Second-Order Functions and Theorems in ACL2

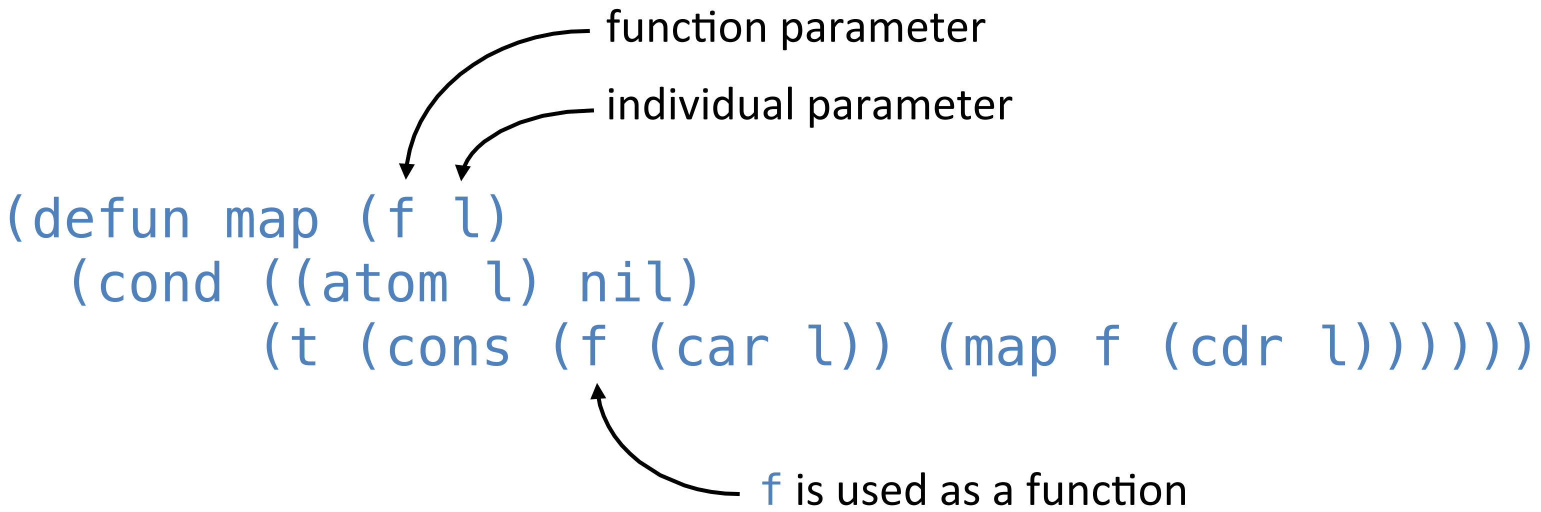
Alessandro Coglio

Kestrel
Institute

ACL2 *Workshop 2015*

If ACL2 were second-order…

… we could define second-order functions:



function parameter

individual parameter

```
(defun map (f l)
  (cond ((atom l) nil)
        (t (cons (f (car l)) (map f (cdr l))))))
```

`f` is used as a function

If ACL2 were second-order…

```
(defun map (f l)
  (cond ((atom l) nil)
        (t (cons (f (car l)) (map f (cdr l))))))
```

… we could prove second-order theorems:

universally quantified over `f` and `l`

```
(defthm len–of–map
  (equal (len (map f l)) (len l)))
```

If ACL2 were second-order…

```
(defun map (f l)
  (cond ((atom l) nil)
        (t (cons (f (car l)) (map f (cdr l))))))

(defthm len-of-map
  (equal (len (map f l)) (len l)))
```

… we could apply second-order functions:

```
(defun rev-fix-cons (a x)
  (cons a (map fix (rev x))))
```

`map` is applied to the function `fix`

If ACL2 were second-order…

```
(defun map (f l)
  (cond ((atom l) nil)
        (t (cons (f (car l)) (map f (cdr l))))))

(defthm len-of-map
  (equal (len (map f l)) (len l)))

(defun rev-fix-cons (a x)
  (cons a (map fix (rev x))))
```

… we could use second-order theorems:

proved using `(:rewrite len-of-map)`

```
(defthm len-of-rev-fix-cons
  (equal (len (rev-fix-cons a x)) (1+ (len x))))
```

With the SOFT (‘Second-Order Functions and Theorems’) tool…

… we can define second-order functions:

we must declare function variables

```
(defunvar f (*) => *)
```

we must use `defun2` (2nd-order version of `defun`)

we must separate function and individual parameters

```
(defun2 map (f) (l)
  (cond ((atom l) nil)
        (t (cons (f (car l)) (map (cdr l))))))
```

we must omit
function parameters
in 2nd-order function calls

With the SOFT (‘Second-Order Functions and Theorems’) tool…

… we can define second-order functions:

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

```
(defun2 map[?f] (?f) ()
  (cond ((atom l) nil)
        (t (cons (?f (car l)) (map[?f] (cdr l))))))
```

“restores” omitted function parameter

possible naming conventions (not enforced by SOFT):

- start function variable names with ?
- include function parameters in names of 2nd-order functions

With the SOFT (‘Second-Order Functions and Theorems’) tool…

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

(defun2 map[?f] (?f) (l)
  (cond ((atom l) nil)
        (t (cons (?f (car l)) (map[?f] (cdr l))))))
```

… we can prove second-order theorems:

a regular `defthm`, whose formula references `?f`

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

universally quantified over `?f` and `l`

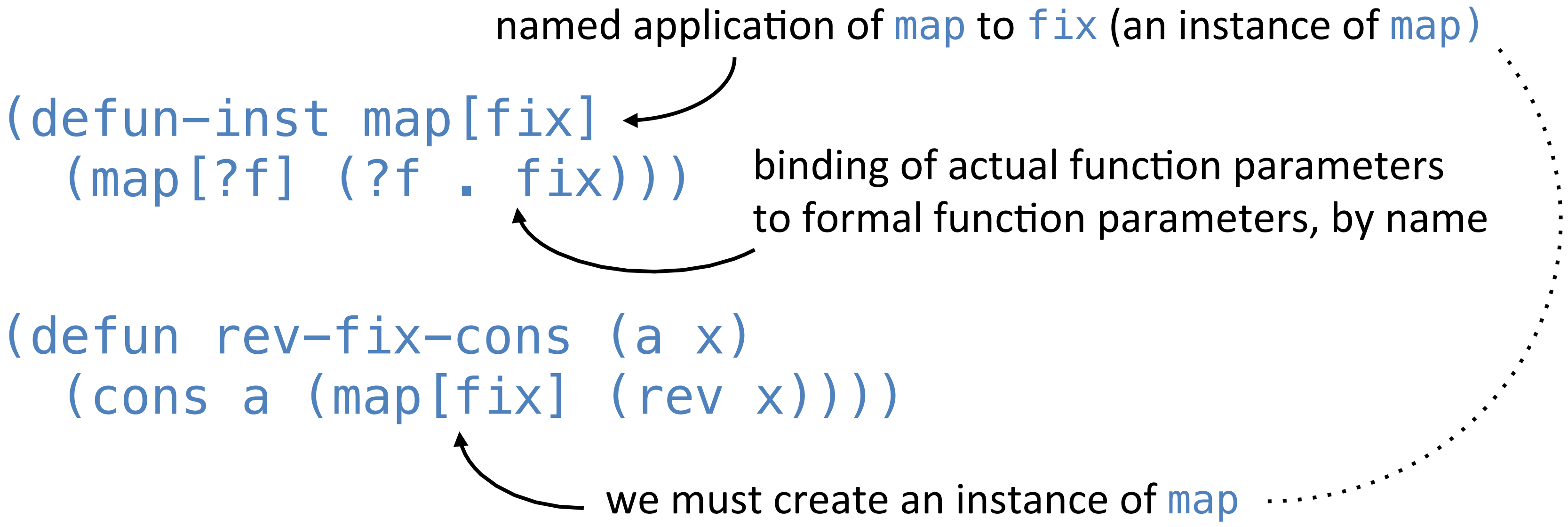
With the SOFT (‘Second-Order Functions and Theorems’) tool…

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

(defun2 map[?f] (?f) (l)
  (cond ((atom l) nil)
        (t (cons (?f (car l)) (map[?f] (cdr l))))))

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

… we can apply second-order functions:



named application of `map` to `fix` (an instance of `map`)

