Automatically Computing Functional Instantiations

J Strother Moore
Department of Computer Sciences
Taylor Hall 2.124 C0500
University of Texas at Austin
1 University Station
Austin, TX 78712 USA
moore@cs.utexas.edu

ABSTRACT

Among the standard books distributed with ACL2 is the `consider-hint` book in the `hints` subdirectory, which implements a heuristic for computing functional instantiations. The implementation of the hint involves four basic algorithms: a second-order pattern matching algorithm that can compute instantiations for constrained and defined functions that call constrained functions, a process for generating variants of a term obtained by applying equations, a process for extending second-order matching through definitions so that when instantiating defined functions the algorithm can pick up appropriate bindings for the constrained functions inside the definitions, and an algorithm for sorting among likely functional substitutions. The second-order matching algorithm is an incomplete and slightly extended implementation of the Huet-Lang algorithm. We describe the four basic algorithms involved in guessing functional instantiations. We briefly suggest further work required to make the utility robust and suggest a new feature that could be added to ACL2 if this utility were sufficiently robust. We hope that some enterprising user or student will take up these challenges.

General Terms

theorem proving, program verification, mechanizing reasoning

Keywords

second order matching problem, definitional schemes, Huet-Lang algorithm, equational matching

Categories and Subject Descriptors

F.4 [**Mathematical Logic and Formal Languages**]: Miscellaneous; G.4 [**Mathematical Software**]

1. EXAMPLE

Consider the function `filter-map-h`. This function maps over its argument, which it treats as a list of elements, it identifies each element having property `hp` and it collects the result of applying `h` to such elements. Here we imagine that `hp` and `h` are constrained function symbols – function symbols that are undefined but perhaps known to satisfy some constraining axioms.

```
(DEFUN filter-map-h (x)
  (IF (ENDP x)
      NIL
      (IF (hp (CAR x))
          (CONS (h (CAR x))
                (filter-map-h (CDR x)))
          (filter-map-h (CDR x)))))
```

We might prove this theorem about `filter-map-h`

```
(defthm FILTER-MAP-H-APPEND
  (EQUAL (filter-map-h (APPEND y z))
         (APPEND (filter-map-h y)
                 (filter-map-h z))))
```

Here we think of `hp`, `h`, and `filter-map-h` as function-valued variable symbols because they may be usefully instantiated by functional substitutions.

Now consider this "concrete" function

```
(DEFUN BUMPER (U V W)
  (IF (CONSP U)
      (CONS (+ (* W (CAR U)) V)
          (BUMPER (CDR U) V W))
      NIL))
```

This function multiplies each element of `u` by `w` and adds `v`, collecting the results. It is possible to "instantiate" `filter-map-h` so that it is equal to `BUMPER`. To make this precise, it is possible to prove

```
(EQUAL (BUMPER (APPEND A B) I J)
       (APPEND (BUMPER A I J)
               (BUMPER B I J)))
```

from `filter-map-h-append` by functional instantiation [3, 4], specifically with the functional substitution:

```
filter-map-h := (LAMBDA (X) (BUMPER X I J))
```

```
hp          := (LAMBDA (X) T)
h           := (LAMBDA (X) (+ (* J X) I))
y           := A
z           := B
```

This instantiation is computed automatically by the `:consider` hint, which is implemented in the distributed ACL2 book `consider-hint` in the `hints` subdirectory. In particular, if the consider hint book has been included and the command `(add-consider-hint)` executed, the following succeeds:

```
(DEFTHM BUMPER-APPEND
 (EQUAL (BUMPER (APPEND A B) I J)
        (APPEND (BUMPER A I J)
                (BUMPER B I J)))
 :hints (("Goal" :consider FILTER-MAP-H-APPEND)))
```

This paper explains the basic algorithms used in the `:consider` hint. They are

- the Huet-Lang second-order matching algorithm for computing substitutions under which "second order" terms are syntactically identical to first order ones
- a rewrite utility that implements certain simple equational identities like `(endp x)` = `(not (consp x))`.
- a "driver" that iteratively extends substitutions, by diving into and matching bodies of defined generic functions and their alleged concrete counterparts
- a scoring heuristic for selecting the "best" of many alternative substitutions.

We leave it to the reader to imagine how all this is tied together to implement the `:consider` hint. The basic idea, however, is that the user supplies the name of a ("second-order") theorem and (optionally) a pattern, a target term, and/or an initial functional substitution to be extended. When not supplied, the last three arguments to the `:consider` hint default in specified ways. The target may be specified in terms of the goal clause, whose exact form may not be known when the hint is given. When the indicated goal arises, the arguments are determined, the algorithm discussed here is used to select all of the highest scoring functional substitutions, and an `:or` hint is generated to force the system to consider each possibility with the prover. Each branch of the `:or` contains a manufactured `:use` hint that provides one of the highest scoring functional substitutions computed. If any branch succeeds, the proof succeeds.[1]

We apologize to the reader for the absence of sufficient documentation on how to use the `:consider` hint. If you include the book `books/hints/consider-hint` and type `:doc consideration` you will see the beginnings of the documentation, which explains how to specify the theorem and optional pattern, target, and substitution seed.

