Second-Order Functions and Theorems in ACL2

Alessandro Coglio
Kestrel Institute
http://www.kestrel.edu/~coglio

SOFT ('Second-Order Functions and Theorems') is a tool to mimic second-order functions and theorems in the first-order logic of ACL2. Second-order functions are mimicked by first-order functions that reference explicitly designated uninterpreted functions that mimic function variables. First-order theorems over these second-order functions mimic second-order theorems universally quantified over function variables. Instances of second-order functions and theorems are systematically generated by replacing function variables with functions. SOFT can be used to carry out program refinement inside ACL2, by constructing a sequence of increasingly stronger second-order predicates over one or more target functions: the sequence starts with a predicate that specifies requirements for the target functions, and ends with a predicate that provides executable definitions for the target functions.

1 The SOFT Tool

SOFT ('Second-Order Functions and Theorems') is a tool to mimic second-order functions and theorems [4] in the first-order logic of ACL2 [3]. Second-order functions are mimicked by first-order functions that reference explicitly designated uninterpreted functions that mimic function variables. First-order theorems over these second-order functions mimic second-order theorems universally quantified over function variables. Instances of second-order functions and theorems are systematically generated by replacing function variables with functions. Theorem instances are proved automatically, via automatically generated functional instantiations [5].

SOFT does not extend the ACL2 logic. It is an ACL2 library, available in the ACL2 community books, that provides macros to introduce function variables, second-order functions, second-order theorems, and instances thereof. The macros modify the ACL2 state only by submitting sound and conservative events; they cannot introduce unsoundness or inconsistency on their own. The main features of the macros are described and exemplified below; full details are in their documentation and implementation.

1.1 Function Variables

A *function variable* is introduced as

```
(defunvar fv (* ... *) => *)
```

where:

- *fv* is a symbol, which names the function variable.
- `(* ... *)` is a list of 1 or more `*`s, which defines the arity, i.e. type [6], of *fv*.

This generates the event

```
(defstub fv (* ... *) => *)
```

i.e. *fv* is introduced as an uninterpreted function with the given type. Furthermore, a `table` event is generated to record *fv* in a global table of function variables.

For example,

To appear in EPTCS.

```
(defunvar ?f (*) => *)
(defunvar ?p (*) => *)
(defunvar ?g (* *) => *)
```

introduce two unary function variables and one binary function variable. Starting function variable names with `?` provides a visual cue for their function variable status, but SOFT does not enforce this naming convention.

1.2 Second-Order Functions

SOFT supports three kinds of second-order functions: plain second-order functions, choice second-order functions, and quantifier second-order functions.

1.2.1 Plain Functions

A *plain second-order function* is introduced as

```
(defun2 sof (fv1 ... fvn) (v1 ... vm) doc decl ... decl body)
```

where:

- *sof* is a symbol, which names the second-order function.
- (fv_1 ... fv_n) is a non-empty list without duplicates of previously introduced function variables, whose order is immaterial, which are the function parameters of *sof*.
- The other items are as in `defun`: individual variables, optional documentation string, optional declarations, and defining body.
- $\mathsf{FV}(\mathit{body}) \cup \mathsf{FV}(\mathit{measure}) \cup \mathsf{FV}(\mathit{guard}) = \{fv_1, \ldots, fv_n\}$, where:
  - *measure* is the measure expression of *sof*, or `nil` if *sof* is not recursive.
  - *guard* is the guard of *sof* (`t` if not given explicitly in the declarations).
  - $\mathsf{FV}(\mathit{term})$ is the set of function variables that either occur in *term* or are function parameters of second-order functions that occur in *term*.

  I.e. the function parameters of *sof* are all and only the function variables that *sof* depends on.[1]

This generates the event

```
(defun sof (v1 ... vm) doc decl ... decl body)
```

i.e. *sof* is introduced as a first-order function using `defun`, removing the function variables. Furthermore, a `table` event is generated to record *sof* in a global table of second-order functions.

For example,

```
(defun2 quad[?f] (?f) (x)
  (?f (?f (?f (?f x)))))
```

introduces a non-recursive function to apply its function parameter to its individual parameter four times. The name `quad[?f]` conveys the dependency on the function parameter and provides a visual cue for the implicit presence of the function parameter when the function is applied, e.g. in `(quad[?f] x)`, but SOFT does not enforce this naming convention.

As another example,

[1] Thus, `defun2` could have been defined to have the same form as `defun`, i.e. without (fv_1 ... fv_n). However, the presence of the functions parameters parallels that of the individual parameters, and the redundancy check may detect user errors.

```
(defun2 all[?p] (?p) (l)
  (cond ((atom l) (null l))
        (t (and (?p (car l)) (all[?p] (cdr l))))))
```

introduces a recursive predicate (i.e. boolean-valued function) that recognizes `nil`-terminated lists whose elements satisfy the predicate parameter.

As a third example,

```
(defun2 map[?f_?p] (?f ?p) (l)
  (declare (xargs :guard (all[?p] l)))
  (cond ((endp l) nil)
        (t (cons (?f (car l)) (map[?f_?p] (cdr l))))))
```

introduces a recursive function that homomorphically lifts `?f` to operate on `nil`-terminated lists whose elements satisfy `?p`. The predicate parameter `?p` only appears in the guard, not in the body.

As a fourth example,

```
(defun2 fold[?f_?g] (?f ?g) (bt)
  (cond ((atom bt) (?f bt))
        (t (?g (fold[?f_?g] (car bt)) (fold[?f_?g] (cdr bt))))))
```

introduces a generic folding function on values as binary trees.

1.2.2 Choice Functions

A *choice second-order function* is introduced as

`(defchoose2` *sof* `(`bv_1 `...` bv_p`) (`fv_1 `...` fv_n`) (`v_1 `...` v_m`)` *body* *key-opts*`)`

where:

- *sof* is a symbol, which names the second-order function.
- `(`fv_1 `...` fv_n`)` are the function parameters, as in `defun2`.
- The other items are as in `defchoose`: bound variables, individual variables, constraining body, and keyed options.
- $\mathsf{FV}(body) = \{fv_1, \ldots, fv_n\}$.