```
(defun-inst map[fix]
  (map[?f] (?f . fix)))
```

binding of actual function parameters to formal function parameters, by name

```
(defun rev-fix-cons (a x)
  (cons a (map[fix] (rev x))))
```

we must create an instance of `map`

With the SOFT (‘Second-Order Functions and Theorems’) tool…

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

(defun2 map[?f] (?f) (l)
  (cond ((atom l) nil)
        (t (cons (?f (car l)) (map[?f] (cdr l))))))

(defthm len-of-map[?f]
  (equal (len (map[?f] l)) (len l)))

(defun-inst map[fix]
  (map[?f] (?f . fix)))

(defun rev-fix-cons (a x)
  (cons a (map[fix] (rev x))))
```

… we can use second-order theorems:

named instance of `len-of-map[?f]`

```
(defthm-inst len-of-map[fix]
  (len-of-map[?f] (?f . fix)))
```

function variable substitution

proved using `(:rewrite len-of-map[fix])`

```
(defthm len-of-rev-fix-cons
  (equal (len (rev-fix-cons a x)) (1+ (len x))))
```

With the SOFT (‘Second-Order Functions and Theorems’) tool…

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

(defun2 map[?f] (?f) (l)
  (cond ((atom l) nil)
        (t (cons (?f (car l)) (map[?f] (cdr l))))))

(defthm len-of-map[?f]
  (equal (len (map[?f] l)) (len l)))

(defun-inst map[fix]
  (map[?f] (?f . fix)))

(defun rev-fix-cons (a x)
  (cons a (map[fix] (rev x))))

(defthm-inst len-of-map[fix]
  (len-of-map[?f] (?f . fix)))

(defthm len-of-rev-fix-cons
  (equal (len (rev-fix-cons a x)) (1+ (len x))))
```

How does this work?

SOFT (‘Second-Order Functions and Theorems’) is an ACL2 library to mimic second-order functions and theorems in the first-order logic of ACL2.

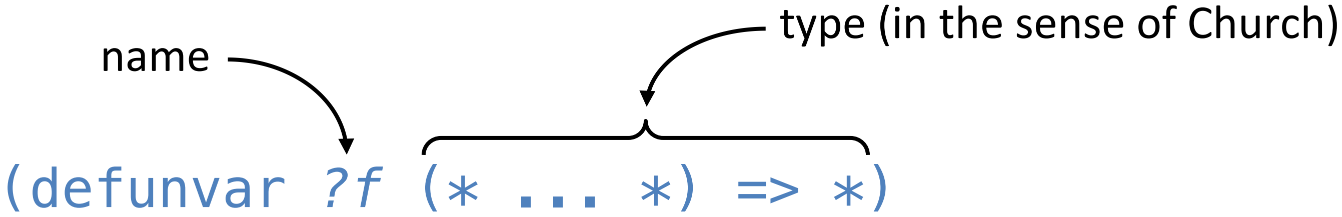
It does not extend the ACL2 logic.

It does not introduce unsoundness or inconsistency on its own.

It provides the following macros:

```
defunvar
defun2
defchoose2
defun-sk2
defun-inst
defthm-inst
```

Macro to introduce function variables:

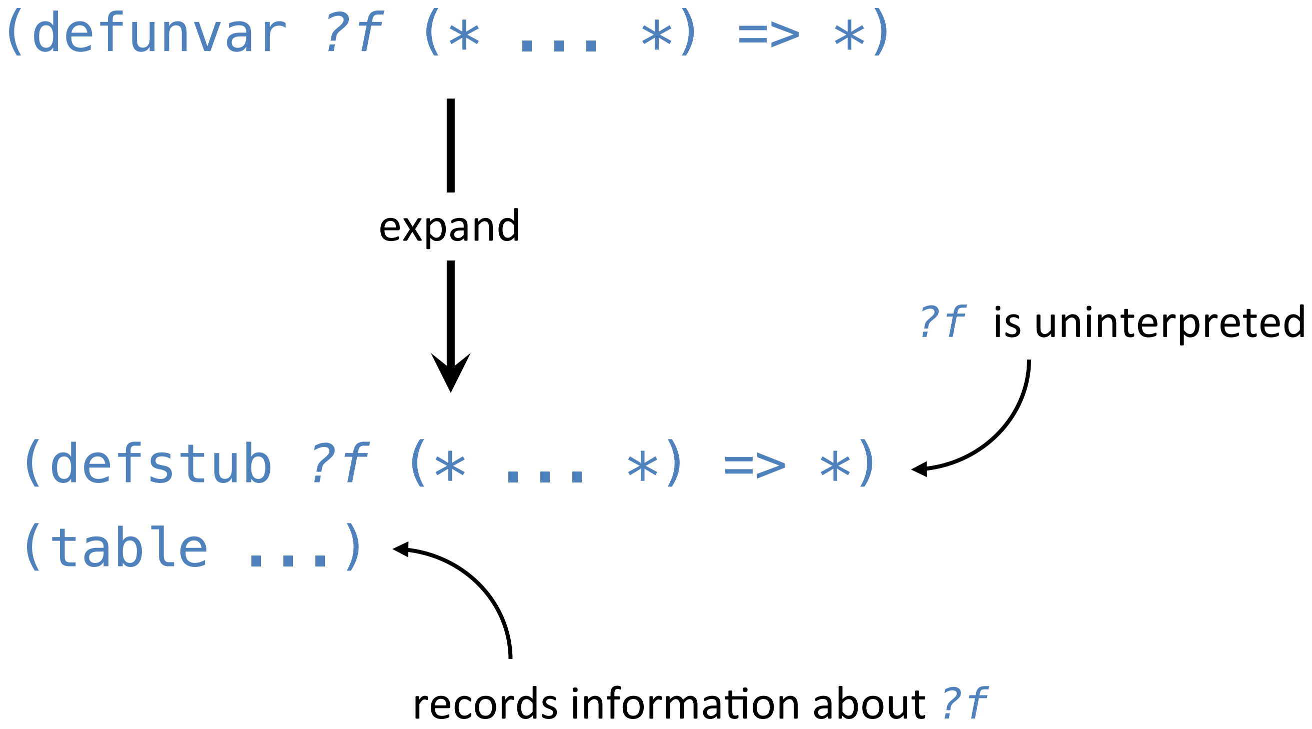


name

type (in the sense of Church)

```
(defunvar ?f (* ... *) => *)
```

Macro to introduce function variables:



