Implementing abstract types in ACL2

Vernon Austel
IBM T.J. Watson Research Center

July 7, 2003

Abstract

This paper summarizes the author's experience implementing something resembling abstract types in ACL2. It assumes some experience with the ACL2 theorem prover.

1 Introduction

This paper summarizes the author's experience implementing something resembling abstract types in ACL2[2]. It assumes some experience with the ACL2 theorem prover.

ACL2 imposes no syntactic restriction on the arguments that a function may be applied to[1]; any function (e.g. `car`) may be applied to any object (e.g `3`), and the result of such function applications must be dealt with in proofs.

Type abstraction separates the interface of a data structure from its implementation [1]. This is accomplished by syntactically restricting which operations can be applied to data of a given type. In typed languages and logics, the typing mechanism is enforced by the compiler or interpreter; one cannot access an object except through its interface. There is no provision for this in ACL2; the best one can do is try to avoid violating the interface.

The motivations for using abstract types are different in programming languages as opposed to logics. In programming languages, it allows the implementation of the type to be changed without affecting the logic of the code that uses it, and hiding representation details makes the code that uses it easier to understand [1, page 89]. It isn't clear that these are important issues in a logic, where one is (often) more concerned about what a specification does that its execution efficiency, and where it is generally possible to understand what it does in much greater detail than any program, regardless of how the data is organized.

The author nevertheless sees advantages in using something like abstract types for reasoning. Theorems tend to have fewer hypotheses, and inductive

[1]Of course, the correct number of arguments must be used, and single-threaded objects must be passed appropriately.

subgoals tend to be more readable. The drawback is that it requires more work to set up the supporting framework.

This paper goes into some detail on how to get much of the flavor of abstract types using equivalence relations and functional instantiation in ACL2. It contains nothing difficult or new (similar ideas are in the book `set-theory.lisp` in the standard distribution), but the example given is more elaborate, and may benefit those who are unfamiliar with the issues raised.

2 A very simple example

We use lists as a simple introductory example. The first of the following two events is a theorem in ACL2, but the second is not, because `append` may be applied to any ACL2 object, including integers and strings; for example, `(append 0 nil)` is equal to `nil` (not `0`).

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

;; This is false.
(thm
   (equal (append l nil) l))
```

The problem is that we are only "interested" in the application of `append` to true lists, but must nevertheless deal with other, "senseless" cases. Here, what may be thought of as the type information concerning the argument is provided by the hypothesis `(true-listp l)`. While making all functions total simplifies the logic, it is a nuisance when stating and proving theorems about those functions.

One may alternatively provide this type information via an equivalence relation; rather than using a hypothesis to screen out the unwanted cases, one makes the equivalence function treat them like sensible ones. We end up with a goal like this (which, given the right definition for `list=`, is a theorem):

```
(thm
   (list= (append l nil) l))
```

One may define equivalence relations in different ways, but the author has found it convenient to define them in terms of "fixing" routines[2]. A fixing routine takes the universe of all ACL2 objects and maps it into the subset of objects that we are interested in. In this case, we map all objects into the set of true lists. Two objects are equivalent if they are mapped to the same true list.

[2]The standard ACL2 book `set-theory` also uses this style. Experienced ACL2 users may be concerned that using fixing functions is inefficient, but the author is not concerned about efficiency in this paper. One may always define a second relation that does not use a fixing function, prove that the second relation is the same as the equivalence relation, use the efficient relation in function definitions and rewrite it to the equivalence relation. The author believes that techniques like this can address any efficiency issues raised by this paper.

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

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

(defequiv list=)
```

With this equivalence, one can prove the goal above, after proving some preliminary lemmas (which actually may not be obvious for those inexperienced with these kinds of proofs).

The whole point of introducing an equivalence is to use it in congruence theorems. Roughly speaking, the following events say that `append` returns results that are `list=` if they are given inputs that are `list=`; in the official terminology, `list=` is maintained by `list=` in the first (and second) argument of `append`.

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

The above `listfix` is not the only possible fixing function for true lists. One could also define `listfix` as follows:

```
(defun another-listfix (x)
   (if (true-listp x)
      x
     nil))
```

Generally speaking, one wants the fixing function to "work well" with functions to be defined on the data type. This particular fixing function does not seem useful, because `list=` is not maintained by `list=` on either argument of `append`.

