A Macro for Reusing Abstract Functions and Theorems

Sebastiaan J.C. Joosten Bernard van Gastel Julien Schmaltz
Open University of the Netherlands
{bas.joosten,Bernard.vanGastel,Julien.Schmaltz}@ou.nl

Even though the ACL2 logic is first order, the ACL2 system offers several mechanisms providing users with some operations akin to higher order logic ones. In this paper, we propose a macro, named `instance-of-defspec`, to ease the reuse of abstract functions and facts proven about them. `Defspec` is an ACL2 book allowing users to define constrained functions and their associated properties. It contains macros facilitating the definition of such abstract specifications and instances thereof. Currently, lemmas and theorems derived from these abstract functions are not automatically instantiated. This is exactly the purpose of our new macro. `instance-of-defspec` will not only instantiate functions and theorems within a specification but also many more functions and theorems built on top of the specification. As a working example, we describe various fold functions over monoids, which we gradually built from arbitrary functions.

1 Introduction

The primary goal of this paper is to provide a way to reason abstractly in ACL2, while being able to specialise later. This kind of reasoning is necessary in the development of large modular proofs. Our main contribution is a macro, called `instance-of-defspec`, providing a convenient way to manipulate abstract functions and their properties. In our approach, we build generic theorems, and use our macro to apply them on more specific instances, that is: reuse proofs. For this to work, we reuse functions as well, by instantiating generic functions with more specific ones. While both function- and proof reuse are possible to some extent in ACL2, we provide a single macro to do both in a way that is more convenient than the existing solutions. Aside from the theoretical example presented in our paper, we briefly discuss the use of our solution in the development of a generic theory of communication networks, called GENOC [10]. This effort is a large and modular proof development of about fifty thousands lines of ACL2 code. Our macro already provides a thousand lines reduction of our code base.

This paper uses monoid-operations and fold operations as leading examples. A monoid is simply a closed and associative operation with an identity element, for which we will write $\circ$ and 0 respectively. A fold operation is an operation that changes a list $(a_0 \dots a_k)$ into a value $(a_0 \circ \dots \circ a_k)$. The leading example is not of particular importance on its own, but used to illustrate the macro `instance-of-defspec`.

Using the `encapsulate` environment, ACL2 provides a way to hide function definitions, while preserving certain function properties. We can then prove a theorem about such functions, using only the function properties in the proof. If we would then write a new function that satisfies all the properties used in the proof, we know that the function will satisfy the theorem as well. However, even though we know that the function satisfies the theorem, the only way to actually use this knowledge in further proofs is by telling it to ACL2 in the form of a new theorem. Using an uninterpreted function in the definition of a concrete function raises similar issues. We provide a macro called `instance-of-defspec`, that will allow ACL2 users to instantiate abstract functions while being able to use higher order theorems and functions.

R. Gamboa and J. Davis (Eds.): ACL2 Workshop 2013 (ACL2 '13).
EPTCS 114, 2013, pp. 29–41, doi:10.4204/EPTCS.114.3

The name `instance-of-defspec` derives from the word `defspec`, which already is a part of ACL2 [9]. In essence, `defspec` is a labeled `encapsulate` environment, such that the functions which are kept local to the `encapsulate` can be referred to as a single package. The advantage of `defspec` is that other functions can be declared to be an instance of this `defspec`. The current way of doing this in ACL2 is by using `DefInstance`, which is a part of `defspec`. This enables us to prove the 'higher order' statement that addition is a monoid. One could then use the `:functional-instance` hint, which was already implemented in Nqthm[1], to prove a specific theorem if it was already proven in a more general context. For example, we can use it to prove a theorem for addition which was already proven for a monoid efficiently.

Our approach combines these previous solutions. At a call of `instance-of-defspec`, the logical world is searched for functions and theorems that depend on the abstract `defspec`. These are copied for the concrete instance. Using `DefInstance` and the `:functional-instance` hint, the proof of every theorem is repeated for the instance without the computational burden.

Since `instance-of-defspec` will copy functions, our approach offers an alternative to `defattach`. Kaufmann and Moore [5] explain how this can be used to add an executable function, to for instance the abstract monoid, such that we can execute it. Unfortunately, `defattach` only allows one such attachment per function, and it does not influence the logical world. Our macro can be seen as an extension to `DefInstance` which combines all of these solutions, making a copy of functions instead of using `defattach` thus actually applying the change in the logical world.