```
(defunvar ?f (* ... *) => *)
```

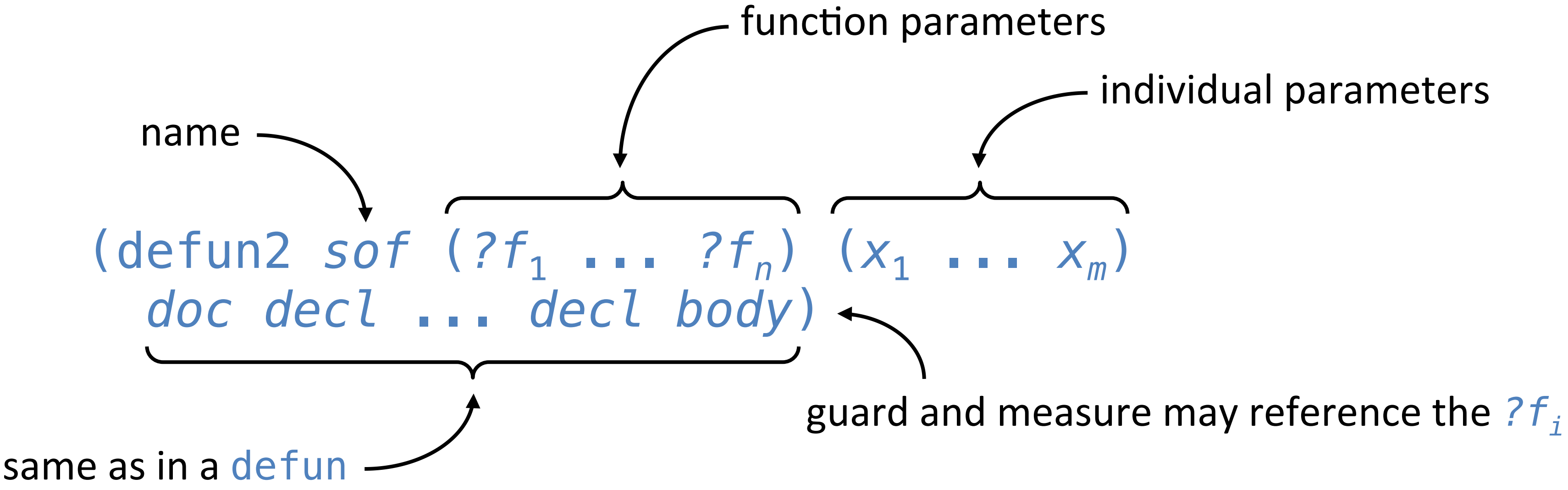
expand

```
(defstub ?f (* ... *) => *)
(table ...)
```

`?f` is uninterpreted

records information about `?f`

Macro to introduce plain second-order functions:



function parameters

individual parameters

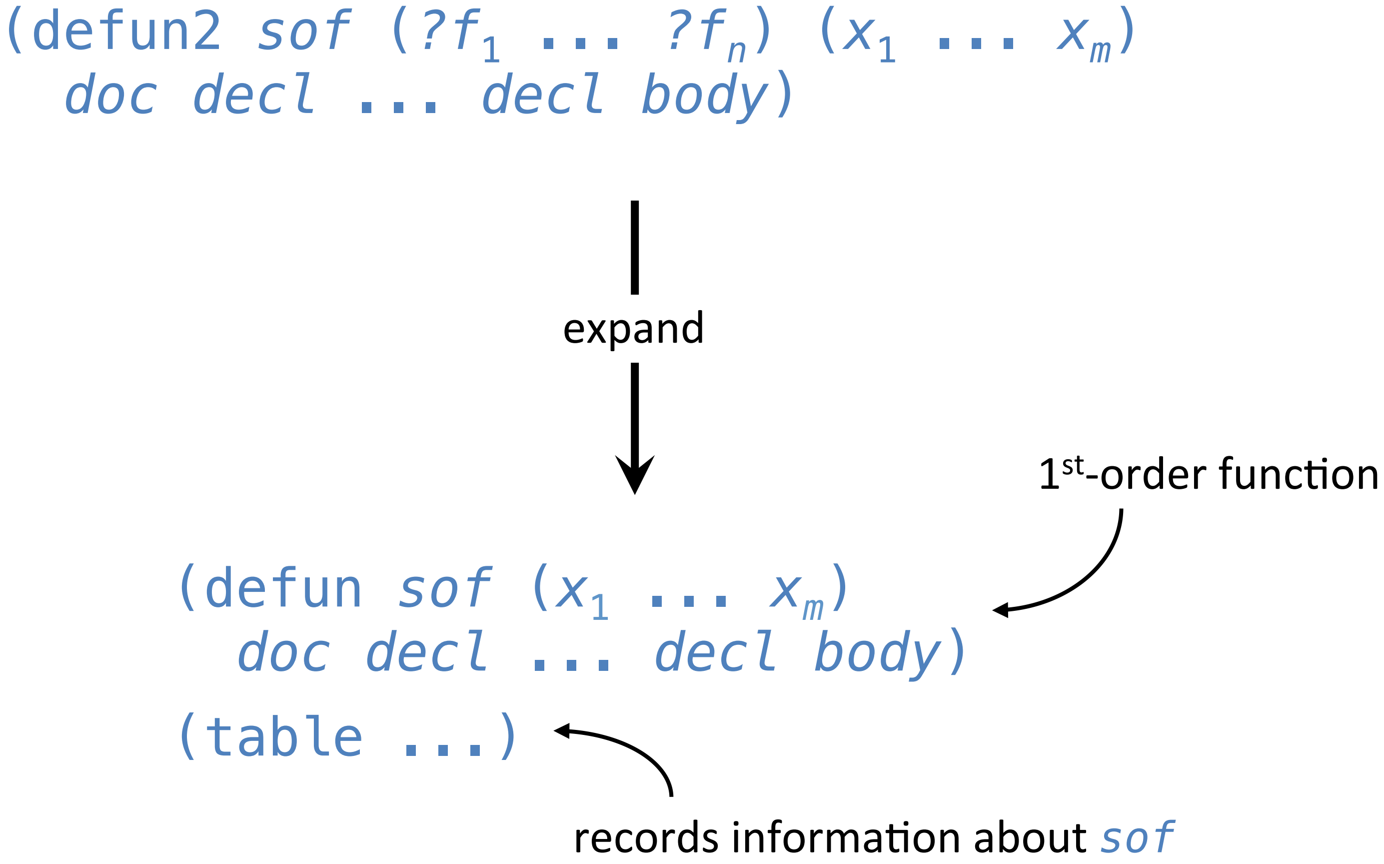
name

(defun2 *sof* ($?f_1$ ... $?f_n$) (x_1 ... x_m)
  doc decl ... *decl body*)

guard and measure may reference the $?f_i$

same as in a `defun`

Macro to introduce plain second-order functions:



```
(defun2 sof (?f1 ... ?fn) (x1 ... xm)
  doc decl ... decl body)
```

expand

1st-order function