This generates the event

`(defchoose` *sof* `(`bv_1 `...` bv_p`) (`v_1 `...` v_m`)` *body* *key-opts*`)`

i.e. *sof* is introduced as a first-order function using `defchoose`, removing the function variables. Furthermore, a `table` event is generated to record *sof* in the same global table where plain second-order functions are recorded.

For example,

```
(defchoose2 fixpoint[?f] x (?f) ()
  (equal (?f x) x))
```

introduces a second-order function constrained to return a fixed point of `?f`, if any exists.

1.2.3 Quantifier Functions

A *quantifier second-order function* is introduced as

```
(defun-sk2 sof (fv_1 ... fv_n) (v_1 ... v_m) body key-opts)
```

where:

- *sof* is a symbol, which names the second-order function.
- $(fv_1$ ... $fv_n)$ are the function parameters, as in `defun2` and `defchoose2`.
- The other items are as in `defun-sk`: individual variables, defining body, and keyed options.
- $\mathsf{FV}(body) \cup \mathsf{FV}(guard) = \{fv_1, \ldots, fv_n\}$, where *guard* is the guard of *sof* (`t` if not given explicitly in the `:witness-dcls` option).

This generates the event

```
(defun-sk sof (v_1 ... v_m) body key-opts)
```

i.e. *sof* is introduced as a first-order function using `defun-sk`, removing the function variables. Furthermore, a `table` event is generated to record *sof* in the same global table where plain and choice second-order functions are recorded.

For example,

```
(defun-sk2 injective[?f] (?f) ()
  (forall (x y) (implies (equal (?f x) (?f y)) (equal x y))))
```

introduces a predicate that recognizes injective functions.

1.3 Instances of Second-Order Functions

An *instance of a second-order function* is a function introduced as

```
(defun-inst f (fv_1 ... fv_n) (sof (fv_1' . f_1') ... (fv_m' . f_m')) key-opts)
```

where:

- *f* is a symbol, which names the new function.
- $(fv_1$ ... $fv_n)$ are optional function parameters. If present, *f* is a second-order function; if absent, *f* is a first-order function.
- *sof* is a previously introduced second-order function.
- $((fv_1$' . f_1') ... $(fv_m$' . f_m')) is an *instantiation* Σ, i.e. an alist whose keys fv_i' are distinct function variables, whose values f_i' are previously introduced function variables, second-order functions, or regular first-order functions, and where each f_i' has the same type as fv_i'. Each fv_i' is a function parameter of *sof*. The notation (*sof* $(fv_1$' . f_1') ... $(fv_m$' . f_m')) suggests the application of *sof* to the functions f_i'; since the function parameters of *sof* are unordered, the application is by explicit association, not positional. An instance of a second-order function is introduced as a named application of the second-order function; SOFT does not support the application of a second-order function on the fly within a term, as in the application of a first-order function. Not all the function parameters of *sof* must be keys in Σ; missing function parameters are left unchanged.
- *key-opts* are keyed options, e.g. to override attributes of *f* that are otherwise derived from *sof*.

- If *sof* is a plain function, $\mathsf{FV}(\Sigma(\mathit{body})) \cup \mathsf{FV}(\Sigma(\mathit{measure})) \cup \mathsf{FV}(\Sigma(\mathit{guard})) = \{fv_1, \ldots, fv_n\}$, where *body*, *measure*, and *guard* are the body, measure expression (nil if *sof* is not recursive), and guard of *sof*, and $\Sigma(\mathit{term})$ is the result of applying Σ to *term* (see below).
- If *sof* is a choice function, $\mathsf{FV}(\Sigma(\mathit{body})) = \{fv_1, \ldots, fv_n\}$, where *body* is the body of *sof*.
- If *sof* is a quantifier function, $\mathsf{FV}(\Sigma(\mathit{body})) \cup \mathsf{FV}(\Sigma(\mathit{guard})) = \{fv_1, \ldots, fv_n\}$, where *body* and *guard* are the body and guard of *sof*.

This generates a defun, defchoose, or defun-sk event, depending on whether *sof* is a plain, choice, or quantifier function. The event introduces *f* with body $\Sigma(\mathit{body})$, measure $\Sigma(\mathit{measure})$ (if *sof* is recursive, hence plain), and guard $\Sigma(\mathit{guard})$ (if *sof* is a plain or quantifier function). *f* is recursive iff *sof* is recursive: defun-inst generates the termination proof of *f* from the termination proof of *sof* using the techniques to instantiate second-order theorems described in Section 1.5.

Furthermore, defun-inst generates a table event to record *f* as the Σ instance of *sof* in a global table of instances of second-order functions. If *f* is second-order, defun-inst also generates a table event to record *f* in the global table of second-order functions.

$\Sigma(\mathit{term})$ is obtained from *term* by replacing the keys of Σ in *term* with their values in Σ. This involves not only explicit occurrences of such keys in *term*, but also implicit occurrences as function parameters of second-order functions occurring in *term*. For example, if the pair (?f . f) is in Σ, sof[...?f...] is a second-order function whose function parameters include ?f, and *term* is (cons (?f x) (sof[...?f...] y)), then $\Sigma(\mathit{term})$ is (cons (f x) (sof[...f...] y)), where sof[...f...] is the Σ' instance of sof[...?f...], where Σ' is the restriction of Σ to the keys that are function parameters of sof[...?f...]. The table of instances of second-order functions is consulted to find sof[...f...]. If the instance is not in the table, defun-inst fails: the user must introduce sof[...f...], via a defun-inst, and then re-try the failed instantiation.

For example, given a function

```
(defun wrap (x) (list x))
```

that wraps a value into a singleton list,

```
(defun-inst quad[wrap]
  (quad[?f] (?f . wrap)))
```

introduces a function that wraps a value four times.

As another example, given a predicate

```
(defun octetp (x) (and (natp x) (< x 256)))
```

that recognizes octets,

```
(defun-inst all[octetp]
  (all[?p] (?p . octetp)))
```

introduces a predicate that recognizes nil-terminated lists of octets.

As a third example,

```
(defun-inst map[code-char]
  (map[?f_?p] (?f . code-char) (?p . octetp)))
```

introduces a function that translates lists of octets to lists of corresponding characters. The replacement code-char of ?f induces the replacement octetp of ?p, because the guard of code-char is (equivalent to) octetp; the name map[code-char] indicates only the replacement of ?f explicitly.