This short example already shows some of the advantages and disadvantages of this style. The theorem we prove has fewer hypotheses (none in this case), but more preparation (functions and theorems) was required to get it. Since rules won't "fire" if their hypotheses aren't relieved, all other things being equal, a rule with fewer hypotheses is easier to use (since one is often in the position of trying to figure out why a particular rule didn't fire); on the other hand, one must remember to use the right equivalence relation, and must remember to prove the proper congruence theorems about functions being used[3]. In sum, using equivalence relations does not have a clear advantage over using type hypotheses; the author nevertheless prefers it for complicated data types.

[3]In some unusual circumstances ACL2 will actually not relieve trivial hypotheses such as `(equiv x x)`, for an equivalence relation `equiv` other than `equal` or `iff`. This can be especially annoying because it is the last thing one thinks of when trying to determine why a rule doesn't fire.

3 A more realistic example: expressions

We now describe the general steps one goes through to define an abstract type in this style, using expression evaluation as the example since that is likely to be familiar. The prepatory steps are:

- 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

The abstract types considered here correspond to what a C programmer would think of as a "union" type where one can distinguish between the union variants. For example, there are many kinds of C expressions: applications of binary and unary operators, pointer and struct dereferences, address expressions, and more. The represention of each kind of expression generally has different subcomponents; we will assume that each subcomponent has a distinct accessor (or "destructor"). There must be some way to determine which union variant a given object corresponds to so that (only) the proper destructor is applied to it.

One way is to define a predicate for each variant whose purpose is to recognize that variant; for example, a predicate `binop-p` would recognize only those expressions that represent binary operations. The problem is that one frequently ends up with goals that contain contradictory information about a particular object, perhaps containing both `(binop-p expr)` and `(unop-p expr)`. With specialized predicates, one must have many rules saying (for example) that an object cannot satisfy both `binop-p` and `unop-p`. These contradictions are resolved faster if there is a single function that returns a symbol or integer representing the union variant that the object corresponds to. One might be tempted to "not bother" using this function for variants that are implemented with primitive types (such as symbols), because one could then use a primitive function (such as `symbolp`) to test for them, but we believe that things work better if the function is used to distinguish between all variants.

In our example there will be only three kinds of expressions: integers, symbols and binary operations. This is the function we use to divide the ACL2 universe into these three categories:

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

There is considerable freedom in how this function is defined. We have found that it simplifies proofs if the variants can be distinguished as easily as possible. One of the variants must be a "leftover" case; here we happen to interpret the leftovers as literals, but there is no particular reason to do so.

This function is *not* used as a "recognizer"; that is, the fact that the `expr-kind` of an object is BINOP does not means it what one would think of as a "well-formed" binary expression. That is a separate concept, which can be developed to be used in guards to improve performance. There is no logical necessity to require objects to be well-formed; indeed, the whole point of using equivalence relations in this case is to avoid the concept of well-formedness in theorems.

This example is designed to be small, but large enough to illustrate the main ideas without being trivial. In actual datatypes most variants are likely to be similar to BINOP; a data structure representing C statements may have ten variants.

Only the BINOP variant in our simple example has subcomponents. They are defined as follows:

```
(defun binop-op (x)
   (if (equal (expr-kind x) 'BINOP)
       (cadr x)
     nil))

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

(defun binop-right (expr)
   (if (equal (expr-kind expr) 'BINOP)
       (cadddr expr)
     nil))
```

The destructors all test the argument to see if it is the appropriate kind; otherwise, they return some default value (which happens to be nil in this case, but could be whatever is convenient). If they did not do so, one probably could not prove useful congruence theorems about them. Obviously, no attempt at all has been made to make the destructors efficient; they are designed to simplify proofs. Issues of efficiency are not addressed in this paper.

It is convenient to prove measure lemmas for function definitions, since in general we want to disable the destructor definitions and regard them as primitives. This is one of the two that must be proved for expressions:

```
(defthm acl2-count-binop-left
   (implies (equal (expr-kind expr) 'BINOP)
            (< (acl2-count (binop-left expr))
               (acl2-count expr)))
   :rule-classes (:rewrite :linear))
```

After this, one defines constructor functions; in this case, there is really only one, but we show two. Our expressions only allow integer literals, and in a sense one "constructs" such a literal by fixing it. This is certainly different from packaging it in a list, but such "constructors" are used in proofs in a way that is similar to ordinary constructors[4]. We use it just to constrast it with our handling of symbols.