```
(defun sof (x1 ... xm)
  doc decl ... decl body)
(table ...)
```

records information about *sof*

SOFT (‘Second-Order Functions and Theorems’) is an ACL2 library to mimic second-order functions and theorems in the first-order logic of ACL2.

It does not extend the ACL2 logic.

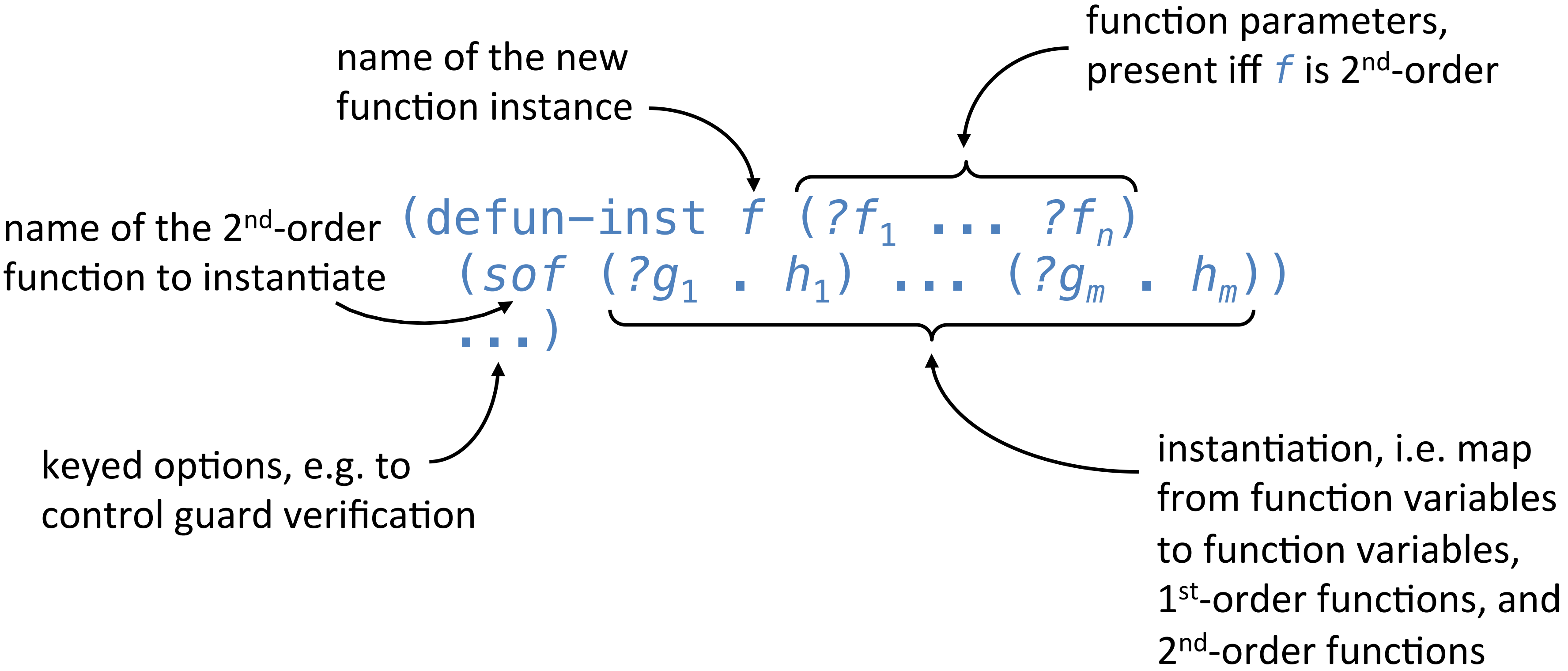
It does not introduce unsoundness or inconsistency on its own.

It provides the following macros:

- `defunvar`
- `defun2`
- `defchoose2`
- `defun-sk2`
- `defun-inst`
- `defthm-inst`

introduce choice 2nd-order functions and quantifier 2nd-order functions (2nd-order versions of `defchoose` and `defun-sk`; analogous to `defun2`)

Macro to instantiate second-order functions:



name of the new function instance

function parameters, present iff f is 2nd-order

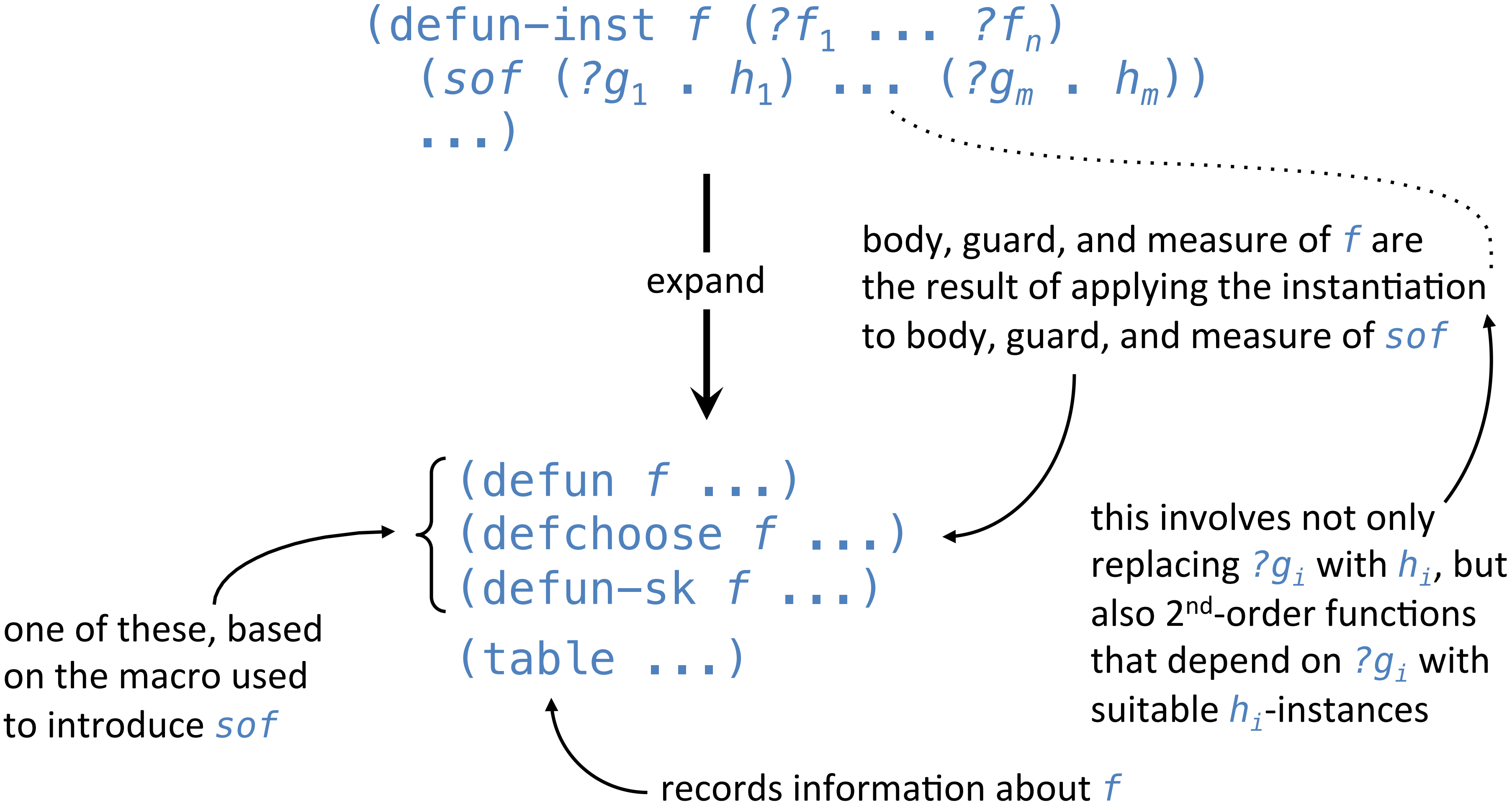
name of the 2nd-order function to instantiate

$$
\begin{array}{l}
\texttt{(defun-inst } f\ \overbrace{\texttt{(?}f_1\ \ldots\ \texttt{?}f_n\texttt{)}}\\
\quad \texttt{(sof } \underbrace{\texttt{(?}g_1\ \texttt{.}\ h_1\texttt{)}\ \ldots\ \texttt{(?}g_m\ \texttt{.}\ h_m\texttt{)}}\texttt{)}\\
\quad \texttt{...)}
\end{array}
$$

keyed options, e.g. to control guard verification