As a leading example for reasoning abstractly, we develop a small theory about monoids. We prove that $(x_0 \circ (x_1 \circ (\ldots \circ (x_n \circ 0))))) = (((x_0 \circ x_1) \circ \ldots) \circ x_n)$ for $n \geq 0$, using any associative operation for $\circ$ with identity element 0. The benefit of such a proof is that we can apply it to arithmetic addition with zero, to multiplication with one, or to appending lists with the empty list.

In this paper, we present this example explaining how `instance-of-defspec` can be used in Section 2. In Section 3 we look at the inner workings of the macro, explaining limitations and opportunities in our approach. We also describe how `instance-of-defspec` can be used to add extra arguments to a function. Similar approaches are compared to `instance-of-defspec` in Section 4

2 Monoids as an example usage

Definition 1 (Monoid) *A* monoid $(0, \circ)$ *on the domain D is an operator* $\circ$ *and a constant* 0 *such that:*

- $\circ$ *is closed: for all* $a, b \in D$ *we have* $(a \circ b) \in D$.
- $\circ$ *is associative:* $(a \circ b) \circ c = a \circ (b \circ c)$ *for* $a, b, c \in D$.
- $0 \in D$ *is an identity element:* $0 \circ a = a = a \circ 0$.

In this section, we build a monoid by adding the three constraints from the definition one at a time. On the unrestricted operator $\circ$, we define different implementations of a fold operation. We will show that they are equivalent for monoids.

This section is self-containing, that is: the ACL2 interpreter should accept the inputs given in this section as-is. For additional theorems and examples, or for less typing, the reader may use the file `closedMonoid.lisp`.

First, we include our instance-of-defspec book.

```
1 (include-book "instance-of-defspec")
```

This book includes an entirely abstract function called `binary-function`, which was encapsulated in a `defspec` called `binary`. We repeat the definition of `binary` here. Note that this event cannot be entered in ACL2 here, since `binary` is already present in our instance-of-defspec book.

```
(defspec binary ((binary-function (x y) t))
  (local (defun binary-function (x y) (cons x y))))
```

Here we use `defspec`, together with `local`. The actual implementation of a `local` event is unknown to ACL2 outside the `defspec`. The only thing ACL2 knows about `binary-function` is that it takes two arguments. Hence we can regard `binary-function` as an arbitrary binary function, despite the chosen implementation of `cons` here.

2.1 Reusing functions

The intuition of the fold operation is that it transforms a list $(x_1 \dots x_n)$ into a value $x_1 \circ \dots \circ x_n$ for any binary operator $\circ$. For transforming the empty list, we need some sort of identity element to build upon. For this reason, we provide a first element x_0. We define `foldl` and `foldr`, which differ in the placement of the brackets. In the case of `foldl`, the brackets are written as $((((x_0 \circ x_1) \circ x_2) \circ x_3) \circ \dots)$. In the case of `foldr` and `foldr1`, brackets are written as: $x_1 \circ (x_2 \circ (x_3 \circ (\dots \circ x_0)))$. For `foldr1` we require that the list has at least one element. For `foldr` (`foldl`) we supply the last (first) element.

Note that, in the end, we will prove that these fold operations all are equivalent, even when providing an identity element as the first element. For this proof, we need associativity and an identity element, which is precisely the requirement of a monoid.

```
(defun foldr (x xs)
  (if (atom xs) x (binary-function (car xs) (foldr x (cdr xs)))))

; Alternatively, we can 'omit' the first element:
(defun foldr1 (xs)
  (if (atom (cdr xs)) (car xs)
    (binary-function (car xs) (foldr1 (cdr xs)))))

(defun foldl (x xs)
  (if (atom xs) x
    (foldl (binary-function x (car xs)) (cdr xs))))
```

Since the focus of this article is the use of `instance-of-defspec`, we do not proceed by proving properties of the fold functions, but show how to actually use these fold functions. A trivial example of a binary function is `cons`. We can instantiate `binary-function` (provided by the `defspec` encapsulate `binary`) with `cons` under a list of substitutions as follows:

```
(instance-of-defspec binary cons '((binary-function cons) (foldr  cons-foldr)
                                   (foldr1 cons-foldr1)   (foldl  cons-foldl)))
```

This instantiates our fold functions as executable functions:

```
ACL2 !> (cons-foldr 'a '(b c))
(B C . A)
ACL2 !> (cons-foldr1 '(a b c))
(A B . C)
```

```
ACL2 !> (cons-foldl 'a '(b c))
((A . B) . C)
```

Now lets add the assumption that our binary function is closed. That is: there is some domain, and if both arguments to the binary function belong to it, so does its result.