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

(defun litfix (x)
   (ifix x))
```

Next, one defines the "fix" function and the equivalence relation, similar to the way `list=` was defined above.

```
(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=)
```

Finally, one shows that the destructors and constructors maintain the appropriate equivalence (often `expr=` or `equal`) on `expr=` in their arguments[5].

```
(defcong expr= equal (binop-op expr) 1)
(defcong expr= expr= (binop-left expr) 1)
(defcong expr= expr= (binop-right expr) 1)

(defcong expr= expr= (mk-binop bop left right) 2)
(defcong expr= expr= (mk-binop bop left right) 3)
```

[4] In fact, ACL2 allows one to define an elimination rule for litfix, just like any other constructor; see the proof script. It is unlikely to be useful, however.

[5] A destructor may maintain an equivalence other than `expr=` or `equal` depending on how its associated constructor is defined. For example, there may be a special equivalence relation for the set of symbols representing binary operations; if `mk-binop` coerced its bop argument to that set (using the associated fixing function), then that would be the equivalence that `binop-op` maintains. If `binop-op` itself coerced its return value, then it maintains `equal`.

It is occasionally useful to have elimination rules for the constructors. Although it doesn't play a role in this discussion, we show what one looks like in this example.

```
(defthm elim-binop
  (implies (equal (expr-kind expr) 'BINOP)
           (expr= (mk-binop (binop-op expr)
                            (binop-left expr)
                            (binop-right expr))
                  expr))
  :rule-classes (:rewrite :elim))
```

4 Defining functions on the type

After this lengthy preparation, one may now define new ACL2 functions in terms of the destructor functions and prove that they maintain an equivalence (in this case, `equal`) on `expr=`. The steps are:

- define a function in terms of the destructors and constructors
- prove "expansion" theorems
- prove the congruence theorem, using a lemma concerning the fix function

Our first example will be the a function that returns the variables used in an expression[6].

```
(defun free-vars (expr)
    (let ((kind (expr-kind expr)))
       (case kind
          (SYMBOL (list expr))
          (LIT  nil)
          (t    (append (free-vars (binop-left expr))
                        (free-vars (binop-right expr)))))))

(defthm expand-free-vars
   (and (implies (equal (expr-kind expr) 'SYMBOL)
                 (equal (free-vars expr) (list expr)))

        (implies (equal (expr-kind expr) 'LIT)
                 (equal (free-vars expr) nil))

        (equal (free-vars (litfix expr)) nil)

        (equal (free-vars (mk-binop op left right))
```

[6]While the concept of "free variables" only makes sense in a context where they can be bound, the author calls them that out of habit.

```
                (append (free-vars left)
                        (free-vars right)))))

(encapsulate
  nil
  (local
   (defthm lemma
     (equal (free-vars (exprfix expr))
            (free-vars expr))
     :rule-classes nil))

  (defcong expr= equal (free-vars expr) 1
    :hints (("Goal" :in-theory (e/d (expr=))
             :use (lemma (:instance lemma (expr expr-equiv))))))
  )
```

The expansion theorem gives the value of the new function when applied to constructors; it actually resembles what would be considered the function definition in the typed logic HOL. The basecases typically cause problems: variants without constructors (here: SYMBOL) need hypotheses, while those with constructors don't. Here, we give both a constructor and non-constructor expansion for variant LIT; it seems that each style has a way of cropping up.

The congruence theorem uses a lemma concerning `exprfix`; this is a consequence of defining the equivalence relation in terms of a fixing function. In this example, if the equivalence relation had been defined directly in terms of destructors, the proof is in some ways easier[7]. There may be no compelling reason to define equivalence relations this way, but the author has come to prefer it[8].

One can write a macro that proves the desired congruence using a generated intermediate lemma; with such a macro the encapsulate event above can be replaced by this:

```
(defcong-fix expr= equal (free-vars expr) 1)
```

The macro assumes that there is a fixing function XXXfix (here: `exprfix`) corresponding to the source equivalence relation XXX= (here: `expr=`).

The second example function is expression evaluation. We omit the expansion and congruence theorems.