instantiation, i.e. map from function variables to function variables, 1st-order functions, and 2nd-order functions

Macro to instantiate second-order functions:



$\texttt{(defun-inst } f \texttt{ (?}f_1 \texttt{ ... ?}f_n\texttt{)}$
$\quad \texttt{(}sof\texttt{ ((?}g_1 \texttt{ . } h_1\texttt{) ... (?}g_m \texttt{ . } h_m\texttt{))}$
$\quad \texttt{...)}$

expand

$\texttt{(defun } f \texttt{ ...)}$
$\texttt{(defchoose } f \texttt{ ...)}$
$\texttt{(defun-sk } f \texttt{ ...)}$

$\texttt{(table ...)}$

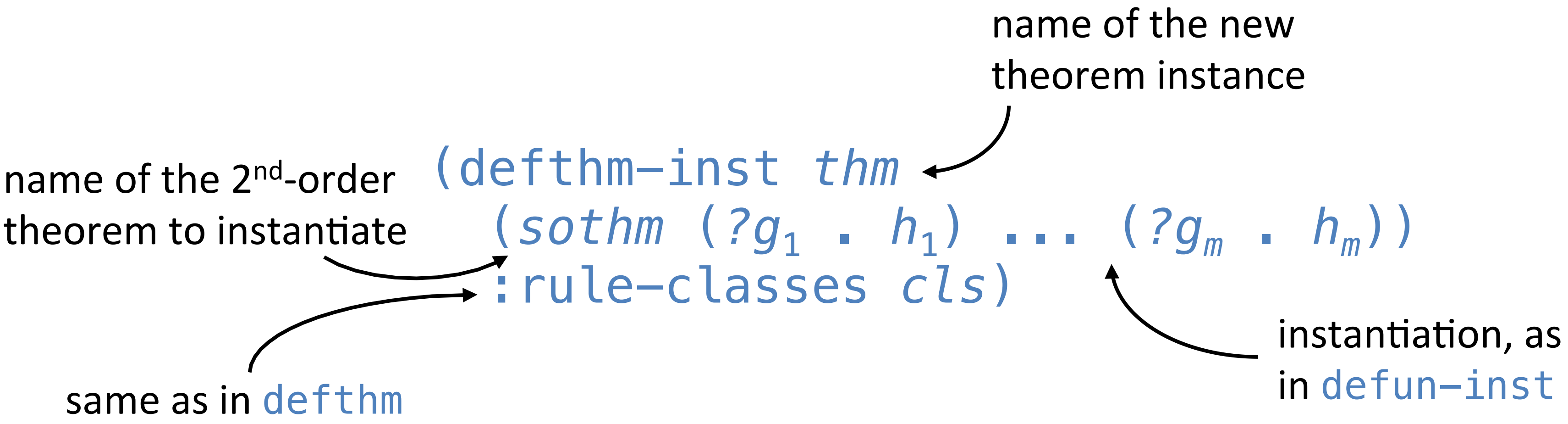
body, guard, and measure of f are the result of applying the instantiation to body, guard, and measure of sof

this involves not only replacing $?g_i$ with h_i, but also 2nd-order functions that depend on $?g_i$ with suitable h_i-instances

one of these, based on the macro used to introduce sof

records information about f

Macro to instantiate second-order theorems:



name of the new theorem instance

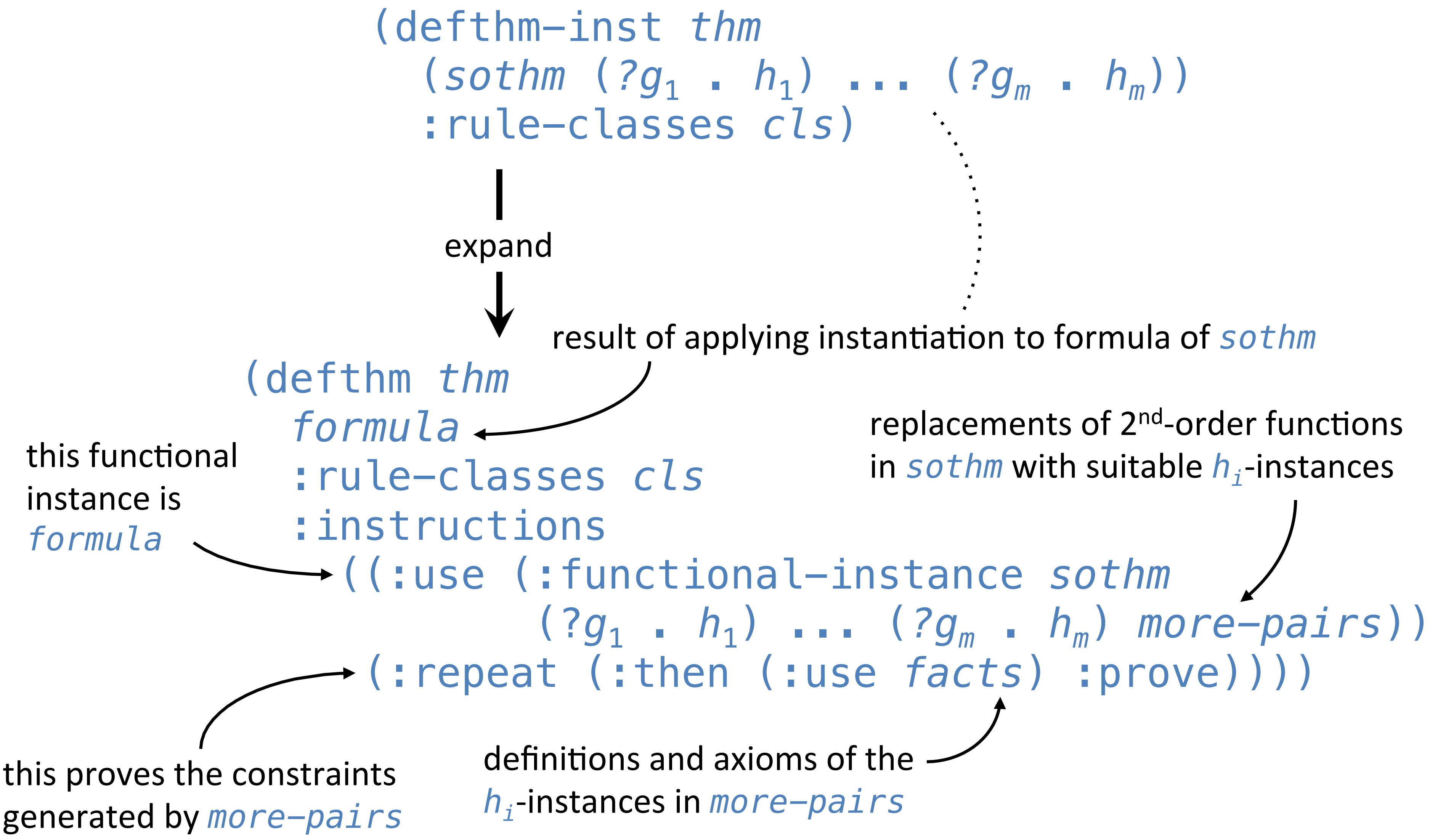
```
(defthm-inst thm
  (sothm (?g1 . h1) ... (?gm . hm))
  :rule-classes cls)
```

name of the 2nd-order theorem to instantiate

same as in `defthm`

instantiation, as in `defun-inst`

Macro to instantiate second-order theorems:



