Implementing abstract types in ACL2

Vernon Austel

IBM

A very simple example

```
(defthm append-nil
   (implies (true-listp x)
            (equal (append x nil) x)))

;; This is false.
(thm
   (equal (append x nil) x))

(defthm list=-append-nil
   (list= (append x nil) x))
```

Equivalence relation and fixer

```
(defun listfix (x)
  (if (endp x)
      nil
    (cons (car x) (listfix (cdr x)))))

(defun list= (x y)
  (equal (listfix x) (listfix y)))

(defequiv list=)
```

congruences on the type

```
(defcong list= list= (append x y) 1)
(defcong list= list= (append x y) 2)

(defthm list=-append-nil
    (list= (append x nil) x))

(thm
   (list= (append (append x nil) y)
            (append x y)))
```

Pros and cons

+ fewer hypotheses
- more prep work
- have to remember to use
  equivalence relation
- doesn't work with linear rewriting
  (e.g. to replace integerp with intgr=)

Chores involving the new type

- define a ``kind'' predicate, if appropriate
- define destructors and constructors
- prove measure lemmas for the destructors
- define a ``fix'' function, using the destructors
- define the equivalence using the fix function
- prove congruence theorems for the destructors and constructors
- prove elimination rules for the constructors

Constructors and destructors

```
(defun expr-kind (expr)
  (cond ((symbolp expr) 'SYMBOL)
        ((consp expr) 'BINOP)
        (t 'LIT)))

(defun binop-left (expr)
    (if (equal (expr-kind expr) 'BINOP)
         (caddr expr)
       nil))

(defun mk-binop (op left right)
    (list 'BINOP op left right))
```

The equivalence relation

```
(defun exprfix (expr)
  (let ((kind (expr-kind expr)))
    (case kind
       (SYMBOL expr)
       (LIT    (litfix expr))
       (otherwise
         (mk-binop
             (binop-op expr)
             (exprfix (binop-left expr))
             (exprfix (binop-right expr)))))))

(defun expr= (x y)
  (equal (exprfix x) (exprfix y)))
(defequiv expr=)
```

Defining functions on the type (1)

```
(defun free-vars (expr)
  (let ((kind (expr-kind expr)))
    (case kind
      (SYMBOL (list expr))
      (LIT  nil)
      (otherwise
        (append (free-vars (binop-left expr))
                (free-vars (binop-right expr))))
(defcong expr= (free-vars expr) 1)
```

Defining functions on the type (2)

```
;; this defines the function and
;; proves the congruence
(defexpr free-vars (expr) equal
   :SYMBOL (list expr)
   :LIT    nil
   :BINOP  (append $left $right))
```

Proving theorems using the type

```
;; this has no type hypothesis for expr
(defexprthm env-irrelevant
   (implies (not (consp (free-vars expr)))
            (equal (eval-expr expr env)
                   (eval-expr expr nil))))
```

Induction using functional instantiation

```
(encapsulate
  ((expr-induct (expr) t))

  (local (defun expr-induct (x) (declare (ignore x)) t))

  (defthm expr-induct-symbol
    (implies (equal (expr-kind expr) 'SYMBOL)
             (expr-induct expr)))

  (defthm expr-induct-lit
    (expr-induct (litfix expr)))

  (defthm expr-induct-binop
    (implies (and (expr-induct left)
                      (expr-induct right))
             (expr-induct (mk-binop binop left right))))

  (defcong expr= iff (expr-induct expr) 1))
```

Induction using functional instantiation

Subgoal 2

```
(implies (and (or (not (not (consp (free-vars left))))
                  (equal (eval-expr left env)
                         (eval-expr left nil)))
              (or (not (not (consp (free-vars right))))
                  (equal (eval-expr right env)
                         (eval-expr right nil)))
              (not (consp (append (free-vars left)
                                  (free-vars right)))))
         (equal (+ (eval-expr left env)
                   (eval-expr right env))
                (+ (eval-expr left nil)
                   (eval-expr right nil)))).
```

Proof performance

Time to prove "env-irrelevant"

Using normal induction:

Time: 0.09 seconds (prove: 0.05, print: 0.01, other: 0.03)

Using functional instantiation:

Time: 0.04 seconds (prove: 0.03, print: 0.00, other: 0.01)

Drawbacks of functional instantiation

- Constraints may be wrong
  - too strong
- Variable names used in constraints may not be used in theorems ("left", "right")
- Induction cannot change arguments in recursive calls (e.g., for an accumulator)

Conclusions

- This is workable, but not easy
- Changes to ACL2 could make it easy
  - guess congruences
    - no proof necessary - syntactic check
  - Modify induction to use constructors
  - only allow type-correct fns and theorems
    - avoids silly mistakes