[1]The user wishing to see the actual functional substitutions selected by a given application of a `:consider` hint should execute `(show-custom-keyword-hint-expansion t)` before the proof in question. This causes ACL2 to print the expansion of all custom keyword hints, of which `:consider` hint is one.

We do not describe the `:consider` hint in this paper. It is implemented as a certified book in terms of "custom keyword" hints and the `:or` hint, both of which are documented in the online documentation. Because of this, the soundness of the algorithms described here is not at issue: if they compute incorrect instantiations, the `:consider` hint's application will cause the theorem prover to fail.

2. THE HUET-LANG ALGORITHM

2.1 Abstract Description

"Matching" is generally used to mean finding a substitution on the variable symbols in a term so that the instance of that term is another term. Unlike unification, matching gets to instantiate only the variable symbols in one of the terms.

While second-order *unification* is undecidable, second-order *matching* is NP-complete. Certain restricted cases of the matching problem are solvable in polynomial time and other restricted cases are still NP-complete [1]. Here, we are interested in the unrestricted case and implement the Huet-Lang second-order matching algorithm [2].

First we establish some terminology. We sometimes call constrained function symbols *functional variables* and divide them into two classes, *simple constrained* symbols like `hp` and `h` which are introduced directly with `encapsulate`, and *defined constrained* symbols like `filter-map-h` which were introduced with `defun` but which use constrained symbols hereditarily in their bodies.

A *second order* term is an ACL2 term in which some of the function symbols are (simple or defined) constrained function symbols. For the purposes of exposition, we will sometimes write instantiable variable symbols, both functional and first-order, in lower case typewriter font and uninstantiable variable symbols, axiomatized ACL2 primitives, and defined (unconstrained) function symbols in upper case typewriter font. We often call the latter function symbols *concrete* symbols. By "uninstantiable variable symbols" we mean first-order ("individual") variable symbols occurring in the target term and unavailable for instantiation by the matching process. However, there is in fact no distinction between `x` and `X` in ACL2 and the distinction we are making typographically is implemented other ways.

Thus, `hp` and `filter-map-h` are constrained function symbols and might be called functional variable symbols. `CAR` and `BUMPER` are concrete function symbols. `(F (G x) y)` is a (first-order and instantiable) term, `(F (G X) Y)` is a candidate target term, containing no instantiable symbols and might sometimes be thought of like the ground term `(F (G (X) (Y)))`, and `(f u v)` is a second-order term.

Italics are used only for meta-variables; f generally denotes some function symbol. We sometimes use F if we know the denoted symbol is concrete.

In this section we assume our input terms do not contain lambda expressions. The implementation will allow but eliminate lambda expressions.

A substitution is a finite map from constrained function symbols and individual variable symbols. Individual variable

symbols are mapped to first-order terms. Constrained function symbols of arity n are mapped to lambda expressions of the form `(LAMBDA (`$v_1 ... v_n$`) `β`)` where β is a term whose only free variable symbols are among the v_i. Thus, { `u := (G (X))`; `v := (Y)` ; `f := (LAMBDA (i j) (F i j))` } is a substitution.

The result, $t/\sigma = t'$, of applying a substitution σ to a term t is obtained as follows.

- If t is a variable symbol in the domain of σ, t' is $\sigma(t)$.
- If $t =$ `(`$f \;\; a_1 \ldots a_k$`)` where f is in the domain of σ, then t' is the beta reduction of `(`$\sigma(f) \;\; a_1/\sigma \ldots a_n/\sigma$`)`
- If t is a variable symbol not in the domain of σ, or t is a constant, t' is t.
- Otherwise, t is of the form `(`$f \;\; a_1 \ldots a_k$`)` and t' is `(`$f \;\; a_1/\sigma \ldots a_n/\sigma$`)`.

The *beta reduction* of `((LAMBDA (`$v_1 \ldots v_n$`) `β`) `$\alpha_1 \ldots \alpha_n$`)` is $\beta/\{v_1 := \alpha_1; \ldots v_n := \alpha_n\}$. Thus, the beta reduction of `((LAMBDA (i j) (F i j)) (G (X)) (Y))` is `(F (G (X)) (Y))`.

A *matching challenge* E is a finite set of ordered pairs, $\langle t_i, s_i \rangle$ such that t_i is a (possibly) second order term and s_i is a first order ground term.[2]

If $\langle t_i, s_i \rangle$ is a pair in a matching challenge, we call t_i the *pattern* and s_i the *target*.

It is convenient to write E/σ to denote the matching challenge obtained from another by applying σ to each of the patterns in E.

The question raised by a matching challenge is whether there exists a σ such that t_i/σ is identical to s_i, for every i? That is the *second-order matching problem.*

To determine whether it is possible to instantiate a second order pattern term t to get a first order target term s, one solves the matching challenge for $\{\langle t, s \rangle\}$.

The Huet-Lang algorithm solves the second-order matching problem [2]. In practice, the algorithm collects and returns all possible substitutions. We describe it below without discussing how the substitutions are recovered, but that will become obvious. The algorithm as described here is based on five transformation rules that map matching challenges to matching challenges. We say $E \Rightarrow E'$ if there is a transformation that maps E to E'. We say $E \Rightarrow^* E'$ if there is a finite sequence of E_i such that $E \Rightarrow E_1 \Rightarrow E_2 \ldots \Rightarrow E'$.