```
(defthm-inst thm
  (sothm (?g_1 . h_1) ... (?g_m . h_m))
  :rule-classes cls)
```

expand

result of applying instantiation to formula of *sothm*

```
(defthm thm
  formula
  :rule-classes cls
  :instructions
  ((:use (:functional-instance sothm
                (?g_1 . h_1) ... (?g_m . h_m) more-pairs))
   (:repeat (:then (:use facts) :prove))))
```

this functional instance is *formula*

replacements of 2nd-order functions in *sothm* with suitable h_i-instances

this proves the constraints generated by *more-pairs*

definitions and axioms of the h_i-instances in *more-pairs*

SOFT (‘Second-Order Functions and Theorems’) is an ACL2 library to mimic second-order functions and theorems in the first-order logic of ACL2.

It does not extend the ACL2 logic.

It does not introduce unsoundness or inconsistency on its own.

It provides the following macros:

- `defunvar`
- `defun2`
- `defchoose2`
- `defun-sk2`
- `defun-inst`
- `defthm-inst`

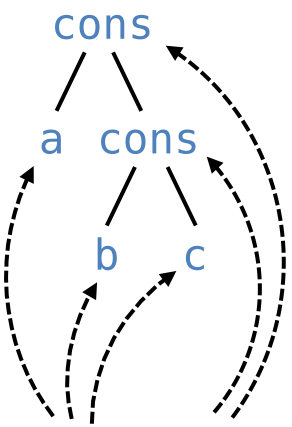
SOFT can be used to formalize algebras and similar mathematical structures in a compositional way, e.g.:

```
(defun-sk2 semigroup[?op] (?op) ()
  (forall (x y z)
          (equal (?op (?op x y) z) (?op x (?op y z)))))
(defun-sk2 identity[?op] (?op) (id)
  (forall x (and (equal (?op id x) x)
                 (equal (?op x id) x))))
(defun2 monoid[?op] (?op) (id)
  (and (semigroup[?op]) (identity[?op] id)))
(defun-sk2 inverse[?op_?inv] (?op ?inv) (id)
  (forall x (and (equal (?op x (?inv x)) id)
                 (equal (?op (?inv x) x) id))))
(defun2 group[?op_?inv] (?op ?inv) (id)
  (and (monoid[?op] id) (inverse[?op_?inv] id)))
```

Unlike `encapsulate`, algebraic properties are expressed by predicates, not by axioms attached to the abstract operations.

SOFT can be used to formalize algorithm schemas, e.g.:

generic folding function
on binary trees



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

divide-and-conquer algorithm schema, specialized to binary trees: to solve a problem on a binary tree, recursively solve the problem on its subtrees and combine the solutions using `?g`; solve the problem on leaves directly using `?f`

SOFT can be used to formalize algorithm schemas, e.g.:

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

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

predicate asserting that `?f` yields valid solutions
on leaves, w.r.t. an input/output relation `?io`
that relates problems with acceptable solutions

SOFT can be used to formalize algorithm schemas, e.g.:

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

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

```
(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)))))
```

predicate asserting that `?g` combines valid solutions on subtrees into valid solutions on trees, w.r.t. the input/output relation `?io` that relates problems with acceptable solutions

SOFT can be used to formalize algorithm schemas, e.g.:

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

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

```
(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)))))
```

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

theorem asserting the correctness of `fold[?f_?g]`, w.r.t. the input/output relation `?io` that relates problems with acceptable solutions

SOFT can be used to formalize algorithm schemas, e.g.:

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

(defun-sk2 atom-io[?f_?io] (?f ?io) ()
  (forall x (implies (atom x)
                     (?io x (?f x)))))

(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)))))

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

Algorithm schemas are useful for program refinement.

SOFT can be used for program refinement.

this could be
a `defun-sk2`

requirements over $n \geq 1$ target functions are specified by a 2nd-order predicate

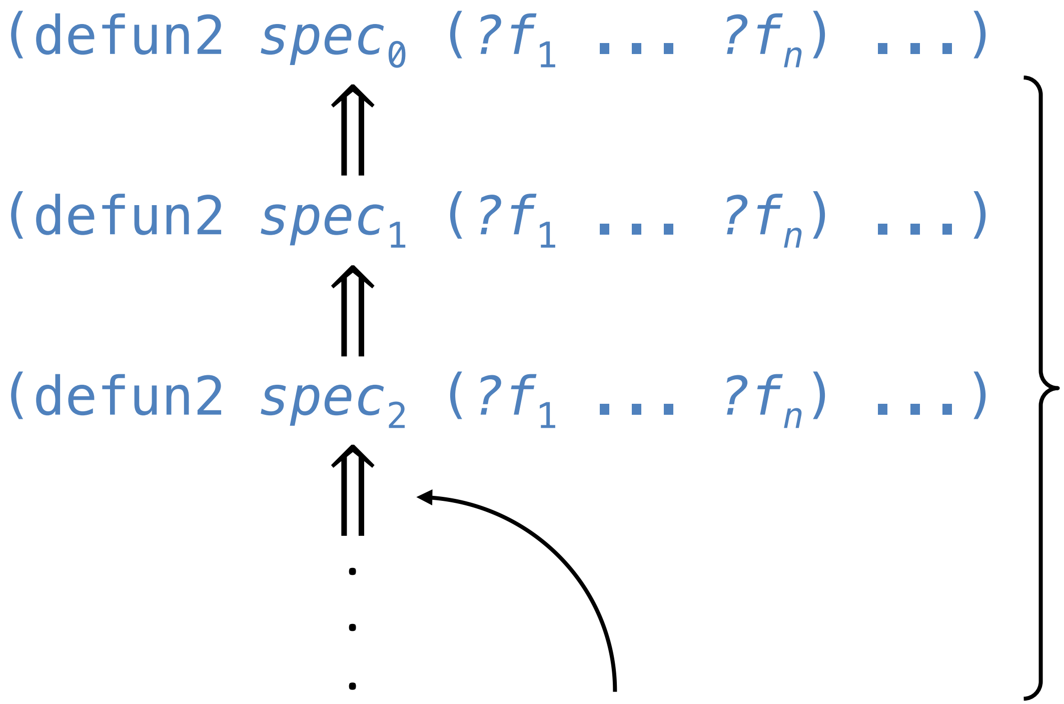
$$\texttt{(defun2 } spec_0 \texttt{ (?}f_1 \texttt{ ... ?}f_n\texttt{) ...)}$$

unlike `encapsulate`, no witnesses for the *n* functions are needed; it is the goal of refinement to generate witnessing implementations

goal: solve $spec_0$ for $?f_1$, …, $?f_n$

inconsistent requirements amount to $spec_0$ being always false, without introducing inconsistencies; no `defaxiom` is used

SOFT can be used for program refinement.



`(defun2` $spec_0$ `(`$?f_1$ `...` $?f_n$`) ...)`

⇑

`(defun2` $spec_1$ `(`$?f_1$ `...` $?f_n$`) ...)`

⇑

`(defun2` $spec_2$ `(`$?f_1$ `...` $?f_n$`) ...)`

⇑

⋮

the specification is stepwise refined by a sequence of increasingly strong 2nd-order predicates

each predicate narrows down the possible implementations or rephrases their description towards their determination

the refinement relation is logical implication

SOFT can be used for program refinement.

