Predicate Refinement and Equivalences

David Greve

Rockwell Collins

Building trust every day

Predicate Refinement and Equivalences

- “A foolish consistency is the hobgoblin of little minds”
  - Ralph Waldo Emerson
- “Predicate Refinement should entail Equivalence Refinement”
  - David “Little Mind” Greve
- ((AType-p x) => (BType-p x)) => ((Bequiv x y) => (Aequiv x y))
- FTY seems to provides no support for conjunctive/disjunctive types
  - It should
  - I was concerned that it might be due to issues related to equivalence
    - Based on this study, I don’t think that is the case

Example

```
(local
 (encapsulate
     ()

   (defthm natp-implies-integerp
     (implies
      (natp x)
      (integerp x)))

   (defun integer-equiv (x y)
     (equal (ifix x) (ifix y)))

   (defequiv integer-equiv)

   (defun nat-equiv (x y)
     (equal (nfix x) (nfix y)))

   (defequiv nat-equiv)

   (defrefinement integer-equiv nat-equiv)

   ))
```


Conjunctions

```
;;
;; ---------------------------------------------------------------
;;
;; There are specific conditions under which new predicates,
;; constructed as conjunctions and disjunctions of existing
;; predicates, exhibit this behavior.
;;
;; Sufficient conditions for 'and' are:
;;
;; (defcong t1-equiv xequiv (t1-and-t2-fix x) 1)
;;
;; (defcong t2-equiv xequiv (t1-and-t2-fix x) 1)
;;
;; See encapsualte below for conditions on 'replacing' fixers/
;; default values.  Note: xequiv is typcially just 'equal'.
;;
;; ---------------------------------------------------------------
```


Conjunction Model

```
(encapsulate
    (
     ((t1-p *) => *)
     ((t2-p *) => *)
     ((t1-witness) => *)
     ((t2-witness) => *)
     ((both-witness) => *)
     ((xequiv * *) => *)
     )

  (defequiv xequiv)
  (defcong xequiv equal (t1-p x) 1)
  (defcong xequiv equal (t2-p x) 1)

  (defthm t1-witness-types
    (t1-p (t1-witness))
    :rule-classes ((:forward-chaining :trigger-terms ((t1-witness)))))

  (defthm t2-witness-types
    (t2-p (t2-witness))
    :rule-classes ((:forward-chaining :trigger-terms ((t2-witness)))))

  (defthm both-witness-types
    (and (t1-p (both-witness))
         (t2-p (both-witness)))
    :rule-classes ((:forward-chaining :trigger-terms ((both-witness)))))
```


Conjunction Assumptions

```
;; For 'replacing' fixers, these are our main assumptions ..

(defthm xequiv-witness-2-to-1
  (implies
   (t1-p (t2-witness))
   (xequiv (t2-witness)
           (t1-witness))))

(defthm xequiv-witness-1-to-both
  (implies
   (t2-p (t1-witness))
   (xequiv (t1-witness)
           (both-witness))))

(defthm xequiv-witness-2-to-both
  (implies
   (t1-p (t2-witness))
   (xequiv (t2-witness)
           (both-witness))))

)
```


Conjuncts

```
;; ---------------------------------------------

(defun t1-fix (x)
  (if (t1-p x) x (t1-witness)))

(defthm t1-fix-id
  (implies
   (t1-p x)
   (equal (t1-fix x) x)))

(defthm t1-p-t1-fix
  (t1-p (t1-fix x)))

(in-theory (disable t1-fix))

(defun t1-equiv (x y)
  (xequiv (t1-fix x) (t1-fix y)))

(defequiv t1-equiv)

(defthm t1-type-equiv-t1-fix
  (t1-equiv (t1-fix x) x))

(defcong t1-equiv xequiv (t1-fix x) 1)

(in-theory (disable t1-equiv))

;; ---------------------------------------------
```

```
;; ---------------------------------------------

(defun t2-fix (x)
  (if (t2-p x) x (t2-witness)))

(defthm t2-fix-id
  (implies
   (t2-p x)
   (equal (t2-fix x) x)))

(defthm t2-p-t2-fix
  (t2-p (t2-fix x)))

(in-theory (disable t2-fix))

(defun t2-equiv (x y)
  (xequiv (t2-fix x) (t2-fix y)))

(defequiv t2-equiv)

(defthm t2-type-equiv-t2-fix
  (t2-equiv (t2-fix x) x))

(defcong t2-equiv xequiv (t2-fix x) 1)

(in-theory (disable t2-equiv))

;; ---------------------------------------------
```