As a fourth example,

```
(defun-inst fold[nfix_plus]
  (fold[?f_?g] (?f . nfix) (?g . binary-+)))
```

adds up all the natural numbers in a tree, coercing other values to 0.

As a fifth example, given a function

```
(defun twice (x) (* 2 (fix x)))
```

that doubles a value,

```
(defun-inst fixpoint[twice]
  (fixpoint[?f] (?f . twice)))
```

introduces a function constrained to return the (only) fixed point 0 of `twice`.

As a sixth example,

```
(defun-inst injective[quad[?f]] (?f)
  (injective[?f] (?f . quad[?f])))
```

introduces a predicate that recognizes functions whose four-fold application is injective.

1.4 Second-Order Theorems

A *second-order theorem* is a theorem whose formula depends on function variables, which occur in the theorem or are function parameters of second-order functions that occur in the theorem. Since function variables are unconstrained, a second-order theorem is effectively universally quantified over the function variables that it depends on. It is introduced via standard events like `defthm`.[2]

For example,

```
(defthm len-of-map[?f_?p]
  (equal (len (map[?f_?p] l)) (len l)))
```

shows that the homomorphic lifting of `?f` to lists of `?p` values preserves the length of the list, for every function `?f` and predicate `?p`.

As another example,

```
(defthm injective[quad[?f]]-when-injective[?f]
  (implies (injective[?f]) (injective[quad[?f]]))
  :hints
  (("Goal" :use
    ((:instance
      injective[?f]-necc
      (x (?f (?f (?f (?f (mv-nth 0 (injective[quad[?f]]-witness))))))
      (y (?f (?f (?f (?f (mv-nth 1 (injective[quad[?f]]-witness)))))))
     (:instance
      injective[?f]-necc
      (x (?f (?f (?f (mv-nth 0 (injective[quad[?f]]-witness)))))
      (y (?f (?f (?f (mv-nth 1 (injective[quad[?f]]-witness))))))
     (:instance
      injective[?f]-necc
```

[2] The absence of an explicit quantification over function variables in second-order theorems parallels the absence of an explicit quantification over individual variables in first-order theorems.

```
      (x (?f (?f (mv-nth 0 (injective[quad[?f]]-witness)))))
      (y (?f (?f (mv-nth 1 (injective[quad[?f]]-witness))))))
     (:instance
      injective[?f]-necc
      (x (?f (mv-nth 0 (injective[quad[?f]]-witness))))
      (y (?f (mv-nth 1 (injective[quad[?f]]-witness)))))
     (:instance
      injective[?f]-necc
      (x (mv-nth 0 (injective[quad[?f]]-witness)))
      (y (mv-nth 1 (injective[quad[?f]]-witness))))))))
```

shows that the four-fold application of an injective function is injective.

As a third example, given a function variable

```
(defunvar ?io (* *) => *)
```

for an abstract input/output relation, a predicate

```
(defun-sk2 atom-io[?f_?io] (?f ?io) ()
  (forall x (implies (atom x) (?io x (?f x))))
  :rewrite :direct)
```

that recognizes functions `?f` that satisfy the input/output relation on atoms, and a predicate

```
(defun-sk2 consp-io[?g_?io] (?g ?io) ()
  (forall (x y1 y2)
          (implies (and (consp x) (?io (car x) y1) (?io (cdr x) y2))
                   (?io x (?g y1 y2))))
  :rewrite :direct)
```

that recognizes functions `?g` that satisfy the input/output relation on `cons` pairs when the arguments are valid outputs for the `car` and `cdr` components,

```
(defthm fold-io[?f_?g_?io]
  (implies (and (atom-io[?f_?io]) (consp-io[?g_?io]))
           (?io x (fold[?f_?g] x))))
```

shows that the generic folding function on binary trees satisfies the input/output relation when its function parameters satisfy the predicates just introduced.

1.5 Instances of Second-Order Theorems

An *instance of a second-order theorem* is a theorem introduced as

`(defthm-inst` *thm* `(`*sothm* `(`fv_1 `.` f_1`) ... (`fv_n `.` f_n`)) :rule-classes ...)`

where:

- *thm* is a symbol, which names the new theorem.
- *sothm* is a previously introduced second-order theorem.
- `((`fv_1 `.` f_1`) ... (`fv_n `.` f_n`))` is an instantiation Σ, where each fv_i is a function variable that *sothm* depends on. The notation `(`*sothm* `(`fv_1 `.` f_1`) ... (`fv_m `.` f_m`))` is similar to `defun-inst`.
- The keyed option `:rule-classes ...` is as in `defthm`.

This generates the event

`(defthm` *`thm`* Σ(*`formula`*) `:rule-classes ... :instructions` *`proof`*`)`

where:

- *`formula`* is the formula of *`sothm`*.
- *`proof`* consists of two commands for the ACL2 proof checker to prove *`thm`* using *`sothm`*.

The first command of *`proof`* is

`(:use (:functional-instance` *`sothm`* `(`fv_1 f_1`) ... (`fv_n f_n`)` *`more-pairs`*`))`

i.e. *`thm`* is proved using a functional instance of *`sothm`*. The pairs that define the functional instance include not only the pairs that form Σ (in list notation instead of dotted notation), but also, in *`more-pairs`* above, all the pairs `(`*`sof f`*`)` such that *`sof`* is a second-order function that occurs in *`sothm`* and *`f`* is its replacement in *`thm`* (i.e. *`f`* is the Σ' instance of *`sof`*, where Σ' is the restriction of Σ to the function parameters of *`sof`*). These additional pairs are determined in the same way as when Σ is applied to *`formula`* (see Section 1.3): thus, the result of `(:functional-instance ...)` above is Σ(*`formula`*), and the main goal of *`thm`* is readily proved.