```
(defun2 spec0 (?f1 ... ?fn) ...)
             ⇑
(defun2 spec1 (?f1 ... ?fn) ...)
             ⇑
(defun2 spec2 (?f1 ... ?fn) ...)
             ⇑
             ·
             ·
             ·
             ⇑
(defun2 specm (?f1 ... ?fn) ()
  (and (equal ?f1 f1)
       ...
       (equal ?fn fn)))
```

$\langle f_1, \ldots, f_n \rangle$ is the obtained implementation of $spec_0$

$f_1, \ldots, f_n$ are executable functions, constructed as part of the refinement process, along with $spec_1, \ldots, spec_m$

the sequence ends with a 2nd-order predicate that provides explicit solutions $f_1, \ldots, f_n$ for the target functions

almost like this

SOFT can be used for program refinement.

(defun2 spec_0 (?f_1 ... ?f_n) ...)

⇑

(defun2 spec_1 (?f_1 ... ?f_n) ...)

⇑

(defun2 spec_2 (?f_1 ... ?f_n) ...)

⇑

.
.
.

⇑

(defun2 spec_m (?f_1 ... ?f_n) ()
  (and (def_1)
       ...
       (def_n)))

1st-order expression of the 2nd-order equality

(defun-sk2 def_i (?f_i) ()
  (forall args
    (equal (?f_i args)
           (f_i args))))

SOFT can be used for program refinement.

(defun2 $spec_0$ ($?f_1$ ... $?f_n$) ...)

⇑

(defun2 $spec_1$ ($?f_1$ ... $?f_n$) ...)

⇑

(defun2 $spec_2$ ($?f_1$ ... $?f_n$ $?f_{n+1}$) ...)

auxiliary target functions may
be introduced along the way

⇑

.
.
.

⇑

(defun2 $spec_m$ ($?f_1$ ... $?f_n$ ... $?f_{n+p}$) ()
  (and (def_1)
       ...
       (def_n)
       ...
       (def_{n+p})))

SOFT can be used for program refinement.

(defun2 $spec_0$ ($?f_1$ ... $?f_n$) ...)

$\Uparrow$

(defun2 $spec_1$ ($?f_1$ ... $?f_n$) ...)

$\Uparrow$

(defun2 $spec_2$ ($?f_1$ ... $?f_n$ $?f_{n+1}$) ...)

$\Uparrow$

.

.

.

$\Uparrow$

(defun2 $spec_m$ ($?f_1$ ... $?f_n$ ... $?f_{n+p}$) ()
  (and (def_1)
       ...
       (def_n)
       ...
       (def_{n+p})))

this approach to program refinement is called ‘shallow pop-refinement’ (see paper for details)

Example of program refinement using SOFT:

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

(defun-sk io (x y)
  (forall e (iff (member e y)
                 (and (leaf e x)
                      (natp e)))))

(defun-sk2 spec[?h] (?h) ()
  (forall x (io x (?h x))))
```

requirements specification

input/output relation

return a list of all and only the leaves that are naturals (in no particular order and possibly with duplicates)

Example of program refinement using SOFT:

since the specification involves binary trees, we use the divide-and-conquer algorithm schema for binary trees

```
(defun-sk2 spec[?h] (?h) ()
  (forall x (io x (?h x))))
```

this refinement step introduces auxiliary target functions; `?h` is determined when `?f` and `?g` are determined

strict implication; this refinement step reduces the set of possible implementations

$\Uparrow$

```
(defun2 spec1[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (forall x (io x (?h x)))))
```

we constrain `?h` to be `fold[?f_?g]` for some `?f` and `?g`

2nd-order equalities and inlined quantifiers are artistic licenses (the real version uses suitable `defun-sk2`s)

Example of program refinement using SOFT:

```
(defun-sk2 spec[?h] (?h) ()
  (forall x (io x (?h x))))
```

⇑

we use the first conjunct to rewrite the second conjunct

```
(defun2 spec1[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (forall x (io x (?h x)))))
```

non-strict refinement (equivalence)

⇑

```
(defun2 spec2[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (forall x (io x (fold[?f_?g] x)))))
```

Example of program refinement using SOFT:

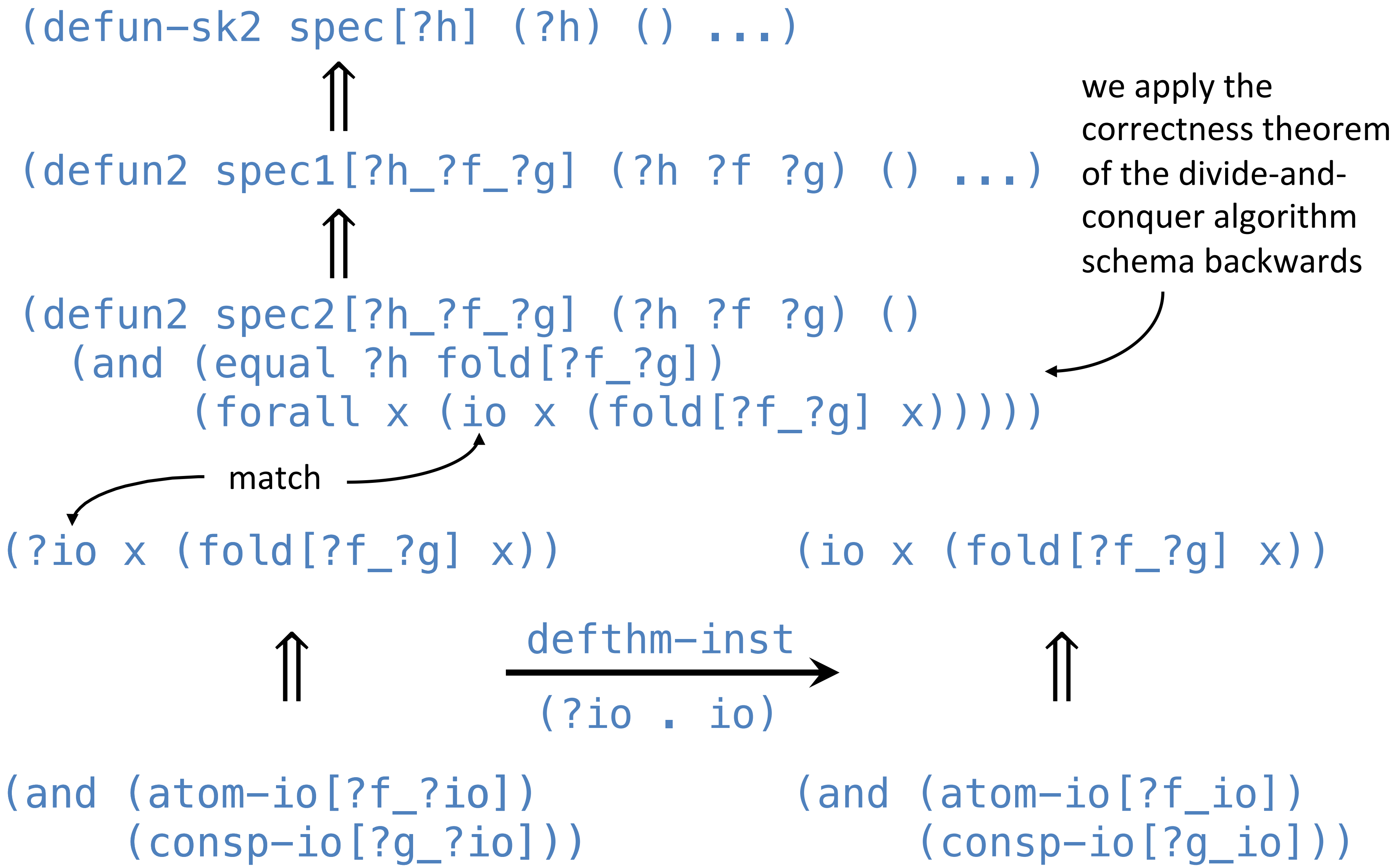
```
(defun-sk2 spec[?h] (?h) () ...)
              ⇑
(defun2 spec1[?h_?f_?g] (?h ?f ?g) () ...)
              ⇑
(defun2 spec2[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (forall x (io x (fold[?f_?g] x)))))
```

we apply the correctness theorem of the divide-and-conquer algorithm schema backwards

