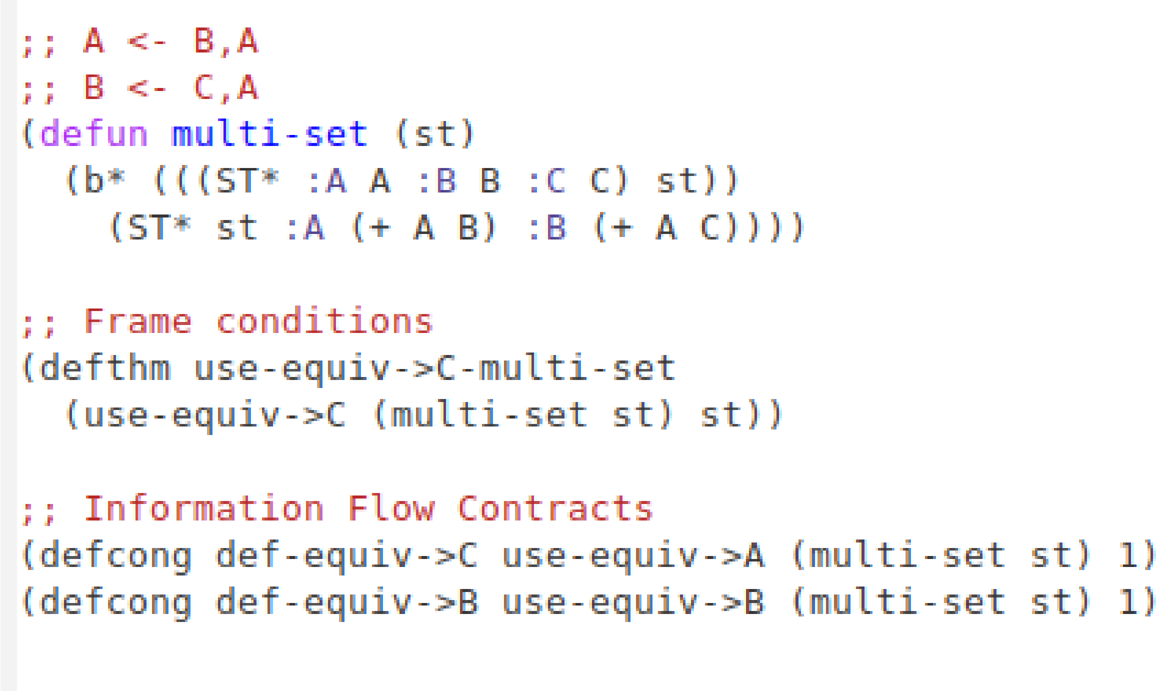
```
;;
;; A <- B,C
;;
(defun set-A-to-B+C (st)
  (b* (((ST* :B B :C C) st))
    (ST* st :A (+ B C))))

;; Frame Conditions
(defthm use-equiv->B-set-A-to-B+C
  (use-equiv->B (set-A-to-B+C st) st))

(defthm use-equiv->C-set-A-to-B+C
  (use-equiv->C (set-A-to-B+C st) st))

;; Information Flow contract
(defcong def-equiv->A use-equiv->A (set-A-to-B+C st) 1)

(defthm def-equiv->A-set-A-to-B+C
  (def-equiv->A (set-A-to-B+C st) st))

(in-theory (disable set-A-to-B+C))
```

Normalization



```
(defthm for-free
  (and
   ;;
   ;; Information Flow Contract
   ;;
   (use-equiv->A (set-A-to-B+C (inc-A (set-A-to-B+C (inc-A st))))
                 (set-A-to-B+C st))
   ;;
   ;; Frame conditions ..
   ;;
   (use-equiv->B (set-A-to-B+C (inc-A (set-A-to-B+C (inc-A st))))
                 st)
   (use-equiv->C (set-A-to-B+C (inc-A (set-A-to-B+C (inc-A st))))
                 st)))
```

Still Limitations ..



```
;; A <- B,A
;; B <- C,A
(defun multi-set (st)
  (b* (((ST* :A A :B B :C C) st))
    (ST* st :A (+ A B) :B (+ A C))))

;; Frame conditions
(defthm use-equiv->C-multi-set
  (use-equiv->C (multi-set st) st))

;; Information Flow Contracts
(defcong def-equiv->C use-equiv->A (multi-set st) 1)
(defcong def-equiv->B use-equiv->B (multi-set st) 1)
```

Conclusion

- Dual equivalence relations (def-equiv)
  - Can capture “complex” information flow contracts
- Contracts could be added to function signatures
  - (def::un foo (st) (declare (xargs :flows ((a . b c))) ..)
- “Optimal” Simplification
  - Would require more powerful/expensive rules