The Huet-Lang Theorem is that a suitable substitution exists for E if and only if $E \Rightarrow^* \phi$, where ϕ is the empty set and the five transformations are:

[2]In the literature, E is called a *matching expression* rather than a matching challenge, but we follow the ACL2 convention of using the word "expression" synonymously with "term."

- **Identity**:
  $\{\langle s, s \rangle\} \cup E \Rightarrow E$.
- **Binding**:
  $\{\langle v, s \rangle\} \cup E \Rightarrow E/\{v := s\}$, where v is an individual variable symbol.
- **Simplification**:
  $\{\langle (F\ t_1 \ldots t_n), (F\ s_1 \ldots s_n) \rangle\} \cup E$
  $\Rightarrow \{\langle t_1, s_1 \rangle, \ldots \langle t_n, s_n \rangle\} \cup E$.
- **Projection**:
  $E \Rightarrow E/\{f :=$ `(LAMBDA (`$v_1 \ldots v_n$`) `v_i`)`$\}$, if one of the elements of E is $\langle$`(`$f \;\; t_1 \ldots t_n$`)`$, s\rangle$ where f is a constrained function symbol.
- **Imitation**:
  $E \Rightarrow E/\{f :=$ `(LAMBDA (`$v_1 \ldots v_n$`)`
  `(`F `(`$h_1 \;\; v_1 \;\ldots\; v_n$`)`
  $\ldots$
  `(`$h_m \;\; v_1 \;\ldots\; v_n$`)))`$\}$,

  if E contains $\langle$`(`$f \;\; t_1 \ldots t_n$`)`$, $`(`$F \;\; s_1 \ldots s_m$`)`$\rangle$ where f is a constrained function symbol and the h_i are new constrained function symbols.

Thus, the "goal" of the algorithm is to eliminate each of the pairs from E by applying the rules. But one must consider applying the rules in all possible ways and the rules are not mutually exclusive.

The first three rules enable recursive descent through terms with identical concrete function symbols, binding individual variable symbols to first order terms when necessary. For example, when two identical terms are paired, they match and we can just drop the pair (Identity). When a first-order variable symbol is paired with a first-order term, we get a match by dropping the pair but substituting the term for the variable symbol (Binding). When two terms with the same concrete function symbol are paired, we can get a match by dropping the pair and replacing it by pairs requiring us to match corresponding arguments (Simplification).

The other two rules both concern the case of a pair in which the pattern is the application of a constrained function symbol f to some t_j.

Projection implies that we must consider that this unknown f simply returns one of its arguments and we must try all of the possibilities. An effect of applying the substitution is the beta reduction that gets rid of the lambda application and replaces it by the corresponding actual, t_i.

Imitation implies that if the target is a call of a (necessarily concrete) function symbol F on some actuals s_k, we could arrange `(`$f \;\; t_1 \ldots t_n$`)` to match it by letting the unknown function symbol f just call F on some terms that we arrange to be those same actuals, constructed by new unknown function symbols, h_i, of the t_j. Again, the beta reduction implicit in applying a substitution followed by Simplication will eliminate the lambda application in favor of matching problems between the compositions of the h_i on the t_j versus the s_k.

It should be clear now how to accumulate the substitutions. Given a path to ϕ through a sequence of transformations, the substitution is just the set of substitution assignments mentioned in the Binding, Projection, and Imitation transformations along that path.

2.2 ACL2 Implementation

It is not difficult to implement this algorithm in ACL2. Because the sources are available in the standard distribution of ACL2 books, we do not reproduce them here. The crux of the algorithm is named `hl-one-way-unify1` (the "`hl`" stands for Huet-Lang, "`one-way-unify`" is ACL2's algorithm for pattern matching, binding only variable symbols from the pattern, and the "`1`" is our convention indicating that this is an auxiliary function to the top-level entry to the algorithm which is named `hl-one-way-unify`). These functions are found in the file `books/hints/huet-lang-algorithm.lisp`. The top-level entry takes care of some initialization and, mainly, a change from ACL2's term representation discussed later. For brevity, we refer to `hl-one-way-unify1` here as *hl* and ignore programming, representational, and top-level entry details.

The function *hl* takes five arguments: the pattern, *pat*, the first-order ground term, *term*, an integer, called *hmax*, used to help us generate new names for the h_i, a partial substitution already accumulated, here called ρ and the ACL2 logical world *wrld*.[3] By passing *wrld* to *hl* we give it access to ACL2's database, specifically so that it can distinguish constrained from concrete function symbols.

A call of *hl* returns a list of pairs, $(hmax' \ . \ \sigma)$, where each σ is a substitution extending ρ such that pat/σ is equal to *term* and $hmax'$ is the largest index of any h_i in σ. This allows *hl* to generate new symbols as it extends σ. The most obvious place this happens is when *hl* sweeps across successive argument positions, obtaining an *hmax* and σ suitable for matching the one position and then extends them to match the rest. We do not discuss *hmax* further.

When faithfully implemented, the Huet-Lang algorithm returns the list of all substitutions σ (up to equivalent variants on *pat*) such that pat/σ is identical to *term*. For heuristic reasons, our *hl* is not complete; some substitutions may be omitted. Furthermore, a σ returned by *hl* does not guarantee that pat/σ is syntactically identical to *term* but that pat/σ is provably equal to *term* under just a few built-in rewrite rules. We discuss these issues after describing the basic idea of the algorithm.