We encapsulate it using a `defspec`, and instantiate it as a binary operation. We will explain the use of `defspec` in Section 3.1. You may think of it as an `encapsulate` environment that hides the local definitions and provides the notion of `closed-binop` in terms of `c-domainp`, `c-binary-function` and the theorem (to be seen as a property) `closed-binop-closed`.

```
(defspec closed-binop ((c-domainp (x) t)
                       (c-binary-function (x y) t))
  (local (defun c-domainp (x) (integerp x)))
  (local (defun c-binary-function (x y) (+ x y)))
  (defthm closed-binop-closed
    (implies (and (c-domainp x) (c-domainp y))
             (c-domainp (c-binary-function x y)))))
(instance-of-defspec binary c) ; choose c (for closed) as the prefix symbol here
```

In the last statement, we instantiate `closed-binop` as an arbitrary binary operator: we use abstract functions as the instantiation of other abstract functions. Even though a closed binary function is an abstraction itself, it is an instantiation of the - more general - binary function.

Note that in this case, we did not specify the list of replacements. We do not have to: the second argument `c` is used as the default prefix. Hence, `c-foldr` is the instantiation of `foldr` with `c-binary-function` as `binary-function`:

```
ACL2 !> :pf c-foldr
(EQUAL (C-FOLDR X XS)
       (IF (CONSP XS)
           (C-BINARY-FUNCTION (CAR XS)
                              (C-FOLDR X (CDR XS)))
           X))
```

In fact, we could have done the same with `cons`. The following would have been a shorter notation for the same instruction we gave earlier:

```
(instance-of-defspec binary cons '((binary-function cons)))
```

2.2 Reusing theorems

We have seen how to reuse functions using `defspec`. We can do the same trick for theorems. In the context of a closed operation, we show that the repetitive application of this operation again yields an element in its domain. For brevity, we only show this for the `foldr1` operation:

```
(defun list-domainp (xs)
  (if (endp xs) t
    (and (c-domainp (car xs)) (list-domainp (cdr xs)))))

(defthm foldr1-closed
```

```
29   (implies (and (list-domainp xs) (consp xs))
30            (c-domainp (c-foldr1 xs))))
```

We reuse this theorem for a semigroup. A closed associative operator $\circ$ is called a semigroup. Note that a semigroup is like a monoid, but without the identity element 0. More formally: a monoid $(0, \circ)$ is a semigroup $\circ$ with identity element 0. So once again, we write a `defspec`, and specify that it is an instance of a closed binary operator.

```
31 (defspec semigroup ((sg-c-domainp (x) t)
32                     (sg-c-binary-function (x y) t))
33   (local (defun sg-c-domainp (x) (integerp x)))
34   (local (defun sg-c-binary-function (x y) (+ x y)))
35   (is-a closed-binop sg semigroup-is-a-closed-binop)
36   (defthm semigroup-assoc
37     (implies (and (sg-c-domainp x)
38                   (sg-c-domainp y)
39                   (sg-c-domainp z))
40              (equal (sg-c-binary-function x (sg-c-binary-function y z))
41                     (sg-c-binary-function (sg-c-binary-function x y) z)))))
42 (instance-of-defspec closed-binop sg) ; reuse the fold operators (again)
```

Note that we used the macro `is-a` inside our semigroup. This copies the theorems from the closed binary operator into the current `defspec`. By doing so, we ensure that the previously defined specification of `closed-binop` will be copied to `semigroup`, since this is required to prove that a `semigroup` is an instance of `closed-binop`. The `is-a` macro is auxiliary to `instance-of-defspec`. Its implementation will be discussed in Section 3.5. In this case, the following theorem is generated in place of the `is-a` macro:

```
ACL2 !> (OLDSPEC 'CLOSED-BINOP 'SEMIGROUP-IS-A-CLOSED-BINOP 'SG () (W STATE))
((DEFTHM SEMIGROUP-IS-A-CLOSED-BINOP-0
         (IMPLIES (IF (SG-C-DOMAINP X)
                      (SG-C-DOMAINP Y)
                      'NIL)
                  (SG-C-DOMAINP (SG-C-BINARY-FUNCTION X Y)))))
```