The use of the functional instance reduces the proof of *`thm`* to proving that, for each pair, the replacing function satisfies all the constraints of the replaced function. Since function variables are unconstrained, nothing needs to be proved for the `(`fv_i f_i`)` pairs. For each `(`*`sof f`*`)` pair in *`more-pairs`*, it must be proved that *`f`* satisfies the constraints on *`sof`*. If *`sof`* references another second-order function *`sof'`* that depends on some fv_i, a further pair `(`*`sof' f'`*`)` goes into *`more-pairs`*, where *`f'`* is the appropriate instance of *`sof'`*, so that the constraints on *`sof`* to be proved are properly instantiated. This further pair generates further constraints to be proved. To properly instantiate these further constraints, another pair `(`*`sof'' f''`*`)` goes into *`more-pairs`*, if *`sof''`* is a second-order function referenced by *`sof'`* that depends on some fv_i, and *`f''`* is the appropriate instance of *`sof''`*. Therefore, *`more-pairs`* includes all the pairs `(`*`sof f`*`)` such that *`sof`* is a second-order function that is directly or indirectly referenced by *`sothm`* and that depends on some fv_i, and *`f`* is the appropriate instance of *`sof`*.

If *`sof`* is a quantifier second-order function, it references a witness function sof_{w} introduced by `defun-sk`. The `defun-sk` that introduces the instance *`f`* of *`sof`* also introduces a witness function f_{w} that is effectively an instance of sof_{w}, but is not recorded in the table of instances of second-order functions because sof_{w} and f_{w} are “internal”. The pair `(`sof_{w} f_{w}`)` goes into *`more-pairs`* as well.

For each pair `(`*`sof f`*`)` in *`more-pairs`*, the constraints of *`sof`* are: the definition of *`sof`* if *`sof`* is a plain function; the constraining axiom of *`sof`* if *`sof`* is a choice function; the definition of *`sof`* and the rewrite rule of *`sof`* if *`sof`* is a quantifier function (the rewrite rule of *`sof`* is generated by `defun-sk`; its default name is *`sof`*`-necc` if the quantifier is universal, *`sof`*`-suff` if the quantifier is existential). Instantiating these constraints yields the corresponding definitions, constraining axioms, and rewrite rules of *`f`*, by the construction of the instance *`f`* of *`sof`*.

The second command of *`proof`* is

`(:repeat (:then (:use` *`facts`*`) :prove))`

where *`facts`* includes the names of all the *`f`* functions in *`more-pairs`*, which are also the names of their definitions and constraining axioms; *`facts`* also includes the names of the rewrite rules for quantifier functions. This command runs the prover on every proof subgoal, after augmenting each subgoal with all the facts in *`facts`*. This command has worked on all the examples tried so far, but a more honed approach could be investigated, should some future example fail; since the constraints are satisfied by construction, this is just an implementation issue.

For example,

```
(defthm-inst len-of-map[code-char]
  (len-of-map[?f_?p] (?f . code-char) (?p . octetp)))
```

shows that `map[code-char]` preserves the length of the list.

As another example, given instances

```
(defun-inst injective[quad[wrap]] (injective[quad[?f]] (?f . wrap)))
(defun-inst injective[wrap] (injective[?f] (?f . wrap)))
```

the theorem instance

```
(defthm-inst injective[quad[wrap]]-when-injective[wrap]
  (injective[quad[?f]]-when-injective[?f] (?f . wrap)))
```

shows that `quad[wrap]` is injective if `wrap` is.

An example instance of `fold-io[?f_?g_?io]` is in Section 2.

1.6 Summary of the Macros

`defunvar`, `defun2`, `defchoose2`, and `defun-sk2` are wrappers of existing events that explicate function variable dependencies and record additional information. They set the stage for `defun-inst` and `defthm-inst`.

`defun-inst` provides the ability to concisely generate functions, and automatically prove their termination if recursive, by specifying replacements of function variables.

`defthm-inst` provides the ability to concisely generate and automatically prove theorems, by specifying replacements of function variables.

2 Use in Program Refinement

In program refinement [9], a correct-by-construction implementation is derived from a requirements specification via a sequence of intermediate specifications. *Shallow pop-refinement* (where 'pop' stands for 'predicates over programs') is an approach to program refinement, carried out inside an interactive theorem prover by constructing a sequence of increasingly stronger predicates over one or more target functions. The sequence starts with a predicate that specifies requirements for the target functions, and ends with a predicate that provides executable definitions for the target functions. Shallow pop-refinement is a form of pop-refinement [8] in which the programs predicated upon are shallowly embedded functions of the logic of the theorem prover, instead of deeply embedded programs of a programming language as in [8].

SOFT can be used to carry out shallow pop-refinement in ACL2, as explained and exemplified below. The example derivation is overkill for the simple program obtained, which can be easily written and proved correct directly. But the purpose of the example is to illustrate techniques that can be used to derive more complex programs, and how SOFT supports these techniques (which are more directly supported in higher-order logic). The hints in some of the theorems below distill their proofs into patterns that should apply to similarly structured derivations, suggesting opportunities for future automation.

2.1 Specifications as Second-Order Predicates

Requirements over $n \geq 1$ target functions are specified by introducing function variables $\mathit{fv}_1, \ldots, \mathit{fv}_n$ that represent the target functions, and by defining a second-order predicate spec_0 over $\mathit{fv}_1, \ldots, \mathit{fv}_n$ that asserts the required properties of the target functions. The possible implementations are all the n-tuples of executable functions that satisfy the predicate. The task is to find such an n-tuple, thus constructively proving the predicate, existentially quantified over the function parameters.

For example, given a function

```
(defun leaf (e bt)
  (cond ((atom bt) (equal e bt))
        (t (or (leaf e (car bt)) (leaf e (cdr bt))))))
```

to test whether something is a leaf of a binary tree, a function to extract from a binary tree the leaves that are natural numbers, in no particular order and possibly with duplicates, can be specified as

```
(defunvar ?h (*) => *)
(defun-sk io (x y) ; input/output relation
  (forall e (iff (member e y) (and (leaf e x) (natp e))))
  :rewrite :direct)
(defun-sk2 spec[?h] (?h) ()
  (forall x (io x (?h x)))
  :rewrite :direct)
```

The task is to solve `spec[?h]` for `?h`, i.e. to find an executable function `h` such that the instance `spec[h]` of `spec[?h]` holds.

Properties implied by the requirements are proved as second-order theorems with $spec_0$ as hypothesis, e.g. for validation purposes. Since the function parameters are universally quantified in the theorem, the properties hold for all the implementations of the specification.