The Conjunction

```
;; -------------------------------------------

(defun t1-and-t2-p (x)
  (and (t1-p x) (t2-p x)))

(defun t1-and-t2-fix (x)
  (if (t1-and-t2-p x) x
    (both-witness)))

(defthm t1-and-t2-fix-id
  (implies
   (t1-and-t2-p x)
   (equal (t1-and-t2-fix x) x)))

(defthm t1-and-t2-p-t1-and-t2-fix
  (t1-and-t2-p (t1-and-t2-fix x))
  :rule-classes ((:forward-chaining :trigger-terms ((t1-and-t2-fix x)))))

(in-theory (disable t1-and-t2-fix))

(defun t1-and-t2-equiv (x y)
  (xequiv (t1-and-t2-fix x)
          (t1-and-t2-fix y)))

(defequiv t1-and-t2-equiv)

(defthm t2-and-t2-type-equiv-t1-and-t2-fix
  (t1-and-t2-equiv (t1-and-t2-fix x) x))

(defcong t1-and-t2-equiv xequiv (t1-and-t2-fix x) 1)

(in-theory (disable t1-and-t2-equiv))

;; -------------------------------------------
```


Conjunction Refinement Properties

```
;; ---------------------------------------------

(defcong t1-equiv xequiv (t1-and-t2-fix x) 1
  :hints (("Goal" :in-theory (enable t1-equiv t1-fix t1-and-t2-fix))))

(defcong t2-equiv xequiv (t1-and-t2-fix x) 1
  :hints (("Goal" :in-theory (enable t2-equiv t2-fix t1-and-t2-fix))))

;; ---------------------------------------------

(defrefinement t1-equiv t1-and-t2-equiv
  :hints (("Goal" :in-theory (enable t1-and-t2-equiv))))

(defrefinement t2-equiv t1-and-t2-equiv
  :hints (("Goal" :in-theory (enable t1-and-t2-equiv))))

;; ---------------------------------------------
```


Rockwell Collins

Disjunctions

```
;; ------------------------------------------------------------
;;
;; There are specific conditions under which new predicates,
;; constructed as conjunctions and disjunctions of existing
;; predicates, exhibit this behavior.
;;
;; Sufficient conditions for 'or' are:
;;
;; (defcong t1-or-t2-equiv xequiv (t1-fix x) 1)
;;
;; (defcong t1-or-t2-equiv xequiv (t2-fix x) 1)
;;
;; See encapsualte below for conditions on 'replacing' fixers/
;; default values.  Note: xequiv is typcially just 'equal'.
;;
;; ------------------------------------------------------------
```


Conjunction Model and Assumptions

```
(encapsulate
    (
     ((t1-p *) => *)
     ((t1-witness) => *)
     ((t2-p *) => *)
     ((t2-witness) => *)
     ((xequiv * *) => *)
     )

  (defequiv xequiv)
  (defcong xequiv equal (t1-p x) 1)
  (defcong xequiv equal (t2-p x) 1)

  (defthm t1-p-t1-witness
    (t1-p (t1-witness))
    :rule-classes ((:forward-chaining :trigger-terms ((t1-witness)))))

  (defthm t2-p-t2-witness
    (t2-p (t2-witness))
    :rule-classes ((:forward-chaining :trigger-terms ((t2-witness)))))

  ;; For 'replacing' fixers, this is our main assumption ..
  (defthm witness-reduction
    (implies
     (t2-p (t1-witness))
     (xequiv (t2-witness) (t1-witness))))

  )
```