Now we try to prove that `foldr1` is `foldl`. Note that this was not the case for `cons-foldr1` and `cons-foldl`. We need to prove it for the semigroup folds: `sg-c-foldr1` and `sg-c-foldl`.

```
43 (defthm foldr1-is-foldl
44   (implies (and (sg-c-domainp x) (sg-c-domainp y)
45                 (sg-list-domainp xs))
46            (equal (sg-c-foldr1 (cons x xs))
47                   (sg-c-foldl x xs))))
```

If we look at the proof output, we will find that the proof has used a theorem we did not define ourselves, but which was automatically copied based on the theorem we added to the closed binary operator:

```
ACL2 !> :pf (:REWRITE SG-FOLDR1-CLOSED)
(IMPLIES (AND (SG-LIST-DOMAINP XS) (CONSP XS))
```

```
          (SG-C-DOMAINP (SG-C-FOLDR1 XS)))
```

Without the theorem `foldr1-closed`, and thus the automatically derived theorem `sg-foldr1-closed`, the proof attempt of `foldr1-is-foldl` would have failed.

2.3 On monoids

As promised, we end with a theory about monoids. A monoid is a semigroup with an identity element. To take care of the names, we use a renaming constant. Note that we can use renaming in the macro `is-a`, exactly like in `instance-of-defspec`.

```
(defconst *monoid-renaming*
  '((sg-c-domainp         mon-domainp)      (sg-c-foldr  mon-foldr)
    (sg-c-binary-function mon-binop)        (sg-c-foldr1 mon-foldr1)
    (sg-list-domainp      mon-list-domainp) (sg-c-foldl  mon-foldl) ))

(defspec monoid ((mon-domainp (x) t) (mon-binop (x y) t)
                 (mon-id () t))
  (local (defun mon-domainp (x) (integerp x)))
  (local (defun mon-binop (x y) (+ x y)))
  (local (defun mon-id () 0))
  (defthm id-in-domain (mon-domainp (mon-id)))
  (is-a semigroup mon monoid-is-a-semigroup *monoid-renaming*)
  (defthm monoid-id-left
    (implies (and (mon-domainp x))
             (equal (mon-binop x (mon-id))
                    x)))
  (defthm monoid-id-right
    (implies (and (mon-domainp x))
             (equal (mon-binop (mon-id) x)
                    x))))
(instance-of-defspec semigroup mon *monoid-renaming*)
```

We introduce function `fold` which acts like `foldr1`, but without the requirement that its argument should be a `consp`.

```
(defun fold (xs) (if (atom xs) (mon-id) (mon-foldr1 xs)))
```

We end the book about monoids, the file `closedMonoid.lisp`, by proving equality between the different versions of `fold`, and giving another instantiation. These proofs are omitted here, but we encourage the reader to take a look.

3 Inner workings

As the sources will be made available with this publication, we do not reproduce them here. Instead, we highlight the main parts to give the reader a rough understanding of the code, and highlight the 'ugly bits' to illustrate the difficulties and limitations of our approach.

The macro `instance-of-defspec` effectively does three things:

1. Look up the `defspec` and prove that the provided instance is an instance of this `defspec` using `DefInstance`.
2. Look up every function that uses one of the `defspec` functions, and copy it as an instantiated function.
3. Look up every theorem that use one of the functions from the previous step, and copy it as an instantiation.

In order to find the functions and theorems, we look at the ACL2 `world`. For this reason, we use make-event. In particular, `instance-of-defspec` is a macro that expands to:

```
`(make-event (instanceOf-defspec-fn ',spec ',prefix ,rename state)
  :check-expansion nil)
```

3.1 Obtaining the `defspec`

To reason abstractly, ACL2 provides the `encapsulate` environment. This environment allows the user to hide events, such that a particular theorem can be stated about a function, without allowing the definition of this function to enter the logical world. When proving further properties outside the encapsulate, the function is seen as an abstract function, since the function is unknown to the logical world.

To prove that a certain concrete function is an instance of this environment, `defspec` was developed by Sandip Ray and Matt Kaufmann. The system book `make-event/defspec.lisp` provides two parts: the first is a macro called `defspec`, which does exactly the same as an `encapsulate` environment, but then also provides a name for it. The second is a macro called `definstance`, which generates a theorem equivalent to stating that some implementation is an actual instantiation of the `defspec` named. We chose to build on this approach for the reason why Ray and Kaufmann developed it. In the comments of their code, they write [9]:

> The real problem is that ACL2 is a theorem prover for first order logic, not higher-order logic, while the statement we want to make is inherently a higher-order statement. (...)
>
> But having one macro that generates the instances will give the evaluators the ability to trust it, rather than hand-coded proofs that all the "corresponding" constraints are satisfied. The hope is that with a lot of use the macros here will be conventionally thought of as higher-order representations.