The algorithm recursively descends through *pat* and *term* extending ρ as it goes. If *pat* is a first-order variable symbol, we determine if it is bound in ρ. If so, its binding we must be *term* in order for matching to succeed. If *pat* is an unbound variable symbol, we can extend ρ by binding *pat* to *term*. In any case, *hl* returns either the empty list of substitutions (indicating failure to match) or the singleton list containing the winning substitution (ρ or its extension). The case when *pat* is a constant is similar but precludes the possibility of binding *pat*, of course. From this description of the base cases it should be obvious how to handle the case of matching the applications of concrete function symbols. If the leading symbols of *pat* and *term* are both concrete and different, the match fails. If they are identical, *hl* successively matches corresponding arguments of *pat* and *term*, extending *each* of the substitutions produced by matching the preceding arguments.

If *pat* is the application of a constrained function symbol f, we must first determine if f is bound in ρ. If so, it is bound to a lambda expression, and we beta reduce the application of that lambda expression to the argument terms in *pat* and then recursively match that to *term* under ρ.

If *pat* is the application of an unbound constrained function symbol f of arity n we must be prepared to do either Projection or both Projection and Imitation, depending on *term*. For example, if *term* is a variable symbol or a quoted constant, we can only do Projection: We generate n variations of ρ, each of which binds f to one of the n projections onto the arguments of f. Then we recursively attempt to match *pat* to *term* under each of those extensions, collecting the ones that succeed. On the other hand, if *term* is the application of a concrete function symbol F, we must try both Projection and Imitation. (The latter requires generating some new h_i and adjusting *hmax* as should be obvious.)

2.3 Incompleteness and Extensions

We lied above when we wrote "if *term* is a variable symbol or quoted constant, we can only do Projection". If *term* is a quoted constant, e.g., `'(1 2 3)` then we could actually do either Projection or Imitation, by thinking of *term* as `(cons 1 '(2 3))`. That is, the binding for f could be a projection onto one of n arguments or could be a lambda expression that calls `cons` and somehow matches new function symbols on the actuals of *pat* to `1` and `'(2 3)`. The source of incompleteness in our implementation is that we do not do Imitation in this case.

Consider matching the pattern `(g x)` with the ground term `'(0 . 0)`. Our code gives two solutions to this problem, one where `x` is the entire constant and `g` is the identity function, and the other where `x` is irrelevant and `g` is the constant function returning `'(0 . 0)`. But had we thought of `'(0 . 0)` as `(cons 0 0)` there would be five solutions, including, for example, one where `x` is `0` and `g` is `(lambda (v1) (cons v1 0))`. These solutions exploiting the internal structure of constants are missed by our code.

It is thus possible that we will find no solutions when in fact there are solutions. Consider, for example, `(IF (g x) (g y) x)` versus `(IF '(0 . 0) '(1 . 0) '0)`. Here, `IF` is just a convenient, concrete 3-place symbol. We find no solutions.[4] But had we phrased the ground term as `(IF (CONS`

[3] Our ρ is actually named `bindings` in the sources. There is another argument, named `restrictions`, that allows us to enforce various restrictions on how variable symbols are bound. But that feature is not used at the moment and is ignored here.

[4] We find two solutions to the first sub-problem, i.e., matching `(g x)` with `'(0 . 0)`. One of those lets `g` be the constant function returning `'(0 . 0)`, but that precludes us from solving the second sub-problem, i.e., matching `(g y)` with `'(1 . 0)`. Our other solution to the first sub-problem lets `g` be the identity and let `x` be `'(0 . 0)`. That allows us to solve the second sub-problem, but then we fail on the

`0 0) (CONS 1 0) 0)` we would find the solution $\{g := \texttt{(lambda (v1) (cons v1 0))}, \texttt{x} := 0, \texttt{y} := 1\}$

Unlike the rest of ACL2, where `'(1 2 3)`, `3`, `'ILOAD` and every other constant is just an abbreviation for a ground term on constructors, *hl* treats constants as atomic objects. We made this decision because ACL2 applications frequently involve huge constants and the explosion of substitutions is just too great.

We now turn to our claim that we find some substitutions σ such that pat/σ is provably equal (but not necessarily syntactically identical) to *term*. Because of the intended application of *hl*, namely to identify instances of definitional schemes (in the sense that `filter-map-h` is a definitional scheme), we find it convenient to include a few special cases for matching `IF` expressions.

Not only is the pattern `(IF (g ...)` pat_2 pat_3`)`, where `g` is a constrained function symbol, matched with a *term* of the form `(IF` t_1 t_2 t_3`)` as the description above says, it is also matched with `(IF (NOT` t_1`)` t_3 t_2`)`. Furthermore, if `(g ...)` can be matched with `T`, we just match pat_2 to *term* and if `(g ...)` can be matched to `NIL`, we just match pat_3 to *term*.

Thus, contrary to the description given, we can match the pattern `(IF (g x) 0 1)` to `(IF (F A) 1 0)`, for example, by binding `g` to `(lambda (v1) (NOT (F v1)))` and `x` to `A`.