Disjuncts

```
;; ---------------------------------------------

(defun t1-fix (x)
  (if (t1-p x) x (t1-witness)))

(defthm t1-fix-id
  (implies
   (t1-p x)
   (equal (t1-fix x) x)))

(defthm t1-p-t1-fix
  (t1-p (t1-fix x)))

(in-theory (disable t1-fix))

(defun t1-equiv (x y)
  (xequiv (t1-fix x) (t1-fix y)))

(defequiv t1-equiv)

(defthm t1-type-equiv-t1-fix
  (t1-equiv (t1-fix x) x))

(defcong t1-equiv xequiv (t1-fix x) 1)

(in-theory (disable t1-equiv))

;; ---------------------------------------------
```

```
;; ---------------------------------------------

(defun t2-fix (x)
  (if (t2-p x) x (t2-witness)))

(defthm t2-fix-id
  (implies
   (t2-p x)
   (equal (t2-fix x) x)))

(defthm t2-p-t2-fix
  (t2-p (t2-fix x)))

(in-theory (disable t2-fix))

(defun t2-equiv (x y)
  (xequiv (t2-fix x) (t2-fix y)))

(defequiv t2-equiv)

(defthm t2-type-equiv-t2-fix
  (t2-equiv (t2-fix x) x))

(defcong t2-equiv xequiv (t2-fix x) 1)

(in-theory (disable t2-equiv))

;; ---------------------------------------------
```


The Disjunction

```
;; ---------------------------------------------

(defun t1-or-t2-p (x)
  (or (t1-p x)
      (t2-p x)))

(defun t1-or-t2-fix (x)
  (if (t1-or-t2-p x) x
    (t1-fix x)))

(defthm t1-or-t2-fix-id
  (implies
   (t1-or-t2-p x)
   (equal (t1-or-t2-fix x) x)))

(defthm t1-or-t2-p-t1-or-t2-fix
  (t1-or-t2-p (t1-or-t2-fix x))
  :rule-classes ((:forward-chaining :trigger-terms ((t1-or-t2-fix x)))))

(in-theory (disable t1-or-t2-fix))

(defun t1-or-t2-equiv (x y)
  (xequiv (t1-or-t2-fix x)
          (t1-or-t2-fix y)))

(defequiv t1-or-t2-equiv)

(defthm t1-or-t2-type-equiv-t1-or-t2-fix
  (t1-or-t2-equiv (t1-or-t2-fix x) x))

(defcong t1-or-t2-equiv xequiv (t1-or-t2-fix x) 1)

(in-theory (disable t1-or-t2-equiv))

;; ---------------------------------------------
```


Disjunction Refinement Properties

```
;; ------------------------------------------

(defcong t1-or-t2-equiv xequiv (t1-fix x) 1
  :hints (("Goal" :in-theory (enable t1-or-t2-equiv t1-or-t2-fix t1-fix t2-fix))))

(defcong t1-or-t2-equiv xequiv (t2-fix x) 1
  :hints (("Goal" :in-theory (enable t1-or-t2-equiv t1-or-t2-fix t1-fix t2-fix))))

;; ------------------------------------------

(defrefinement t1-or-t2-equiv t1-equiv
  :hints (("Goal" :in-theory (enable t1-equiv))))

(defrefinement t1-or-t2-equiv t2-equiv
  :hints (("Goal" :in-theory (enable t2-equiv))))

;; ------------------------------------------
```


Rockwell Collins

Conclusion

- The assumptions under which type refinement entails equivalence refinement are not onerous and are likely to be common
- Good type disciple should include providing and proving these refinement relations when possible

