Specware in ACL2

Ben Selfridge

Kestrel Institute

selfridge@kestrel.edu

September 27, 2013

Kestrel Institute

- Non-profit computer science research center in Palo Alto, CA
- "Our mission is to advance the art and practice of **synthesizing provably correct code from high-level specifications**, to increase assurance, security, safety, productivity, and performance."

Kestrel Institute

Some of Kestrel's projects[1]:

- APAC - static analysis of Android apps
- CRASH - synthesis of concurrent garbage collectors
- HACMS - high assurance TCP/IP protocol stack
- Specware - Kestrel's flagship tool for general purpose synthesis

[1] http://www.kestrel.edu/home/projects/

Specware

- Software developed by Kestrel, used for many of their synthesis efforts
- Consists of:
  - A high-level functional programming/specification language
  - A logic
  - A transformation system for deriving programs from specifications

Specware

- Basic methodology: Start with a **high-level spec** for a program, then gradually **refine** it through transformations and morphisms
- Each of these transformations emits certain **proof obligations**
- Result: executable code that is correct by construction

Goal

- Current Specware implementation:
  - Translate code to executable Common LISP for execution
  - Translate code & theorems into Isabelle for verification
- We can unify this approach: use a single system (ACL2) for both execution and verification

Challenges

Specware:

- Higher order
- Supports general quantifiers
- Strongly-typed (Hindley-Milner, subtypes, dependent types)

Challenges

ACL2:

- Has no native data structures other than cons pairs
- First order
- Limited support for quantifiers
- Is untyped

Challenges

ACL2:

- Has no native data structures other than cons pairs
- First order
- Limited support for quantifiers
- Is untyped

But, sometimes minimalism is a blessing.

Addressing the challenges

My basic approach: Use ACL2 **macros** to **mimic high-level features** that Specware supports natively.

Make ACL2 “look like” Specware.

Addressing the challenges

Mapping from Specware language features to ACL2 constructs:

- Types become predicates
- Coproducts become calls to new `defcoproduct` macro
- Pattern matching: "case" statements become new `case-of` macro (thanks Sol)
- Ops become calls to new `defun-typed` macro
- Theorems become calls to new `defthm-typed` macro

(Bonus: other ACL2 users can use these macros to do typed functional programming.)

Coproducts

- Otherwise known as "union type" or "sum type" (think `data` declarations in Haskell)
- Not natively supported in ACL2
- Building on existing macros in the ACL2 books repository[2], we provide the `defcoproduct` macro
  - The Specware to ACL2 translator converts Specware coproduct definitions to `defcoproduct`
  - The macro introduces a number of theorems that will be used automatically by the prover
  - A simple definition of integer lists produces around 50 function definitions and theorems.

[2] Sol Swords' and William Cook's `defsum`

Coproducts

```
type IList =
  | INil
  | ICons Int * IList
```

becomes

```
(defcoproduct IList
  (INil)
  (ICons Int IList))
```

Coproducts

Things you get:

1. Type recognizers: `IList-p`, `ICons-p`, `INil-p`
2. Constructors/destructors: `ICons`, `ICons-arg-1`, etc.
3. Elimination rules, à la `(cons (car x) (cdr x)) = x`
4. Lots of other miscellaneous rules
    - `(NOT (EQUAL (ICONS ARG-1 ARG-2) ARG-2))`
    - `(IMPLIES (NOT (EQUAL ARG-1 (ICONS-ARG-1 X))) (NOT (EQUAL (ICONS ARG-1 ARG-2) X)))`
    - `(IMPLIES (INIL-P X) (EQUAL X (INIL)))`

Ops

- Specware's word for "functions"
- We provide the `defun-typed` macro
- Expands to a regular `defun` with guards on the input types
- Also includes a theorem restricting the output type of the function

Ops

```
op IAppend (x:IList, y:IList) : IList =
case x of
  | INil -> y
  | ICons (hd,tl) -> ICons (hd, (IAppend (tl,y)))
```

becomes

```
(defun-typed IAppend
  ((x IList) (y IList))
  IList
  (case-of x
   ((INil) y)
   ((ICons hd tl) (ICons hd (IAppend tl y)))))
```

Ops

The `defun-typed` for `IAppend` expands to:

```
(PROGN
 (DEFUN IAPPEND (A B)
   (DECLARE (XARGS :GUARD (AND (ILIST-P A) (ILIST-P B))
                   :VERIFY-GUARDS NIL))
   ...
   (CASE-OF A ((INIL) B)
              ((ICONS X XS) (ICONS X (IAPPEND XS B))))
   ...)
 (DEFTHM IAPPEND-TYPE
   (IMPLIES (AND (ILIST-P A) (ILIST-P B))
            (ILIST-P (IAPPEND A B)))
   :RULE-CLASSES (:TYPE-PRESCRIPTION :REWRITE))
 (VERIFY-GUARDS IAPPEND))
```

Theorems

- Theorems in Specware have to conform to the type rules
- We provide the `defthm-typed` macro to enforce this
- Expands to a defthm whose body is guard-verified