To obtain the `defspec`, we use `acl2::decode-logical-name` to skip to the place where the defspec was declared, and lookup the corresponding encapsulate event. In the case of `closed-binop` as written in the previous section, it might look like this:

```
(EVENT-LANDMARK GLOBAL-VALUE 8844
                (ENCAPSULATE (C-DOMAINP C-BINARY-FUNCTION)
                             . :COMMON-LISP-COMPLIANT)
                ENCAPSULATE
                ((C-DOMAINP (X) T)
                 (C-BINARY-FUNCTION (X Y) T))
                (LOCAL (DEFUN C-DOMAINP (X) (INTEGERP X)))
                (LOCAL (DEFUN C-BINARY-FUNCTION (X Y) (+ X Y)))
                (DEFTHM CLOSED-BINOP-CLOSED
                        (IMPLIES (AND (C-DOMAINP X) (C-DOMAINP Y))
                                 (C-DOMAINP (C-BINARY-FUNCTION X Y)))))
```

From this, we obtain the corresponding functions by taking the `cadar` of the third element. In this case, it returns `(C-DOMAINP C-BINARY-FUNCTION)`. At this point, we would like to note that we do not know whether this is an appropriate way to find the encapsulated functions. We just chose this to identify the `defspec` because it consistently returned these functions in multiple tests.

Instantiating the `defspec` is done with a `definstance`. At the instantiation of `closed-binop` as a semigroup, we generated the following statement:

```
(DEFINSTANCE CLOSED-BINOP SG-CLOSED-BINOP
       :FUNCTIONAL-SUBSTITUTION ((C-DOMAINP SG-C-DOMAINP)
                                 (C-BINARY-FUNCTION SG-C-BINARY-FUNCTION)))
```

This is the first statement generated. It is also the main proof obligation: it will fail when ACL2 cannot prove that the instance presented is an implementation of the `defspec`.

3.2 Obtaining functions

Once we have found the functions defined in the `defspec`, we look up all functions that depend on these. Note that we apply this transitively. That is: if we use the abstract function f inside g, and g is used inside h, then h has to be instantiated as well. To do so, we wrote function `get-derived-funs`.

Function `get-derived-funs` goes through the `world` multiple times, keeping track of a list of 'discovered' functions while looking for functions that use one of these and adding them to the list. The search for functions is done by looking for `DEF-BODIES` in the `world`, which ensures that macros have been eliminated from the definition. We terminate this search once the list does not grow any further. For each separate function, we stop searching once we hit the definition of that function. The rationale behind this is that a function cannot be used before it was declared. Since the procedure will just be used in a `make-event`, we can leave it in `:program` mode, saving us the trouble to prove termination. We take care not to add functions twice, which does provide termination: there are a finite number of functions in the current logical world.

When copying the function, we wish to be as similar to the original function as possible. However, we have to replace the abstract functions for their instantiations. To do so, we expand all macros in the function body using the `:trans` macro. After this, we replace the functions with their definitions using a function we called `replacefns`. This function takes a list of desired substitutions, and a list of terms in which this replacement should take place.

```
ACL2 !> :replacefns ((foo bar) (bar foo))
               ((+ ((lambda (foo j) (foo foo j)) x y) (bar x y)))
((+ ((LAMBDA (FOO J) (BAR FOO J)) X Y) (FOO X Y)))
```

Copying our function as identically as possible has the advantage of copying documentation and other parameters as well. There are cases where the generated `defun` fails, which occur when ACL2 fails to prove guards or termination of the generated function.

3.3 Obtaining theorems

Obtaining the theorems happens in a similar way. The main difference is that the list of functions does not grow while searching for theorems that use these functions. We can therefore find all theorems in one pass through the `world`.

When generating the theorems, we do not try to let the copied theorem mimic the original, as we did with functions. In the case of functions, it was possible that ACL2 tried to prove something (like guards

or termination), but failed. In the case of theorems, we want to prevent this from happening, by making sure that ACL2 does not redo an entire proof. To achieve this, we use the `:functional-instance` hint. Also, we do not try to copy the original event, but look for its effect on the logical world and copy the effect.

