Functional Instantiation in First Order Logic

R. S. Boyer, D. M. Goldschlag,M. Kaufmann, and J S. Moore[1]

Technical Report 44 Revised January, 1991

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

TEL: +1 512 322 9951
FAX: +1 512 322 0656

[1]This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Orders 6082, 9151 and 7406. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.

Abstract

We describe a new facility, for a first-order-logic theorem prover [1], that permits the instantiation of theorems by the replacement of function symbols with new function symbols, provided certain axioms about the original function symbols can be proved about the new function symbols. Although this facility in effect provides something of the spirit of higher order logic, the underlying first-order logic itself is in no way extended. A proof of the correctness of the facility is provided, and many examples are given.

0.1 Introduction

In this paper we describe `CONSTRAIN` and `FUNCTIONALLY-INSTANTIATE`, two new user commands (events) that we have added to the "NQTHM" prover [1]. `FUNCTIONALLY-INSTANTIATE` implements a derived rule of inference that provides something of the flavor of higher order logic in that it permits one to infer new theorems by instantiating function symbols instead of variables. To be sure that such an instantiation actually produces a theorem, we first check that the formulas that result from similarly instantiating certain of the axioms about the function symbols being replaced are also theorems. Intuitively speaking, the correctness of this derived rule of inference consists of little more than the trivial observation that one may systematically change the name of a function symbol to a new name in a first-order theory without losing any theorems, modulo the renaming. However, we have found that this trivial observation has important potential practical ramifications in reducing mechanical proof efforts. We also find that this observation leads to superficially shocking results, such as the proof of the associativity of `APPEND` by instantiation rather than induction. And finally, we are intrigued by the extent to which such techniques permit one to capture the power of higher-order logic within first-order logic.

In order to make effective use of `FUNCTIONALLY-INSTANTIATE`, we have found it necessary to augment our facility for defining functions, `DEFN`, with a facility for constraining, but not completely characterizing, new function symbols. `CONSTRAIN` events are like `DEFN` events in that they add axioms about new function symbols consistently, i.e., an NQTHM history free of any use of `ADD-AXIOM` (the mechanism for adding an arbitrary axiom) is consistent, even if `DEFN` and `CONSTRAIN` are used repeatedly. We permit the introduction of several function symbols simultaneously with a single `CONSTRAIN`. `CONSTRAIN` is weaker than `DEFN` in the following sense. In general, any application of `CONSTRAIN` can be replaced by one or more `DEFN`s, and the axiom added by the `CONSTRAIN` can be proved after the `DEFN`s have been added. Intuitively, a good way to think about a `CONSTRAIN` event is to imagine defining a new function symbol, proving a theorem about that function symbol, and then forgetting the defining equation while remembering the theorem. In fact, in the implementation of `CONSTRAIN`, we insist that the user provide us with an already defined "witness" function and we check that the proposed new axiom is satisfied by the witness.

`FUNCTIONALLY-INSTANTIATE` implements a derived rule of inference. That is, anything that can be proved with `FUNCTIONALLY-INSTANTIATE` can be proved without it. `FUNCTIONALLY-INSTANTIATE` permits one to infer about a function symbol f anything that one has inferred about a function symbol g provided that the relevant axioms about g can be proved about f. It is intended that `FUNCTIONALLY-INSTANTIATE` will be used in coordination with `CONSTRAIN`.

0.2 Motivating Examples

0.2.1 Foldr v. Foldl

Consider the idea of iteratively applying some dyadic function $\oplus$ to the elements of a list, `x`, starting with some base value `n`. Let us denote the elements of the list by $\mathtt{x}_1$, $\mathtt{x}_2$, ... $\mathtt{x}_k$. Two algorithms come to mind. The first proceeds in an "inside-out" fashion and computes $\mathtt{x}_1 \oplus (\mathtt{x}_2 \oplus (\ldots\ (\mathtt{x}_k \oplus \mathtt{n})\ldots))$. The second algorithm proceeds in an "outside-in" fashion and computes $(\ldots((\mathtt{n} \oplus \mathtt{x}_1) \oplus \mathtt{x}_2) \oplus \ldots) \oplus \mathtt{x}_k$. In the functional programming language SASL[11] the inside-out result is produced by `foldr(`$\oplus$`, x, n)` while the outside-in result is `foldl(`$\oplus$`, x, n)`. We use those names below.

If $\oplus$ is commutative and the `foldl` algorithm is applied to the reverse of the original list, `'(`$\mathtt{x}_k$ `...` $\mathtt{x}_2$ $\mathtt{x}_1$`)`, the result is the same as the `foldr` algorithm. How can we say this in the first-order, quantifier-free logic of NQTHM?