```
(defun eval-expr (expr env)
   (let ((kind (expr-kind expr)))
     (case kind
```

[7] An example implementation is given in the proof script.

[8] The author believes that fixing functions simplify congruence proofs in some difficult cases, because any induction machine that correctly changes `expr` in subgoals will also correctly change `(exprfix expr)`, but it is unlikely that it will change two variables (`expr` and `expr-equiv`) correctly. In that case, one must write a dummy function to generate an induction machine that guides the other variable. Fixing functions seem easier, especially with a macro like `defcong-fix`, introduced below.

```
          (SYMBOL  (cdr (assoc expr env)))
          (LIT     (litfix expr))
          (t       (+ (eval-expr (binop-left expr) env)
                      (eval-expr (binop-right expr) env))))))
```

5 Using functional instantiation for proofs

We may now state and prove a theorem about our example functions: if an expression contains no symbols, then it doesn't matter what environment is passed to `eval-expr`[9].

```
(defthm env-irrelevant
  (implies (not (consp (free-vars expr)))
           (equal (eval-expr expr env)
                  (eval-expr expr nil))))
```

The proof of `env-irrelevant` is inductive, and its subgoals involve calls to expr-kind and the expr destructor functions (such as `binop-left`), because they are used in the definitions of the functions being reasoned about. There is nothing wrong with this, but can avoid having these destructors arise in proofs using functional instantiation. In effect, one does an abstract recursive proof once and instantiates it as needed. We now describe how this is done.

First, one uses encapsulate to specify what it means for a property to be inductive on expressions[10].

```
(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))))
```

[9]This lemma is dangerous as written, since it will probably cause overflow if it ever used, but it was difficult to think of a simple, non-trivial theorem. One could always add a syntaxp hypothesis to avoid overflow.

[10]This is naturally very similar to the so-called induction machine corresponding to the recursive functions such as `free-vars`.

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

The constraints use constructors, not destructors. We can do this because we require that the inductive relation maintain `iff` on `expr=` in its argument. As in the expansion theorem, the basecases may be given in different styles. Here, we give the basecase for literals using the "constructor" `litfix`, but the basecase for symbols is given using a hypothesis. The hypothesis could just as well have been `(symbolp expr)`, but we favor the "type kind" predicate (here: `expr-kind`) over lower-level predicates such as `symbolp`[11].

Given these assumptions, we can prove that this property must be true for all arguments.

```
(encapsulate
  nil
  (local
   (defthmd lemma
     (expr-induct (exprfix expr))
     :hints (("Goal" :in-theory (disable
                        expr=-implies-iff-expr-induct-1)))))

  (defthm expr-induct-thm
    (expr-induct expr)
    :hints (("Goal" :use lemma)))
  )
```

The author finds it interesting that in the proof of the lemma, `exprfix` converts `expr` into an ACL2 term that uses `expr` constructors which allows the subgoal to be rewritten using the appropriate constraint rule[12]; this greatly simplifies the proof. The trick is that one must prevent `(exprfix expr)` from immediately being rewritten to `expr`! Since `expr-induct` maintains `iff` on `expr=` (by hypothesis), this will happen unless we do something to prevent it[13].

With this general theorem, we can now prove our theorem using functional instantiation as follows:

```
(defthm env-irrelevant
   (implies (not (consp (free-vars expr)))
            (equal (eval-expr expr env)
                   (eval-expr expr nil)))
   :hints (("Goal"
        :use (:functional-instance
              expr-induct-thm
```

[11] Alternatively, one could introduce a "constructor" for symbols.

[12] The basecases pose a problem precisely when they have no constructors, and usually need helper lemmas, which are omitted here for brevity.

[13] The rune `expr=-implies-iff-expr-induct-1` refers to the `defcong` event that is the last constraint in the encapsulate event introducing `expr-induct`. The `defcong` macro generates this name from the arguments passed to it.