The theorem `foldr1-closed` from Section 2.2 is stored in the `world` as:

```
(...
 (FOLDR1-CLOSED THEOREM IMPLIES
                (IF (LIST-DOMAINP XS) (CONSP XS) 'NIL)
                (C-DOMAINP (C-FOLDR1 XS))) ...
 (FOLDR1-CLOSED CLASSES (:REWRITE))
 ...)
```

Note that the theorem and the classes are stored in different `world` items. We obtain the theorems by combining these two parts in the `world`. The first holds the (translated, macro-free) theorem, and the second holds the rule-class(es). Some rule-classes are stored with extra attributes. For example, a typing rule will define a typed term, and a forward-chaining rule has a trigger-term.

The `:functional-instance` hint will generate various subgoals, which can be proven using the proof generated by a previously proven `definstance` theorem (which is the first theorem we generate, even before copying the functions). We also add the `:in-theory` hint, giving a theory that contains only the newly generated functions. Although we do not know how to prove that this is a sound way to copy theorems, we can at least give the anecdotical evidence that we have not found an example where the proof attempt for a generated theorem failed.

While developing `instance-of-defspec`, we found it useful to be able to keep track of the theorems defined. With relatively little effort, we have written the macro `symbol-lemmas` which, like `:pl`, shows theorems containing its argument. The main difference being that the former shows all theorems, while the latter only shows those in which its argument is a trigger.

3.4 Adding arguments

The main use of `instance-of-defspec` is to provide a single method to create functions like `map`, and corresponding theorems. An issue we did not foresee was that some of the functions we would like to map, though effectively unary, are actually binary.

We have solved this by allowing for lambda functions in the function substitution. As an example, we investigate a generic function that checks whether all elements in some list satisfy some predicate:

```
(defspec list-predicate ((predicate (x) t))
  (local (defun predicate (x) x)))
(defun predicate-listp (lst)
  (if (atom lst) (null lst) ; require true-lists
    (and (predicate (car lst)) (predicate-listp (cdr lst)))))
```

Now suppose we want our predicate to be `member-equal`. This would enable us to test whether all elements in some list occur in some other list, which is exactly the condition for subsets. We can create a new function `subset-equal` (written without a `p` to prevent collision with the builtin `subsetp-equal`):

```
(instance-of-defspec list-predicate members
  '((predicate (lambda (x) (member-equal x y)))
    (predicate-listp (lambda (lst) (subset-equal lst y)))))
```

At this point, we require the lambda expression to have exactly the same argument as the arguments of the function it replaces:

```
ACL2 !> (instance-of-defspec list-predicate members
          '((predicate (lambda (x) (member-equal x y)))
            (predicate-listp (lambda (xs) (subset-equal xs y)))))

ACL2 Error in COPYFUN:  The lambda construct
(LAMBDA (XS) (SUBSET-EQUAL XS Y)) takes as input (XS), which should
be an exact match of the original arguments of the original function:
(LST)
```

This syntactic limitation will hopefully prevent users unintentionally swapping arguments, and helps identify which variables are substituted.

3.5 The `is-a` macro

To prove something is an instance of some `defspec`, we need to prove all theorems included in that `defspec`. If we want to add a property to a previous `defspec`, we proceed by creating a new `defspec`, and indicating that the new is an instance of the former. One way to do this, is by using `definstance`, and another is by using `is-a`. The main advantage of the latter is that theorems are be copied individually, and can remain enabled. This results in a different behaviour of subsequent proof attempts. Another advantage is that the notation of `is-a` is very similar to that of `instance-of-defspec`.

The implementation of `is-a` is a lot like `definstance`, in the sense that it uses the function `constraint` defined in the `defspec` book to get all theorems that have to hold. For every function it finds in the `defspec`, the function `constraint` is called which returns a theorem that must hold. While it would be possible to use `is-a` outside of a `defspec`, it would make more sense to use `instance-of-defspec` there.

4 Discussion

Related work Before we started working with `instance-of-defspec`, we used ACL2's macro system to avoid code duplication. This approach does not involve inspecting the world, or `make-event`. We just used plain macros that write out 'instances'. Apart from functions, we used these macro's to generate theorems as well. In this approach we lose the proof obligation that the proposed instance is actually an instance, or even having to create an encapsulated environment for the abstract structure. The price we pay, however, is having to do full proofs for all instantiated theorems, which can be a heavy computational burden. Also, we found the current approach of writing theorems about abstract instances directly in the logical world to be more convenient than writing theorems in a macro. Carl Eastlund and Matthias Felleisen make a case against using ACL2's macros, proposing hygienic macros [2]. If we were to use hygienic macros for this purpose, writing out functions and theorems using a macro could be more convenient. We would still need to redo theorems.