2.4 Representational Tricks

Our *hl* does not actually represent terms quite the same way ACL2 does. It is convenient to change the representation of terms for purposes of efficiency. Our implementation actually operates on an extended representation called *pseudo-terms* which we illustrate below. We similarly extend the notions of substitution and matching challenge. We leave most of the details to the reader.

The h_i created by Imitation can ultimately all be eliminated by beta reduction. Since it can be expensive to create new function symbols, our pseudo-terms allow the use of integers as constrained function symbols. Thus, instead of generating h_i we use simply i. Similarly, the lambda expressions required by the transformations need not actually have formal variable symbols, v_i. In pseudo-terms, we use integers for those too. For example, `((lambda (1 2) (1 2 1)) A B)` is a pseudo-term which beta reduces to the pseudo-term `(1 B A)`, which denotes the application of some constrained function symbol ("h_1") to `B` and `A`. In fact, were we to bind f to that lambda expression we would simply write `(1 2 1)` because the flag in the binding "pair" tells us it is a lambda expression, the arity is known by context, and the formals are implicit.

Not all functional variable symbols are represented by integers. Only the h_i introduced by the algorithm are so represented. The constrained functions of the original pattern will be treated as functional variable symbols also. They retain their normal symbolic names. Thus, a term like `(x X)`, where `x` is a defined constrained function symbol, is problematic.[5] `X`, as a function symbol, may be bound to a lambda expression and `X`, as a variable symbol, may be bound to a term. To avoid this ambiguity, binding pairs in our substitutions are really triples: `(`*pat* *fnp* `.` *val*`)`, where *pat* is an individual or functional variable symbol, *fnp* is non-`nil` if *pat* is a functional variable symbol, and *val* is the binding. Our *fnp* can take on two non-`nil` values, `T` and `DONE`. We explain later.

When a term like `(REV1 x a)` occurs in a pattern, the `x` and `a` are instantiable variable symbols. But if it occurs in a target, i.e., if we would write it here as `(REV1 X A)`, those same symbols must be regarded as uninstantiable. However, the location of the occurrence (pattern versus target) is an inadequate indicator of the meaning of a variable symbol because the symbol `a` in the pattern `(REV1 x a)` may be replaced by the target symbol `X`, producing the pattern `(REV1 x X)` in which the first `x` is to be understood as a variable symbol and the second as a constant, when in fact `x` and `X` cannot be distinguished. We must make the original target term a ground term by replacing its variable symbols by constants. It might appear workable to convert the target `(REV1 X A)` to `(REV1 (X) (A))` but that assumes there are no constrained function symbols `X` and `A`. Instead, we replace the variable symbol v by `(:CONSTANT` v`)` in pseudo-terms.

Finally, we must be able to convert a pseudo-term back to a term. The main subtlety concerns such pseudo-terms as `(lambda (x) (rev1 x (:constant x)))` which we cannot convert to `(lambda (x) (rev1 x x))`! So during conversion we avoid capture and generate `(lambda (x1) (rev1 x1 x))`.

3. REWRITING TERMS IN "ALL POSSIBLE WAYS"

Effective use of *hl* to match definitional schemes to concrete definitions requires considering some simple transformations of the concrete term since different authors use different definitional idioms, e.g., `consp` versus `endp`, etc.

We implement a simple rewrite engine that, used iteratively, will rewrite a term in all possible ways under a given set of unconditional rewrite rules. But, since we are searching for a variant of *term* that *hl*-matches *pat*, we use *hl* matching after each iteration of the simple engine.

3.1 The Simple Rewrite Engine

For example, applying the following ground rewrite rules (here phrased as equalities but used left to right)

```
((EQUAL (G '1) (G '2))       ; rule-g1
 (EQUAL (G '2) (G '3))       ; rule-g2
 (EQUAL (F (G '1)) (F '1))   ; rule-f-g1
 (EQUAL (F (G '2)) (F '2))) ; rule-f-g2
```

to the term `(F (G '1))`, our simple rewrite engine produces a list of four results:

```
((F (G '2))                  ; use rule-g1 on (g '1)
 (F '2)                      ; use rule-g1 on (g '1)
                             ;  and then rule-f-g2
```

[5]Recall, `x` and `X` are the same symbol; case is being used here to indicate class.

```
(F (G '1))                ; use no rules
(F '1))                   ; use rule-f-g1
```

Note that we never used `rule-g2` and hence never produced `(F (G '3))`. That is because we would have had to apply `rule-g2` to the immediate output of `rule-g1`. Instead, our simple rewrite engine ascends after applying rules to the arguments and rewrites the resulting calls.[6]

However, if we apply the simple engine again to each of the outputs above, we will rewrite `(F (G '2))` and obtain the additional list `((F (G '3)) (F (G '2)) (F '2))`. Note that this naive process will introduce duplications produced by earlier iterations.

3.2 Iteration and Search

Because the rules may not converge, we impose an artificial cutoff on the number of iterations. In our experiments, we typically use just one round.

Furthermore, because we are actually searching for a version of *term* that *hl*-matches *pat*, we do *hl* pattern matching on every variant produced before we iterate with the simple engine. We stop as soon as we find a match, producing a list of functional substitutions.