For example, the members of the output of every implementation of `spec[?h]` are natural numbers:

```
(defthm natp-of-member-of-output
  (implies (and (spec[?h]) (member e (?h x))) (natp e))
  :hints (("Goal" :use (spec[?h]-necc (:instance io-necc (y (?h x)))))))
```

2.2 Refinement as Second-Order Predicate Strengthening

The specification $spec_0$ is stepwise refined by constructing a sequence $spec_1, \dots, spec_m$ of increasingly stronger predicates over $fv_1, \dots, fv_n$. Each such predicate embodies a decision that either narrows down the possible implementations or rephrases their description towards their determination. The correctness of each step $j \in \{1, \dots, m\}$ is expressed by the second-order theorem `(implies (`$spec_j$`) (`$spec_{j-1}$`))`.

The sequence ends with $spec_m$ asserting that each fv_i is equal to some executable function f_i:[3]

```
(defun-sk2 def_1 (fv_1) () (forall x (equal (fv_1 x) (f_1 x))))
...
(defun-sk2 def_n (fv_n) () (forall x (equal (fv_n x) (f_n x))))
(defun2 spec_m (fv_1 ... fv_n) () (and (def_1) ... (def_n)))
```

The tuple $\langle f_1, \dots, f_n \rangle$ is the implementation. Chaining the implications of the m step correctness theorems yields the second-order theorem `(implies (`$spec_m$`) (`$spec_0$`))`. Its Σ instance, where Σ is the instantiation `((`fv_1` . `f_1`) ... (`fv_n` . `f_n`))`, is essentially Σ`((`$spec_0$`))` (because Σ`((`$spec_m$`))` is trivially true), which asserts that the implementation $\langle f_1, \dots, f_n \rangle$ satisfies $spec_0$.

More precisely, in the course of the derivation, function variables $fv_{n+1}, \dots, fv_{n+p}$ may be added to represent additional target functions $f_{n+1}, \dots, f_{n+p}$ called by $f_1, \dots, f_n$. This may happen as the task

[3] The body of each `(defun-sk2 def`$_i$` ...)` is a first-order expression of the second-order equality $fv_i = f_i$.

of finding $f_1, \ldots, f_n$ is progressively reduced to simpler sub-tasks of finding $f_{n+1}, \ldots, f_{n+p}$. If fv_{n+k} is added at refinement step j, since spec_{j-1} does not depend on fv_{n+k}, the universal quantification of fv_{n+k} over the step correctness theorem `(implies (`spec_j`) (`spec_{j-1}`))` is equivalent to an existential quantification of fv_{n+k} over the hypothesis `(`spec_j`)` of the theorem. The complete implementation that results from the derivation is $\langle f_1, \ldots, f_n, f_{n+1}, \ldots, f_{n+p} \rangle$.

The function variables fv_i are placeholders for the target functions in the spec_j predicates. Each fv_i remains uninterpreted throughout the derivation; no constraints are attached to it via axioms. Each spec_j is defined, so it does not introduce logical inconsistency. Inconsistent requirements on the target functions amount to spec_0 being always false, not to logical inconsistency. Obtaining an implementation witnesses the consistency of the requirements.

For example, `spec[?h]` from Section 2.1 can be refined as follows.

Step 1 Since the target function represented by `?h` operates on binary trees, `spec[?h]` is strengthened by constraining `?h` to be the folding function on binary trees from Section 1.2.1:

```
(defun-sk2 def-?h-fold[?f_?g] (?h ?f ?g) ()
  (forall x (equal (?h x) (fold[?f_?g] x)))
  :rewrite :direct)
(defun2 spec1[?h_?f_?g] (?h ?f ?g) ()
  (and (def-?h-fold[?f_?g]) (spec[?h])))
(defthm step1 (implies (spec1[?h_?f_?g]) (spec[?h]))
  :hints (("Goal" :in-theory '(spec1[?h_?f_?g]))))
```

The predicate `spec1[?h_?f_?g]` adds to `spec[?h]` the conjunct `def-?h-fold[?f_?g]`. Thus, the task of finding a solution for `?h` is reduced to the task of finding solutions for `?f` and `?g`: instantiating `def-?h-fold[?f_?g]` with solutions for `?f` and `?g` yields a solution for `?h`, in Step 5 below.

Step 2 The theorem `fold-io[?f_?g_?io]` from Section 1.4, which shows the correctness of the folding function (with respect to an input/output relation) under suitable correctness assumptions on the function parameters, is instantiated with the input/output relation `io` used in `spec[?h]`:

```
(defun-inst atom-io[?f] (?f) (atom-io[?f_?io] (?io . io)))
(defun-inst consp-io[?g] (?g) (consp-io[?g_?io] (?io . io)))
(defthm-inst fold-io[?f_?g] (fold-io[?f_?g_?io] (?io . io)))
```

Since the conclusion `(io x (fold[?f_?g] x))` of `fold-io[?f_?g]` equals the matrix `(io x (?h x))` of `spec[?h]` when `def-?h-fold[?f_?g]` holds, `spec1[?h_?f_?g]` is strengthened by replacing the `spec[?h]` conjunct with the hypotheses of `fold-io[?f_?g]`:

```
(defun2 spec2[?h_?f_?g] (?h ?f ?g) ()
  (and (def-?h-fold[?f_?g]) (atom-io[?f]) (consp-io[?g])))
(defthm step2 (implies (spec2[?h_?f_?g]) (spec1[?h_?f_?g]))
  :hints (("Goal" :in-theory '(spec1[?h_?f_?g] spec2[?h_?f_?g] spec[?h]
                               def-?h-fold[?f_?g]-necc fold-io[?f_?g]))))
```

Step 3 The predicate `atom-io[?f]` specifies requirements on `?f` independently from `?g` and `?h`. An implementation `f` can be derived by constructing a sequence of increasingly stronger predicates over `?f`, in the same way in which `spec[?h]` is being refined stepwise. This is a possible final result:

```
(defun f (x) (if (natp x) (list x) nil))
(defun-inst atom-io[f] (atom-io[?f] (?f . f)))
(defthm atom-io[f]! (atom-io[f]))
```