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

correctness theorem of the divide-and-conquer algorithm schema

Example of program refinement using SOFT:



```
(defun-sk2 spec[?h] (?h) () ...)
```

⇑

```
(defun2 spec1[?h_?f_?g] (?h ?f ?g) () ...)
```

⇑

```
(defun2 spec2[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (forall x (io x (fold[?f_?g] x)))))
```

we apply the correctness theorem of the divide-and-conquer algorithm schema backwards

match

```
(?io x (fold[?f_?g] x))
```

```
(io x (fold[?f_?g] x))
```

⇑

`defthm-inst`

`(?io . io)`

⇑

```
(and (atom-io[?f_?io])
     (consp-io[?g_?io]))
```

```
(and (atom-io[?f_io])
     (consp-io[?g_io]))
```

Example of program refinement using SOFT:

```
(defun-sk2 spec[?h] (?h) () ...)
             ⇑
(defun2 spec1[?h_?f_?g] (?h ?f ?g) () ...)
             ⇑
(defun2 spec2[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (forall x (io x (fold[?f_?g] x)))))
             ⇑
(defun2 spec3[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (atom-io[?f_io])
       (consp-io[?g_io])))
```

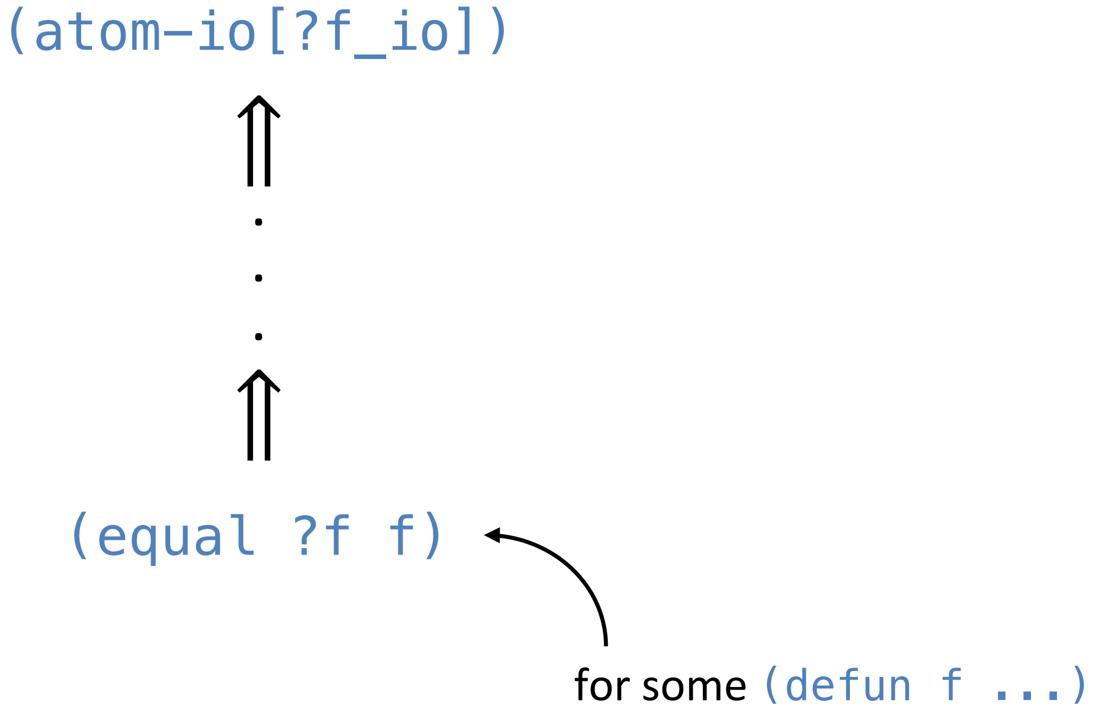
we apply the correctness theorem of the divide-and-conquer algorithm schema backwards

Example of program refinement using SOFT:

```
(defun-sk2 spec[?h] (?h) () ...)
              ⇑
(defun2 spec1[?h_?f_?g] (?h ?f ?g) () ...)
              ⇑
(defun2 spec2[?h_?f_?g] (?h ?f ?g) () ...)
              ⇑
(defun2 spec3[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (atom-io[?f_io])      }
       (consp-io[?g_io])))   }
```

these are requirements specifications for `?f` and `?g` that can be stepwise refined independently

Example of program refinement using SOFT:



`(atom-io[?f_io])`

⇑

⋮

⇑

`(equal ?f f)`

for some `(defun f ...)`

Example of program refinement using SOFT:

`(atom-io[?f_io])`

⇑

.

.

.

⇑

`(equal ?f f)`

we apply this implication backwards
in the main refinement sequence

Example of program refinement using SOFT:

```
(defun-sk2 spec[?h] (?h) () ...)
             ⇑
(defun2 spec1[?h_?f_?g] (?h ?f ?g) () ...)
             ⇑
(defun2 spec2[?h_?f_?g] (?h ?f ?g) () ...)
             ⇑
(defun2 spec3[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (atom-io[?f_io])
       (consp-io[?g_io])))
             ⇑
(defun2 spec4[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (equal ?f f)
       (consp-io[?g_io])))
```

Example of program refinement using SOFT:

```
(defun-sk2 spec[?h] (?h) () ...)
```

⇑

.
.
.

⇑

```
(defun2 spec4[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (equal ?f f)
       (consp-io[?g_io])))
```

we proceed analogously for `?g`

```
(consp-io[?g_io])
```

⇑ (stepwise)

```
(equal ?g g)
```

Example of program refinement using SOFT:

```
(defun-sk2 spec[?h] (?h) () ...)
             ⇑
             .
             .
             .
             ⇑
(defun2 spec4[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (equal ?f f)
       (consp-io[?g_io])))
             ⇑
(defun2 spec5[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (equal ?f f)
       (equal ?g g)))
```

Example of program refinement using SOFT:

```
(defun-sk2 spec[?h] (?h) () ...)
```

⇑

.

.

.

⇑

```
(defun2 spec5[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[?f_?g])
       (equal ?f f)
       (equal ?g g)))
```

we use the second and
third conjuncts to
rewrite the first conjunct

Example of program refinement using SOFT:

```
(defun-sk2 spec[?h] (?h) () ...)
```

⇑

.
.
.

⇑

```
(defun2 spec5[?h_?f_?g] (?h ?f ?g) ()
  (and (equal ?h fold[f_g])
       (equal ?f f)
       (equal ?g g)))
```

⟨`fold[f_g] f`, `g`⟩ is the obtained implementation of `spec`

the implementation witnesses the consistency of `spec`

SOFT is available in the ACL2 community books:

```
tools/soft.lisp
tools/soft-paper-examples.lisp
```