Note that stopping when we find the first *hl* match is a radical abandonment of completeness under the given rules. It could well happen that *hl* would find "better" substitutions if we allowed rewriting to proceed further.

3.3 The Rules

Since we are looking for instances of definitional schemes, the rules we apply single out a particular function symbol, namely that of the concrete function alleged to be an instance of the scheme, e.g., `BUMPER` in our example. If F is that function symbol (and assuming below that F is of arity 2), then the rules we apply are

```
((EQUAL (IF xi (F yi v2) (F zi v2))            ; 1
        (F (IF xi yi zi) v2))
 (EQUAL (IF x (F v1 v2) (F w1 w2))             ; 2
        (F (IF x v1 w1) (IF x v2 w2)))
 (EQUAL (IF xi (F v1 yi) (F v1 zi))            ; 3
        (F v1 (IF xi yi zi)))
 (EQUAL (IF x (F v1 v2) (F w1 w2))             ; 4
        (F (IF x v1 w1) (IF x v2 w2)))
 (EQUAL (IF (CONSP lst)                        ; 5
            (IF test lst else1)
            else2)
        (IF (CONSP lst)
            (IF (IF test lst 'NIL)
                (IF test lst 'NIL)
                else1)
            else2))
 (EQUAL (ENDP x) (NOT (CONSP x)))              ; 6
 (EQUAL (IF p out1 (IF q out2 out3))           ; 7
        (IF (IF p 'T q) (IF p out1 out2) out3))
 (:META FOLD-TO-ISOLATE))                      ; 8
```

[6]Sweeping across the arguments produces a list of equivalent terms for each argument position; we then construct all possible combinations of those terms to obtain the "resulting calls."

Note that the number of rules depends on the arity of F. Rules 1 and 3 combine two calls of F that differ only in a single argument position. Rules 2 and 4 combine two calls of F that differ (potentially) in each argument position. We call these four rules "isolation" rules because they tend to create isolated occurrences of the target function symbol. The last rule, 8, is a metafunction that isolates more general calls of F. These five rules are the only ones that depend on F.

Rule 8 introduces a lambda application to isolate calls of the target function symbol F in which different things are done to the output of the two calls. For example, it will transform

```
(IF τ (φ (F α1 α2)) (ψ (F β1 β2)))
```

to

```
((LAMBDA (Z)
         (IF τ (φ Z) (ψ Z)))
 (F (IF τ α1 β1))
    (IF τ α2 β2)))
```

with appropriate passing through the lambda of the variable symbols used in the schemas denoted by Greek letters. The simpler isolation rules will not handle this situation because of the presence of ϕ and ψ.

The very peculiar rule 5 is present only to allow us to recognize `MEMBER-EQUAL`, for example, as an `OR` of some predicate on successive tails. The complication is that `MEMBER-EQUAL` is not an `OR` of the predicate it tests, because it tests `(EQUAL E (CAR LST))` but returns `LST`. This can be "fixed" if `MEMBER-EQUAL` is rewritten so that the predicate it tests is `(IF (EQUAL E (CAR LST)) LST 'NIL)`.

Rule 6 just exposes the definition of `ENDP`.

Rule 7 implements another idiom: Sometimes authors combine two nearly identical cases into one by testing an `OR` and implement the finer-grained differences inside the handler. Other times they treat each disjunct independently.

Clearly, these rules are *ad hoc*. They are communicated to the simple rewrite engine as a list of explicit rules as shown, but the list is built-in (computed with respect to F). In the current implementation the set of rules cannot be extended by the user, but since the whole package is just a certified book, the user wishing to experiment could change them in the code.

4. DRIVING THROUGH DEFINITIONS

A naive attempt to use *hl*-matching to compute an instance of the pattern `(filter-map-h x)` to map `(BUMPER A I J)` would simply replace `filter-map-h` by a lambda expression calling `BUMPER`. This would fail as a functional instantiation because the constraints could not be proved unless the instantiation also picks up appropriate bindings for `hp` and `h` as used inside the definition of `filter-map-h`. Thus, to compute helpful functional substitutions it is necessary to drive through definitions of defined constrained function symbols

and try to match their bodies with the bodies of their proposed concrete counterparts.

Recall from subsection 2.4 that our substitution "pairs" are actually triples of the form `(`f flg `.` F`)` where a non-`nil` flag, flg, indicates that f is a function variable symbol rather than an individual variable symbol. Furthermore, we cryptically indicated that the non-`nil` values used were `T` and `DONE`. We now explain.

A substitution is said to be *unfinished* if there is some binding in it with a flag of `T`. When *hl* binds function symbols it uses the flag `DONE` if the symbol denotes a simple constrained function and uses the flag `T` if the symbol denotes a defined constrained function.

The basic idea of driving through definitions is to pick an unfinished binding of some defined constrained f bound to a call of a concrete function symbol G and *hl*-match the body of f against the body of G after replacing G's formals by the actuals used in the lambda expression to which f is bound. This relatively straightforward description is complicated by the problem of keeping ones variable symbols straight.

Suppose we wish to find a functional substitution for instantiating pattern *pat* to yield concrete term *term*. Create an initial list of substitutions by *hl*-matching *pat* and *term* with iterated rewriting as described above. Call this list of substitutions *the pool*. Our goal is to extend some of these substitutions to pick up bindings for the subfunctions of unfinished bound function symbol. We proceed as follows.