The predicate spec2[?h_?f_?g] is strengthened by replacing the atom-io[?f] conjunct with one that constrains ?f to be f:

```
(defun-sk2 def-?f (?f) () (forall x (equal (?f x) (f x))) :rewrite :direct)
(defun2 spec3[?h_?f_?g] (?h ?f ?g) ()
  (and (def-?h-fold[?f_?g]) (def-?f) (consp-io[?g])))
(defthm step3-lemma (implies (def-?f) (atom-io[?f]))
  :hints (("Goal" :in-theory '(atom-io[?f] atom-io[f]-necc
                               atom-io[f]! def-?f-necc))))
(defthm step3 (implies (spec3[?h_?f_?g]) (spec2[?h_?f_?g]))
  :hints (("Goal" :in-theory '(spec2[?h_?f_?g] spec3[?h_?f_?g] step3-lemma))))
```

Step 4 The predicate consp-io[?g] specifies requirements on ?g independently from ?f and ?h. An implementation g can be derived by constructing a sequence of increasingly stronger predicates over ?g, in the same way in which spec[?h] is being refined stepwise. This is a possible final result:

```
(defun g (y1 y2) (append y1 y2))
(defun-inst consp-io[g] (consp-io[?g] (?g . g)))
(defthm member-of-append ; used to prove CONSP-IO[G]-LEMMA below
  (iff (member e (append y1 y2)) (or (member e y1) (member e y2))))
(defthm consp-io[g]-lemma ; used to prove CONSP-IO[G]! below
  (implies (and (consp x) (io (car x) y1) (io (cdr x) y2))
           (io x (g y1 y2)))
  :hints (("Goal" :in-theory (disable io) :expand (io x (append y1 y2)))))
(defthm consp-io[g]! (consp-io[g]) :hints (("Goal" :in-theory (disable g))))
```

The predicate spec3[?h_?f_?g] is strengthened by replacing the consp-io[?f] conjunct with one that constrains ?g to be g:

```
(defun-sk2 def-?g (?g) ()
  (forall (y1 y2) (equal (?g y1 y2) (g y1 y2)))
  :rewrite :direct)
(defun2 spec4[?h_?f_?g] (?h ?f ?g) ()
  (and (def-?h-fold[?f_?g]) (def-?f) (def-?g)))
(defthm step4-lemma (implies (def-?g) (consp-io[?g]))
  :hints (("Goal" :in-theory '(consp-io[?g] consp-io[g]-necc
                               consp-io[g]! def-?g-necc))))
(defthm step4 (implies (spec4[?h_?f_?g]) (spec3[?h_?f_?g]))
  :hints (("Goal" :in-theory '(spec3[?h_?f_?g] spec4[?h_?f_?g] step4-lemma))))
```

Step 5 Substituting the solutions f and g into fold[?f_?g] yields a solution for ?h:

```
(defun-inst h (fold[?f_?g] (?f . f) (?g . g)))
(defun-sk2 def-?h (?h) () (forall x (equal (?h x) (h x))) :rewrite :direct)
```

The conjunct def-?h-fold[?f_?g] of spec4[?h_?f_?g] is replaced with def-?h, which is equivalent to def-?h-fold[?f_?g] given the conjuncts def-?f and def-?g:

```
(defun2 spec5[?h_?f_?g] (?h ?f ?g) () (and (def-?h) (def-?f) (def-?g)))
(defthm step5-lemma
  (implies (and (def-?f) (def-?g)) (equal (h x) (fold[?f_?g] x)))
  :hints (("Goal" :in-theory '(h fold[?f_?g] def-?f-necc def-?g-necc))))
(defthm step5 (implies (spec5[?h_?f_?g]) (spec4[?h_?f_?g]))
  :hints (("Goal" :in-theory '(spec4[?h_?f_?g] spec5[?h_?f_?g]
                               def-?h-fold[?f_?g] def-?h-necc step5-lemma))))
```

This concludes the derivation: `spec[?h_?f_?g]` provides executable solutions for `?h`, `?f`, and `?g`. The resulting implementation is $\langle \mathtt{h}, \mathtt{f}, \mathtt{g} \rangle$. Chaining the implications of the step correctness theorems shows that these solutions satisfy the requirements specification:

```
(defthm chain[?h_?f_?g] (implies (spec5[?h_?f_?g]) (spec[?h]))
  :hints (("Goal" :in-theory '(step1 step2 step3 step4 step5))))
```

More explicitly, instantiating the end-to-end implication shows that `h` satisfies the requirements specification:

```
(defun-inst def-h (def-?h (?h . h)))
(defun-inst def-f (def-?f (?f . f)))
(defun-inst def-g (def-?g (?g . g)))
(defun-inst spec5[h_f_g] (spec5[?h_?f_?g] (?h . h) (?f . f) (?g . g)))
(defun-inst spec[h] (spec[?h] (?h . h)))
(defthm-inst chain[h_f_g] (chain[?h_?f_?g] (?h . h) (?f . f) (?g . g)))
(defthm spec5[h_f_g]! (spec5[h_f_g])
  :hints (("Goal" :in-theory '(spec5[h_f_g]))))
(defthm spec[h]! (spec[h])
  :hints (("Goal" :in-theory '(chain[h_f_g] spec5[h_f_g]!))))
```

3 Related Work

The `instance-of-defspec` tool [14] and the `make-generic-theory` tool [17] automatically generate instances of functions and theorems that reference functions constrained via encapsulation [15], by replacing the constrained functions with functions that satisfy the constraints. The instantiation mechanisms of these tools are similar to the ones of SOFT; constrained functions in these tools parallel function variables in SOFT. However, in SOFT function variables are unconstrained; constraints on them are expressed via second-order predicates (typically with quantifiers), and the same function variables can be used as parameters of different constraining predicates. Unlike SOFT, `instance-of-defspec` and `make-generic-theory` do not handle choice and quantifier functions, and do not generate termination proofs for recursive function instances. SOFT generates one function or theorem instance at a time, while `instance-of-defspec` and `make-generic-theory` can generate many. These two tools are more suited to developing and instantiating abstract and parameterized theories; SOFT is more suited to mimic second-order logic notation.

The `:consider` hint [19] heuristically generates functional instantiations to help prove given theorems. SOFT generates function and theorem instances for given replacements of function variables; from these replacements, the necessary functional instantiations are generated automatically.

The `def-functional-instance` tool in the ACL2 community books generates theorem instances for given replacements of functions. This tool has more general use than SOFT's `defthm-inst`, but it requires a complete functional instantiation, while defthm-inst only requires replacements for function variables.