A very promising alternative to our approach was presented by Gamboa and Patterson [4]. The authors introduce a way to use polymorphism in ACL2. One of the downsides is the use of a `stobj` called `memory`. In addition, it is presented as an alternative to the current `encapsulate` environments, instead of building on what is already present in ACL2.

Although we could not find a paper on it, there is a macro called `def-functional-instance` inside the ACL2 tools directory which provides an easier way to instantiate previous theorems using the `:functional-instance` hint. The problem is solved more rigorously by Moore [8], by providing an alternative to the `:functional-instance` hint which computes its substitutions automatically. Our `instance-of-defspec` macro does all this, and these two previous solutions stress the importance of doing so.

For conveniently instantiating functions, the macro by Martín-Mateos et al. [7] is most similar to ours. The same instantiations are performed: both functions and theories can be reused. The authors did not use the `defspec` environment, nor do their macros rewrite proof hints to aid the instantiated proofs. Also, at some point in the process the user is required to specify exactly which events to copy (these must be `defun` and `defthm` events). A macro call must be placed around these events, and the output of this macro should be copied manually. The user then obtains a program which will define a constant that can be used for reuse. One of the reasons that their approach was less convenient than ours, is that ACL2 did not have some of its current features, especially `make-event`, at the time. Indeed, such improvements are present in later work. In the code accompanying the more recent paper from this group [6], many of the improvements suggested in the earlier paper [7] have been made. Fundamental differences still are:

- The macro by Martín-Mateos only copies a handful of events, where our macro will take any event that rewrites itself to a `defun` or a `defthm`, which includes those handled by Martín-Mateos.
- The macro by Martín-Mateos requires the general theory to be wrapped inside a macro entirely. Our approach only requires this for the local `defspec`, allowing the general theory to be distributed over multiple books.
- Our macro allows arguments to be added to function calls without changing the original function.
- Leaving the definitions of the instantiation enabled might cause the macro by Martín-Mateos to fail. In our approach the theorem prover is guided by automatically generated hints. Since both approaches use the `:functional-instance` hint, we expect this to be something which can be resolved rather easily.

Carl Eastlund and Matthias Felleisen built a module system on top of ACL2, using the Dracula environment [3]. In this system, the abstract entities are called interfaces, which have signatures and contracts. This compares well to the `defspec` which is already built in ACL2. A major difference is that, in ACL2, a function has to be declared `local`, and the theorems must hold for it. In essence, the user must provide a witness for the `defspec`, while the Dracula environment does not require this. The interfaces can be instantiated as modules, or reused in modules, by using export and import commands respectively. An advantage of Dracula Modular ACL is that modules are checked independently, which might improve the performance of ACL2 by reducing memory requirements. For our approach, one would have to write different files in order to get such behavior. We see two disadvantages to the modular approach.

- Reasoning about interfaces directly is not possible. A familiar proof states that there can only be one identity element for a monoid. In the modular approach, one would have to create a module which imports the interface, and exports another interface which also has the property that there is only one identity element. In their paper, Eastlund and Felleisen prove correctness of the executable modules. Hence the module that describes a property about the monoid interface will not be proven to be correct until a witness for the monoid is provided.
- The Dracula environment requires the user to trust both the ACL2 system and the module system, whereas in our approach all proofs are performed within the ACL2 system. Although we believe

the module system to be sound, bugs in ACL2 up to version 2.3 show that reasoning with encapsulated properties is particularly error-prone. The reader may, for instance, read the discussion on subversive recursions in the miscellaneous section of the ACL2 documentation (version 3 and up).

Use in practice Our group is involved in a large scale generic proof about Network-On-Chips, called GeNoC [10]. The proofs consists of almost fifty thousand lines of ACL2 code. In GeNoC all kinds of properties about classes of networks are proven. To maximise proof reusability, an important aspect is partially instantiating these generic proofs for classes of networks, and later fully instantiating the proofs for concrete instantiations of networks. Currently we use many hand coded functional substitution rules in the `definstance`. These are hard and error-prone to maintain. By using the `instance-of-defspec` construct we already reduce our effort to maintain our codebase. Also a lot of instantiations of generic theorems can be removed, because they are automatically generated. Both these changes reduced our code base by one thousand lines.