Theorems

```
theorem IAppend_associative is
  fa (x:IList, y:IList, z:IList)
    IAppend (IAppend (x,y), z) =
    IAppend (x, IAppend (y,z))
```

becomes

```
(defthm-typed IAppend_associative
  ((x IList)
   (y IList)
   (z IList))
  (equal (IAppend (IAppend x y) z)
         (IAppend x (IAppend y z))))
```

Theorems

The `defthm-typed` for `IAppend_associative` expands to:

```
(PROGN
  (DEFUN-TYPED IAPPEND_ASSOCIATIVE-BODY
    ((X ILIST) (Y ILIST) (Z ILIST))
    BOOL
    (EQUAL (IAPPEND (IAPPEND X Y) Z)
           (IAPPEND X (IAPPEND Y Z))))
  (DEFTHM IAPPEND_ASSOCIATIVE
    (IMPLIES (AND (ILIST-P X)
                  (ILIST-P Y)
                  (ILIST-P Z))
             (EQUAL (IAPPEND (IAPPEND X Y) Z)
                    (IAPPEND X (IAPPEND Y Z))))))
```

Polymorphism

Polymorphic lists in Specware:

```
Seq a =
  | SeqNil
  | SeqCons a * (Seq a)
```

Polymorphism

- Polymorphic types are somewhat of a higher-order notion
- So, how do we handle them in ACL2, a first-order system?
- Answer: we can mimic the notion of an “arbitrary type” with constrained predicates

Polymorphism

Introduce a function symbol, A-P, without a definition, which will represent our arbitrary type:

```
(ENCAPSULATE
  (((A-P *) => *))
  (LOCAL (DEFUN A-P (X) (DECLARE (IGNORE X)) T))
  (DEFTHM A-TYPE (BOOLEANP (A-P X))
    :RULE-CLASSES :TYPE-PRESCRIPTION))
```

If we prove a theorem about A-P, we have proven it for all types, since A-P is an arbitrary predicate.

Polymorphism

We can supply a coproduct definition with type variables, like so:

```
(defcoproduct Seq
  :type-vars (a)
  (SeqCons a (:inst Seq a))
  (SeqNil))
```

This call creates a macro, `Seq-instantiate`, which instantiates Seq for a specific type.

Polymorphism

Now, we can define a function over the `Seq a` type:

```
(defun-typed SeqAppend
  :type-vars (a)
  ((x (:inst Seq a)) (y (:inst Seq a)))
  (:inst Seq a)
  (case-of x
   (((:inst SeqNil a)) y)
   (((:inst SeqCons a) hd tl)
    ((:inst SeqCons a) hd ((:inst SeqAppend a) tl y)))))
```

This creates a macro, `SeqAppend-instantiate`, instantiating `SeqAppend` for a specific type.

Polymorphism

Finally, we can define a theorem over the `Seq a` type:

```
(defthm-typed SeqAppend_Associative
  :type-vars (a)
  ((x (:inst Seq a))
   (y (:inst Seq a))
   (z (:inst Seq a)))
  (equal ((:inst SeqAppend a)
          ((:inst SeqAppend a) x y)
          z)
         ((:inst SeqAppend a)
          x
          ((:inst SeqAppend a) y z))))
```

Polymorphism

- To instantiate `SeqAppend_Associative` for any specific type, we have to prove it for that type!
- But if we prove it for the arbitrary type `A-P`, we should be able to use that in the proof
- We use functional instantiation to do just that, and the proof goes through automatically

Proofs and ACL2 Hints

- When proofs don't go through automatically, we need to give hints to the prover
- We use the `proof ACL2` pragma in MetaSlang to accomplish this
- Like the case for Isabelle, the hints are written in the target language (ACL2) using ACL2's `:hints` construct

Proofs and ACL2 Hints

```
theorem ILength_of_IAppend is
  fa (a:IList,b:IList)
    ILength(IAppend(a,b)) = (ILength(a) + ILength(b))

proof ACL2 ILength_of_IAppend
  :enable (IAppend ILength)
end-proof
```

becomes...

Proofs and ACL2 Hints

```
(defthm-typed ILength_of_IAppend
  ((a IList)
   (b IList))
  (equal (ILength (IAppend a b))
         (+ (ILength a) (ILength b)))
  :enable (IAppend ILength))
```

For simple theorems, `:enable` hints are usually all that's needed.

Worked Examples

- DeMorgan's Laws
- Binary Trees
- Insertion sort (using `ILists`)
- Polymorphic Lists (`Seq`) - incomplete, but works so far

Advantages of ACL2 over Common Lisp/Isabelle

- Unified approach to execution/verification
- ACL2's minimalism makes it more flexible
  - Fewer type-related "paradigm clashes" since ACL2 doesn't really have types
  - Example: types are predicates, so subtypes are just stronger predicates!
- High degree of proof automation
- Very efficient execution
- Extensive collection of useful libraries
- Extremely mature prover with lots of useful features & community support

Future Work

Completing the big picture...

- Higher-order functions
    - This can be faked with macros
    - Similar approach as polymorphic types
    - Currying?
- Morphisms/transformations
    - This may be relatively straightforward
    - We just haven't tried it