1. If there are no unfinished substitutions in the pool, return the pool. Otherwise, let σ be one of the unfinished substitutions and let f be an unfinished functional variable symbol bound in it. By definition, f is a defined constrained function symbol. Let the formals used in the `defun` of f be `(`$x_1 \ldots$`)` and let the body be β. Let the binding of f in σ be `(lambda (`$v_1 \ldots$`)` τ`)`. Let β' be the result of renaming the x_i with the corresponding v_i so that it is now as though f had been defined in the first place with formals `(`$v_1 \ldots$`)` and body β'.

2. Is τ a call of a defined function symbol? If not, delete σ from the pool and start over at step 1.[7] Otherwise τ is `(`G $a_1 \ldots$`)`, where G is a concrete defined function symbol. Intuitively, the a_i involve the v_i, after all, `(`G $a_1 \ldots$`)` is the body of a lambda with formals `(`$v_1 \ldots$`)`. However, there may be free variable symbols in the a_i and not every v_i may be used. Let `(`$u_1 \ldots$`)` be the formals of the `defun` of G and let γ be the body. Let γ' be obtained from γ by replacing the u_i by the corresponding a_i, effectively expanding the call of G in the lambda binding of f and introducing v_i into γ'.

3. Let ρ be a renaming variable substitution that renames the v_i to distinct new variable symbols that do not occur bound in σ. Let w_i be the new name of v_i.

4. Let β'' be β'/ρ. Let γ'' be γ'/ρ. Roughly speaking, β'' is the body of f in which all the variable symbols are free in σ and γ'' is the expanded call of G on the same variable symbols.

5. Set the flag for the binding of f in σ to `DONE`, obtaining σ'.

6. Use *hl* with iterated rewriting on the pattern β'' and concrete term γ'' extending σ'. Obtain some list of extensions, Σ.

7. Remove from each substitution in Σ any binding of any w_i, obtaining Σ'.

8. Delete σ from the pool and add all the substitutions in Σ'.

9. Repeat from step 1.

We clean up the resulting substitutions by eliminating all h_i. This is easy since they are all ultimately bound to lambda expressions in the *hl* process (because the result of applying the substitution to *pat* is a concrete term). We also throw out duplicate substitutions. We also throw out duplicate substitutions.

The description above is necessarily brief and ignores some subtleties concerning free variable symbols in lambda applications. (Functional instantiations permit that, but the internal form of substitutions in *hl* does not.) The interested reader is urged to look at the code.

5. SCORING

The process above typically produces many functional substitutions. We next sort them according to a rather arbitrary heuristic scoring mechanism. The scoring mechanism exploits an oft-used philosophy among ACL2 developers: make the obvious easy. Completeness can be sacrificed as long as the obvious avenues are explored.

So what are the "obvious" uses of functional instantiation? The intuitions behind the scoring is as follows:

- Disfavor substitutions that do not bind all the functional variable symbols.
- Disfavor substitutions that use identity (projection) lambdas like `(lambda (...x...) x)`. The thinking is that if the user saw fit to provide a constrained function symbol in a definitional scheme it is odd not to use it, though it is easy to construct useful models with often-unused "hooks."
- Disfavor substitutions that use lambdas that do not use all their formals. Why did the user provide a formal in the scheme if it is not used?
- Disfavor substitutions that use lambdas that unnecessarily use free vars, e.g., `(lambda (x) (cons x A))` is disfavored if `(lambda (x) (cons x x))` would work.

[7] This is an inadequacy of our implementation and shows that we are focused on recursive definitional schemes and not just compositional ones.

For each of these criteria we compute a rational score between 0 and 1 indicating how well a given σ satisfies the criteria. For example, a substitution that binds four of five functional variable symbols scores `4/5`. We then sum the scores for each σ and choose the ones with the highest score.

To illustrate the necessity of some kind of scoring, consider matching the pattern `(h x)` with the concrete term `(CAR (CDR A))`. Below are the resulting functional substitutions and the score assigned to each.

```
((7/2 . ((x . A)
         (h . (LAMBDA (X) (CAR (CDR X))))))
 (7/2 . ((x . (CDR A))
         (h . (LAMBDA (X) (CAR X)))))
 (3   . ((x . (CAR (CDR A)))
         (h . (LAMBDA (X) X))))
 (5/2 . ((h . (LAMBDA (X) (CAR (CDR A)))))))
```

Thus, according to the heuristics used here, the two "most likely" choices for `h` are `CADR` and `CAR`. A less "likely" choice is the identity function. A final choice is the constant function that returns `(CAR (CDR A))`.

Of course, no syntactic scoring mechanism can solve the real problem: for which of these functional substitutions can we prove the (here unknown) constraints on `h`?

6. EXAMPLES

The `hints` subdirectory of the distributed books directory contains two files of interest here. One is the book `consider-hint-tests`, which contains many examples of the `:consider` hint, and the other is `huet-lang-tests`, which shows how to run specific matching problems and see the ranked substitutions produced.

Consider the classic generic iterator over a list:

```
(DEFUN generic-list-iterator (x ans)
  (COND ((ENDP x) ans)
        (T (generic-list-iterator (CDR x)
                                  (g (CAR x) ans)))))
```

where `g` is a constrained function symbol. This function is a tail-recursive mapper that applies `g` to each element of `x`, accumulating the answers in `ans`.