```
(expr-induct
 (lambda (expr)
   (implies (not (consp (free-vars expr)))
            (equal (eval-expr expr env)
                   (eval-expr expr nil)))))))))
```

The subgoals of this proof are generated from the constraints on `expr-induct`. There is a small amount of flexibility concerning how those constraints are stated; if nothing else, sensible variable names can be chosen (the exact same variables will appear in the subgoals).

On average, this proof takes less time than the inductive proof, especially when the definitions of the inductive functions are disabled. This is not unusual; the author believes that a speedup of five or so is typical for types with more variants and for more complex goals.[14]. It should be stressed that this speedup is only an added bonus, not the author's main motivation for using this style, which is to avoid annoying type hypotheses as much as possible.

It cannot be denied that this is a cumbersome way to write the proof. With the appropriate macro[15] the event using functional instantiation looks almost the same as the one using induction:

```
(defexprthm env-irrelevant
   (implies (not (consp (free-vars expr)))
            (equal (eval-expr expr env)
                   (eval-expr expr nil))))
```

6 Using functional instantiation for definitions

The proof of the congruence theorem for the functions `free-vars` and `eval-expr` may also be done using functional instantiation. Just as we defined an abstract "representative" inductive property on expressions, we may also define a "representative" function on expressions, together with conditions that prove that it is congruent on expressions. Because our example functions are so simple, they return exactly the same thing when passed equivalent arguments (that is why the second equivalence in the defcong is equal). That won't always be the case. We can accomodate the more general case by introducing a function to represent the target equivalence.

```
(encapsulate
  ((expr-fn= (x y) t))
  (local (defun expr-fn= (x y) (equal x y)))
  (defequiv expr-fn=)
  )
```

[14]This is not the result of a systematic comparison of the two styles. It is simply the author's habit to record the runtime of proofs that take a long time. The times for these proofs were dramatically reduced when they were converted to the new style.

[15]The macro definition is given in the proof script, but is not shown here.

We then introduce one constrained function for each `expr` variant. These functions take an argument for each destructor in the variant, or just one argument representing the whole expression if there are none. For those variants that are recursive (BINOP in our example), not only are there arguments for the recursive subcomponents, but there are arguments representing the recursive calls on those subcomponents; we denote these values with (part of) the name of the destructor prefixed with a dollar sign[16]. The `expr-symbol-fn` function has no constraints, because it doesn't have to; expressions are `expr=` iff they are `equal`[17].

```
(encapsulate
  ((expr-symbol-fn (expr) t)
   (expr-lit-fn (expr) t)
   (expr-binop-fn (op left $left right $right) t))

  (set-ignore-ok t)
  (set-irrelevant-formals-ok t)

  (local (defun expr-symbol-fn (expr) t))
  (local (defun expr-lit-fn (expr) t))
  (local (defun expr-binop-fn (op left $left right $right) t))

  (defcong expr=    expr-fn= (expr-lit-fn expr) 1)

  (defcong expr=    expr-fn= (expr-binop-fn
                               op left $left right $right) 2)
  (defcong expr-fn= expr-fn= (expr-binop-fn
                               op left $left right $right) 3)
  (defcong expr=    expr-fn= (expr-binop-fn
                               op left $left right $right) 4)
  (defcong expr-fn= expr-fn= (expr-binop-fn
                               op left $left right $right) 5)
  )
```

We may now define a general-purpose function in terms of these constrained ones and show that it maintains `expr-fn=` on `expr=` in its argument. The congruence proof resembles the proof for `free-vars`, so the details are omitted.

```
(defun expr-fn (expr)
     (let ((kind (expr-kind expr)))
       (case kind
          (SYMBOL  (expr-symbol-fn expr))
          (LIT  (expr-lit-fn expr))
          (t    (expr-binop-fn (binop-op expr)
```

[16] This is a completely arbitrary convention.

[17] This isn't obvious. The author initially constrained it, and only realized this when attempting to define the example function `expr-subst`.

```
                                       (binop-left expr)
                                       (expr-fn (binop-left expr))
                                       (binop-right expr)
                                       (expr-fn (binop-right expr)))))))
(defcong expr= expr-fn= (expr-fn expr) 1)
```

Once again, a macro makes things easier. The `defexpr` macro defines the function, generates an expansion theorem and also a congruence theorem, proving both by functional instantiation[18]. Rather than show the macro's definition, which is somewhat complicated, we will just show calls to the macro for our examples.