Wrapping existing events to record information for later use (as done by SOFT's defunvar, defun2, defchoose2, and defun-sk2) has precedents. For example, the def:un-sk tool [11] is a wrapper of defun-sk that records information to help prove theorems involving quantifiers. It may be useful to combine def:un-sk with SOFT's defun-sk2 wrapper.

There are several tools to generate functions and theorems according to certain patterns, such as std::deflist in the ACL2 standard library and fty::deflist in the FTY library [23]. These tools may use SOFT to generate some of the functions and theorems as instances of pre-defined second-order functions and theorems.

A general-purpose theorem prover like ACL2 can represent a variety of specification and refinement formalisms, e.g. [1, 2, 12, 13, 18, 20, 22]; derivations can be carried out within the logic. But given the close ties to Applicative Common Lisp, a natural approach to program refinement in ACL2 is to specify requirements on one or more target ACL2 functions, and progressively strengthen the requirements until the functions are executable and performant.

Alternatives to SOFT's second-order predicates, for specifying requirements on ACL2 functions, include encapsulate (possibly via the wrappers defspec and defabstraction in the ACL2 community books), defaxiom, and defchoose. But these are not as suited to program refinement:

- An encapsulate involves exhibiting witnesses to the consistency of the requirements, which amounts to writing an implementation and proving it correct. But it is the purpose of program refinement to construct an implementation and its correctness proof.
- A defaxiom obviates witnesses but may introduce logical inconsistency.
- A defchoose obviates witnesses and is logically conservative, but:
  - It expresses requirements on single functions, necessitating the combination of multiple target functions into one.
  - It expresses requirements on function results (the bound variables) with respect to function arguments (the free variables), but not requirements involving different results and different arguments, such as injectivity, non-interference [10], and other hyperproperties [7].
  - It prescribes underspecified but fixed function results. For example, there is no clear refinement relation between the function introduced as (defchoose f (y) (x) (> y x)) and the function introduced as (defun g (x) (+ x 1)).

In contrast, a second-order predicate can specify any kind of requirements, on multiple functions, maintaining logical consistency, and doing so without premature witnesses.

In the derivation in Section 2.2, the use and instantiation of the generic folding function on binary trees is an example of the application of algorithm schemas in program refinement, as in [21] but here realized via second-order functions and theorems. Second-order functions express algorithm schemas, and second-order theorems show their correctness under suitable conditions on the function parameters. Applying a schema adds a constraint that defines a target function to use the schema, and introduces simpler target functions corresponding to the function parameters, constrained to satisfy the conditions for the correctness of the schema.

A refinement step from a specification spec_j can be performed manually, by writing down spec_{j+1} and proving (implies (spec_{j+1}) (spec_j)). It is sometimes possible to generate spec_{j+1} from spec_j, along with a proof of (implies (spec_{j+1}) (spec_j)), using automated transformation techniques. Automated transformations may require parameters to be provided and applicability conditions to be proved, but should generally save effort and make derivations more robust against changes in requirements specifications. At Kestrel Institute, we are developing ACL2 libraries of automated transformations for program refinement.

4 Future Work

Guards `defun-inst` could be extended with the option to override the default guard $\Sigma(\mathit{guard})$ with a different *guard'*, generating the proof obligation `(implies` *guard'* $\Sigma(\mathit{guard})$`)`. This would be useful in at least two situations.

A first situation is when the function instance has more guard conditions to verify than the second-order function being instantiated, due to the replacement of a function parameter (which has no guards) with a function that has guards. Providing a stronger guard to the function instance would enable the verification of the additional guard conditions. For example, an instance `quad[cdr]` of `quad[?f]` from Section 1.2.1 could be supplied with the guard `(true-listp x)`.

A second situation is when the guard of the second-order function being instantiated includes conditions on function parameters that involve a quantifier, e.g. the condition that the binary operation `?op` of a generic folding function over lists is closed over the type `?p` of the list elements. Instantiating `?p` with `natp` and `?op` with `binary-+` satisfies the condition, but $\Sigma(\mathit{guard})$ still includes a quantifier that makes the instance of the folding function non-executable. Supplying a *guard'* that rephrases $\Sigma(\mathit{guard})$ to omit the satisfied closure condition would solve the problem. As guard obligations on individual parameters are relieved when functions are applied to terms in a term, it makes sense to relieve guard obligations on function parameters when second-order functions are “applied” to functions in `defun-inst`.

`defun-inst` could also be extended with the ability to use the instances of the verified guard conditions of the second-order function being instantiated, to help verify the guard conditions of the function instance. This may completely verify the guards of the instance, when no guard overriding is needed.

Partial Functions SOFT could be extended with a macro `defpun2` to introduce partial second-order functions, mimicked by partial first-order functions introduced via `defpun` [16]. `defun-inst` could be extended to generate not only partial function instances, but also total function instances when the instantiated `:domain` or `:gdomain` restrictions are theorems. Partial second-order functions would be useful, in particular, to define recursive algorithm schemas whose measures and whose argument updates in recursive calls are, or depend on, function parameters. An example is a general divide-and-conquer schema.[4]

Mutual Recursion SOFT could be extended with a macro `mutual-recursion2` to introduce mutually recursive plain second-order functions (with `defun2`), mimicked by mutually recursive first-order functions introduced via `mutual-recursion`. `defun-inst` could be extended to generate instances of mutually recursive second-order functions.

Lambda Expressions `defun-inst` and `defthm-inst` could be extended to accept instantiations that map function variables to lambda expressions, similarly to `:functional-instance`.

[4] The folding function from Section 1.2.1 is a divide-and-conquer schema specialized to binary trees.

Instantiation Transitivity If *sof'* is introduced as the Σ instance of *sof*, and *f* is introduced as the Σ’ instance of *sof'*, then *f* should be the Σ” instance of *sof*, where Σ” is a suitably defined composition of Σ and Σ’. Currently `defun-inst` does not record *f* as an instance of *sof* when *f* is introduced, but it could be extended to do so. With this extension, `injective[quad[wrap]]` in Section 1.5 would be the `((?f . quad[wrap]))` instance of `injective[?f]` in Section 1.2.3.