Here is a typical concrete function that uses an accumulator to collect all the integers in `x` that are greater than or equal to `min`.

```
(DEFUN GET-BIG-INTEGERS (X MIN A)
  (COND ((CONSP X)
         (COND ((AND (INTEGERP (CAR X))
                     (>= (CAR X) MIN))
                (GET-BIG-INTEGERS (CDR X)
                                  MIN
                                  (CONS (CAR X) A)))
               (T (GET-BIG-INTEGERS (CDR X) MIN A))))
        (T A)))
```

The concrete function is an instance of the generic one. In particular, this functional substitution works:

```
((g . (LAMBDA (X Y)
       (IF (INTEGERP X)
           (IF (< X MIN) Y (CONS X Y))
           Y)))
 (generic-list-iterator
    . (LAMBDA (X ANS)
       (GET-BIG-INTEGERS X MIN ANS)))
 (ans . A)
 (x . X))
```

The computation of this substitution critically requires isolating the recursive calls of `GET-BIG-INTEGERS` by pushing the `if`s in. Note the "creative" choice of `g` Thus, if we have

```
(DEFTHM GENERIC-LIST-ITERATOR-APPEND
  (EQUAL (generic-list-iterator (APPEND u v) a)
         (generic-list-iterator v
           (generic-list-iterator u a))))
```

then we can prove the following by instantiation

```
(thm
 (EQUAL (GET-BIG-INTEGERS (APPEND X Y) MIN Z)
        (GET-BIG-INTEGERS Y MIN
          (GET-BIG-INTEGERS X MIN Z)))
 :HINTS (("Goal"
          :CONSIDER GENERIC-LIST-ITERATOR-APPEND)))
```

The theorem to be instantiated need not be second-order. For example, suppose the associativity of `append` had been proved as `assoc-of-append` with `:rule-classes nil`. Then the following succeeds:

```
(THM (EQUAL (APPEND (APPEND A A) A)
            (APPEND A (APPEND A A)))
     :HINTS (("Goal" :CONSIDER ASSOC-OF-APPEND)))
```

This is because our function *hl* includes first-order matching (with iterated rewriting) as a special case.

7. CONCLUSION

There is much work remaining to make this a truly useful utility. Among the profitable avenues to explore are:

- Clean up the code in the both `consider-hint` and `huet-lang-algorithm`. The code evolved over several years of on-again-off-again work and there is "dead code" and other warts that obscure what is happening and make it harder to think about.
- Explore less explosive ways to consider "all possible rewrites," including modern matching algorithms that consider equational theories.
- Investigate ways to produce fewer candidate substitutions. Too often the system just grinds to a halt because of the explosion of possibilities. Because the user can "steer" the `:consider` hint by providing a seed substitution it is better, in our opinion, to abort and force the user to help than to run indefinitely. Since ACL2 aims, often, to make the obvious easy, completeness is not as important as insuring that the obvious choices are considered.

- Investigate ways to take into account the actual constraints on the function symbols being instantiated. This could actually lead to a reduction of the search space.

Once this utility is sufficiently robust, one could try to recognize when new concrete definitions are instances of known definitional schemas and automatically provide the corresponding lemmas about the concrete definitions.

Automatically instantiating existing "generic" lemmas would often require also automatically introducing some auxiliary concrete functions obtained by instantiating generic ones used in the lemmas. (Recall `filter-map-h` and imagine generic lemma about it that has a hypothesis, phrased in terms of a generic recursive function, checking that at least one element of `x` satisfies `hp`. The concrete counterpart of that function may not yet be introduced by the user who defined a concrete version of `filter-map-h` and stands ready to inherit its lemmas.)

This new feature would circumvent the use of functional instantiation in proofs (except to prove the lemmas) and would off-load the inefficiencies of this utility to definition-time processing. It would also provide a new way that experts could aid the novice: by capturing general definitional schemas and their properties and automatically providing rewrite rules.

However, as noted by a reviewer, the number of possible instantiations and the difficulty of choosing *a priori* among them may diminish the hope that one can produce an effective set of first-order lemmas from a generic one, unless the generic set contains enough structure and perhaps pragmatic advice to guide the selection of the relevant instantiations.

These critical improvements notwithstanding, we believe the basic ideas illustrated in here are very promising. It is our hope that by describing them as we have here some enterprising user or student will take up the challenge of extending this work.

8. ACKNOWLEDGEMENTS

As usual, I wish to thank the entire ACL2 community for helping push this work along. But I especially want to thank Warren Hunt for obtaining the funding that make it possible for me to work on it, Warren and Bill Legato for consistently pushing for features like this, and Matt Kaufmann who took on the unenviable task making my books compatible with the distribution. In addition, Matt provided valuable suggestions during the design of the algorithms.

9. REFERENCES

[1] Kouichi Hirata, Keizo Yamada, and Masateru Harao. Tractable and intractable second-order matching problems. In *In Proc. 5th Ann. Int. Computing and Combinatorics Conference (COCOON'99), LNCS 1627*, pages 432–441. Springer, 1999.

[2] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1997.

[3] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.

[4] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.