```
(defexpr free-vars (expr) equal
    :SYMBOL (list expr)
    :LIT    nil
    :BINOP  (append $left $right))

(defexpr eval-expr (expr env) equal
    :SYMBOL (cdr (assoc expr env))
    :LIT    (litfix expr)
    :BINOP  (+ $left $right))
```

By convention, the expression argument is always called `expr`[19]. For each kind of expression, there is a keyword argument whose key has the same name as the `expr-kind` (so for example `:SYMBOL` corresponds to `expr-kind SYMBOL`). The expression giving the value for a particular variant may have free variables, corresponding to that variant. The names of these variables follow the same conventions as were used for `expr-induct` and `expr-fn`: for expression kinds with no destructors (such as `SYMBOL`), the free variable is `expr` itself, `left` is used for the `binop-left` subcomponent of a binop expression, and `$left` denotes the value of a recursive call on `left`. In the case of a function with more than one argument, the second and subsequent arguments are passed unchanged to the recursive calls.

The symbol given after the argument list (here: `equal`) is the equivalence relation that the function maintains on `expr=`; that is what is used for `expr-fn=` in the functional instantiation.

7 Drawbacks of functional instantiation

There is no guarantee that the induction constraints for the type work! As a simple example, if one neglected to write one of the two hypotheses for `expr-induct-binop`, one would have no trouble proving `expr-induct-thm`, but would be hard pressed to prove `env-irrelevant` by instantiating it.

[18] The macro doesn't need to use functional instantiation for these proofs, of course.

[19] This may appear to be an annoying restriction, but the in the author's experience, it is not in practice.

Variables in constraints of functions being instantiated are not allowed to be used at all in the goal being proved, so for example, the variable `left` may not appear in a goal to be proved by `defexprthm`. This hasn't proved to be a practical problem.

Experienced users will recognize the similarity between the constraints on `expr-induct` and the induction machine for functions like `free-vars` and `eval-expr`[20]. This should be no surprise. The problem is when variables other than `expr` need to change during the induction. Sometimes it is necessary to combine induction schemes from different functions to prove a goal, and it is remarkable that ACL2 can usually do this. Such proofs cannot be proved by instantiating `expr-induct`; since it only has a single argument, arguments other than `expr` in recursive calls will be passed unchanged. One can define a more complex version of `expr-induct` that has a second parameter whose value in recursive calls is determined by still more constrained functions, but this may be more complicated than it is worth.

ACL2 attempts destructor elimination only after the goal has been fully simplified. If ACL2 did constructor elimination (triggered by a designated set of destructors) immediately after creating inductive subgoals, and if the user could suggest the new variable names to use, one could get the benefits of this style without the problems associated with functional instantiation. The author is not yet knowledgeable enough about the internals of ACL2 to experiment with that.

8 Comparison with related techniques

The use of constructors and destructors may remind the reader of the `ADD-SHELL` facility of NQTHM, or the `defstructure` macro defined in standard ACL2 book `books/data-structures/structures`. However, there really is not much in common. Those approaches introduce well-formedness predicates (something like `stack-p` for a stack data type) that (generally) must be used in proofs about the new data structures; the main point of this paper is to show how to avoid using such well-formedness predicates in favor of equivalence relations.

Eventually, one may want to combine the two techniques, using well-formedness predicates in guards for efficient execution and equivalence relations in theorems to simplify proofs, but the author has not explored that yet.

9 Conclusion

The author has presented an example of how to define an equivalence relation for a non-trivial data structure, together with its associated destructor and constructor functions, in order to be able to treat the data structure as an abstract

[20] ACL2 prints the induction scheme it is going to use before starting an inductive proof, following the phrase "We will induct". If it was suggested by only one term, then it is (usually) the induction machine for the functor of that term (or at least, so the author believes).

type. He also suggests a style of macro that makes it simpler to define functions that recur on such types; the advantage of the macro is that it automatically generates and proves the appropriate `defcong`. Finally, he shows how one may define another macro to make inductive proofs on the data type use the type constructors rather than the destructors, which seems to make intermediate (unsolved) goals more readable. None of this is new, but it may be helpful to see the various techniques worked out in this setting.

References

[1] Christopher T. Haynes Daniel P. Friedman, Mitchell Wand. *Essentials of programming languages*. McGraw-Hill, 1992.

[2] Matt Kaufmann and J Moore. An industrial strength theorem prover for a logic based on common lisp. *Transactions on Software Engineering*, 23(4):203–213, April 1997.