As `instance-of-defspec` copies every function and theorem based on a `defspec`. Because of this, instantiating a defspec early in a proof session and instantiating it late in the session could have different meanings. For this reason, we ensured that it would be possible to instantiate the same set of functions as a defspec twice. One might worry that all extra instantiated definitions can clutter up the logical world and namespace. In practice, however, we only needed to instantiate each set of functions as one defspec, and did not run into problems with a cluttered up namespace. Nevertheless we provided a feature, per request by one of the reviewers, which allows you to not copy some theorems. To do this, add a theorem to the rename list without providing a new name for it. For instance, if `sometheorem` would normally be instantiated by a call to `instance-of-defspec`, add `(sometheorem newname)` to use `newname` as the new name for this theorem, or `(sometheorem)` to not copy `sometheorem` at all.

Future work Using our macro in *all* instances in GeNoC still presents a problem. We cannot handle function definitions inside `encapsulate` environments correctly. To find functions to use as `local` witness, we need to look inside the `encapsulate` environment, which is forbidden. For this reason, the `defun-sk` and `defevaluator` events, which rewrite to events including the `encapsulate`, are not supported.

In the current example, we use `is-a` to include a previous `defspec` into a new one. In practice, not all functions may be duplicated in this step.

We can already illustrate how this problem arises by a modification in the monoid example. Suppose that in Section 2.1, instead of adding the `domainp` function to `closed-binop`, we would have used `predicate` and the corresponding generic function `predicate-listp` (possibly included from a different file). Proving that `closed-binop` is an instance of `binary` will not be a problem. Proving that the `semigroup` is an instance of `closed-binop`, however, requires instantiating `predicate` with `sg-c-domainp` *while* instantiating the semigroup operator with `closed-binop`. We are in the process of finding a convenient syntax for this, and we will provide a way to do this in the near future.

5 Conclusions

The macro `instance-of-defspec` enables us to reuse abstract functions and facts proven about them. It automatically instantiates lemmas and theorems derived from the abstract functions. We are convinced that together with `defspec`, our macro provides a way to reason abstractly in ACL2, while being able to specialise later. The great variety of similar approaches shows that there is a strong need for abstract

reasoning in ACL2. This variety of solutions also suggests that no approach has become the standard for the community. We believe that our approach is more convenient than the existing solutions, and we hope it will become a standard in the ACL2 community.

Acknowledgments We thank the reviewers for their detailed comments and apposite insights.

References

[1] Robert S Boyer, David M Goldschlag, Matt Kaufmann & J Strother Moore (1991): *Functional instantiation in first order logic*. In: *GROUP, UNIVERSITY OF GOTEBORG*, Citeseer, pp. 7–26, doi:10.1016/B978-0-12-450010-5.50007-4.

[2] C. Eastlund & M. Felleisen (2011): *Hygienic macros for ACL2*. *Trends in Functional Programming*, pp. 84–101, doi:10.1007/978-3-642-22941-1_6.

[3] Carl Eastlund & Matthias Felleisen (2009): *Making induction manifest in modular ACL2*. In: *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*, ACM, pp. 105–116, doi:10.1145/1599410.1599424.

[4] R. Gamboa & M. Patterson (2003): *Polymorphism in ACL2*. In: *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*.

[5] M. Kaufmann & J.S. Moore (2011): *How Can I Do That with ACL2? Recent Enhancements to ACL2*. In: *Tenth International Workshop on the ACL2 Theorem Prover and Its Applications*, 70, doi:10.4204/EPTCS.70.4.

[6] L. Lambán, F.J. Martín-Mateos, J. Rubio & J.L. Ruiz-Reina (2012): *Formalization of a normalization theorem in simplicial topology*. *Annals of Mathematics and Artificial Intelligence* 64(1), pp. 1–37, doi:10.1007/s10472-011-9274-6.

[7] F. J. Martín-Mateos, J. A. Alonso, M. J. Hidalgo & J. L. Ruiz-Reina (2002): *A generic instantiation tool and a case study: A generic multiset theory*. In: *Third International Workshop on the ACL2 Theorem Prover and Its Applications*, pp. 188–203.

[8] J S. Moore (2009): *Automatically computing functional instantiations*. In: *Proceedings of the Eighth International Workshop on the ACL2 Theorem Prover and its Applications*, ACM, pp. 11–19, doi:10.1145/1637837.1637839.

[9] S. Ray & M. Kaufmann (2006): *Defspec.lisp*. ACL2 books/make-event directory.

[10] Freek Verbeek & Julien Schmaltz (2012): *Easy Formal Specification and Validation of Unbounded Networks-on-Chips Architectures*. *ACM Transactions on Design Automation of Electronic Systems* 17(1), doi:10.1145/2071356.2071357.