In a related but different situation, given *sof*, *sof'*, *f*, Σ, Σ’, and Σ” as above, but with *f* introduced as the Σ” instance of *sof*, and *sof'* introduced as the Σ instance of *sof*, in either order (i.e. *f* then *sof'*, or *sof'* then *f*), then *f* should be the Σ’ instance of *sof'*. Currently `defun-inst` does not record *f* as an instance of *sof'* when *f* (after *sof'*) or *sof'* (after *f*) is introduced, but could be extended to do so. With this extension, if `injective[quad[wrap]]` were introduced as the `((?f . quad[wrap]))` instance of `injective[?f]`, and `injective[quad[?f]]` were introduced as the `((?f . quad[?f]))` instance of `injective[?f]` as in Section 1.3, then `injective[quad[wrap]]` would be the `((?f . wrap))` instance of `injective[quad[?f]]`.

An alternative to these two extensions of `defun-inst` is to extend SOFT with a macro to claim that an existing instance of a second-order function is also an instance of another second-order function according to a given instantiation. The macro would check the claim (by applying the instantiation and comparing the result with the function) and extend the table of instances of second-order functions if the check succeeds. In the first scenario above, the macro would be used to claim that *f* is the Σ” instance of *sof*; in the second scenario above, the macro would be used to claim that *f* is the Σ’ instance of *sof'*.

Function Variable Constraints Currently the only constraints on function variables are their types. `defunvar` could be extended to accept richer signatures for function variables, with multiple-value results and single-threaded arguments and results. `defun-inst` and `defthm-inst` would then be extended to check that instantiations satisfy these additional constraints. A more radical extension would be to attach logical constraints to certain function variables, as in encapsulations.

Automatic Instances As explained in Section 1.3, when an instantiation is applied to a term, the table of instances of second-order functions is consulted to find replacements for certain second-order functions, and the application of the instantiation fails if replacements are not found. Thus, all the needed instances must be introduced before applying the instantiation, e.g. in Section 1.5 the two `defun-insts` had to be supplied before the last `defthm-inst`. SOFT could be extended to generate automatically the needed instances of second-order functions.

SOFT could also be extended with a macro `defthm2` to prove a second-order theorem via `defthm` and to record the theorem in a table, along with information about the involved second-order functions. `defun-inst` could be extended with the option to generate instances of the second-order theorems that involve the second-order function being instantiated. `defthm2` could include the option to generate instances of the theorem that correspond to the known instances of the second-order functions that the theorem involves. These extensions would reduce the use of explicit `defthm-insts`.

The convention of including function variables in square brackets in the names of second-order functions and theorems, could be exploited to name the automatically generated function and theorem instances, as suggested by the examples throughout the paper.

Other Events SOFT could be extended to provide second-order counterparts of other function and theorem introduction events, e.g. `define`, `defines`, and `defrule` in the ACL2 community books.

References

[1] Martín Abadi & Leslie Lamport (1991): *The Existence of Refinement Mappings*. Journal of Theoretical Computer Science 82(2), doi:10.1016/0304-3975(91)90224-P.

[2] Jean-Raymond Abrial (1996): *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, doi:10.1017/CBO9780511624162.

[3] *The ACL2 Theorem Prover*. `http://www.cs.utexas.edu/~moore/acl2`.

[4] Peter B. Andrews (2002): *An Introduction to Mathematical Logic and Type Theory: To Thruth Through Proof*. Springer, doi:10.1007/978-94-015-9934-4.

[5] R. S. Boyer, D. M. Goldschlag, M. Kaufmann & J S. Moore (1991): *Functional Instantiation in First Order Logic*. Technical Report 44, Computational Logic Inc.

[6] Alonzo Church (1940): *A Formulation of the Simple Theory of Types*. The Journal of Symbolic Logic 5(2), doi:10.2307/2266170.

[7] Michael Clarkson & Fred Schneider (2010): *Hyperproperties*. Journal of Computer Security 18(6), doi:10.3233/JCS-2009-0393.

[8] Alessandro Coglio (2014): *Pop-Refinement*. Archive of Formal Proofs. `http://afp.sf.net/entries/Pop_Refinement.shtml`, Formal proof development.

[9] Edsger W. Dijkstra (1968): *A Constructive Approach to the Problem of Program Correctness*. BIT 8(3), doi:10.1007/BF01933419.

[10] Joseph Goguen & José Meseguer (1982): *Security Policies and Security Models*. In: Proc. IEEE Symposium on Security and Privacy, doi:10.1109/SP.1982.10014.

[11] David Greve (2009): *Automated Reasoning with Quantified Formulae*. In: Proc. 8th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2009), doi:10.1145/1637837.1637855.

[12] C. A. R. Hoare (1972): *Proof of Correctness of Data Representations*. Acta Informatica 1(4), doi:10.1007/BF00289507.

[13] Cliff Jones (1990): *Systematic Software Development using VDM*, second edition. Prentice Hall.

[14] Sebastiaan J. C. Joosten, Bernard van Gastel & Julien Schmaltz (2013): *A Macro for Reusing Abstract Functions and Theorems*. In: Proc. 11th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2013), doi:10.4204/EPTCS.114.3.

[15] Matt Kaufmann & J Strother Moore (2001): *Structured Theory Development for a Mechanized Logic*, doi:10.1023/A:1026517200045.

[16] Panagiotis Manolios & J Strother Moore (2003): *Partial Functions in ACL2*, doi:10.1023/B:JARS.0000009505.07087.34.

[17] F. J. Martìn-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: Proc. 3rd International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2002).

[18] Robin Milner (1971): *An Algebraic Definition of Simulation between Programs*. Technical Report CS-205, Stanford University.

[19] J Strother Moore (2009): *Automatically Computing Functional Instantiations*. In: Proc. 8th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2009), doi:10.1145/1637837.1637839.

[20] Carroll Morgan (1998): *Programming from Specifications*, second edition. Prentice Hall.

[21] Douglas R. Smith (1999): *Mechanizing the Development of Software*. In Manfred Broy, editor: Calculational System Design, Proc. Marktoberdorf Summer School, IOS Press.

[22] J. M. Spivey (1992): *The Z Notation: A Reference Manual*, second edition. Prentice Hall.

[23] Sol Swords & Jared Davis (2015): *Fix Your Types*. In: Proc. 13th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2015).