We cannot define `FOLDR` and `FOLDL` to take functions as arguments. However, we could declare `FN` (with NQTHM's `DCL` command) to be an undefined function symbol of two arguments and then define the two folders to use `FN` explicitly:

```
(DEFN FOLDR-FN (X N)
  (IF (LISTP X)
      (FN (CAR X) (FOLDR-FN (CDR X) N))
      N))

(DEFN FOLDL-FN (X N)
  (IF (LISTP X)
      (FOLDL-FN (CDR X) (FN N (CAR X)))
      N))
```

After defining the `REVERSE` function we could then claim that `(FOLDR-FN X N)` is `(FOLDL-FN (REVERSE X) N)`, provided `FN` is commutative:

```
   ∀ X ∀ Y (EQUAL (FN X Y) (FN Y X))
 →
   (EQUAL (FOLDR-FN X N) (FOLDL-FN (REVERSE X) N)).
```

But this statement requires the universal quantifier, which is also outside of the NQTHM logic.

To tackle that problem we could add an axiom about `FN`, e.g.,

```
(ADD-AXIOM FN-IS-COMMUTATIVE (REWRITE)
  (EQUAL (FN X Y) (FN Y X)))
```

Since axioms (and theorems in general) are implicitly universally quantified this axiom accomplishes our goal of constraining `FN` to be commutative. (The token `REWRITE` has no logical significance; it only tells the system to use this equality as a rewrite rule.)

We could then state our first result:

```
(PROVE-LEMMA FOLDR-IS-FOLDL ()
  (EQUAL (FOLDR-FN X N) (FOLDL-FN (REVERSE X) N)))
```

This theorem can be proved automatically by NQTHM.

If there are no other axioms about `FN` we can informally regard `FOLDR-FN` and `FOLDL-FN` as two recursion schemas expressed in terms of an arbitrary commutative function `FN`, and we could regard `FOLDR-IS-FOLDL` as expressing the equivalence of those schemes (modulo the `REVERSE`). There are two problems however.

First, how do we know the axiom we added did not render the system inconsistent? In general, adding axioms to a logic as rich as NQTHM's is treacherous. Our solution is to avoid the addition of arbitrary axioms and instead encourage the use of a new, derived logical act, implemented as the `CONSTRAIN` event, which permits the introduction of new function symbols that are constrained by a given formula relating them. Logically speaking, `CONSTRAIN` requires that the constraint formula be satisfiable. The implementation enforces this by requiring the user to supply "witnesses" for the new symbols that make the constraint a theorem. That is, the user must show that existing functions of the logic have the desired relationship in order to constrain new functions to have that relationship.

For example, rather than `DCL FN` and then use `ADD-AXIOM` we could introduce `FN` with the new event

```
(CONSTRAIN FN-COMMUTATIVE (REWRITE)
           (EQUAL (FN X Y) (FN Y X))
           ((FN PLUS))).
```

Observe that the last argument to CONSTRAIN is a "functional substitution" that supplies the "witnesses" for the about-to-be introduced functions. In this case we use the Peano addition function, PLUS, as the witness for a commutative function. The system confirms that PLUS has the property required of FN, thus establishing the satisfiability of the proposed constraint, and then declares FN to be a function symbol of two arguments and adds the constraining axiom about FN.

The second problem noted above is that FOLDR-IS-FOLDL is not very useful to the NQTHM user, even if he informally understands its logical content. In particular, if the user defines two recursive functions, say FOLDR-TIMES and FOLDL-TIMES that are just instances of the two "schemas" above but use the defined function TIMES in place of FN, there is no direct way to use FOLDR-IS-FOLDL to deduce that the two new functions are equivalent. Of course, the two new functions could be proved equivalent: the proof would go just like the proof of FOLDR-IS-FOLDL, except that where the old proof appeals to the commutativity of FN the new one would appeal to the commutativity of TIMES.

We don't consider reconstructing the proof for two reasons. The first is merely practical: while we know a proof exists, it might take the system a long time to find it. This is especially true if the original proof was complicated or, more likely, the instantiation of the schemas is complicated. For example, if in the instantiation we replace FN by several pages of propositional logic, chances are NQTHM would fail to reproduce the analogous proof because its normalization procedures would trigger a combinatoric explosion. More importantly, the whole point of proving lemmas is to avoid the necessity of proving their instances. It does us no good to "informally regard" FOLDR-IS-FOLDL as the NQTHM statement of the equivalence if NQTHM cannot deduce the obvious from it.

To that end we introduce a new derived rule of inference, called "functional instantiation" by which we permit the immediate deduction of the equivalence of FOLDR-TIMES and FOLDL-TIMES from FOLDR-IS-FOLDL once it has been established that TIMES satisfies the constraints on FN. An example of the implemented event is

```
(FUNCTIONALLY-INSTANTIATE FOLDR-TIMES-IS-FOLDL-TIMES (REWRITE)
  (EQUAL (FOLDR-TIMES X N)
         (FOLDL-TIMES (REVERSE X) N))
  FOLDR-IS-FOLDL
  ((FOLDR-FN FOLDR-TIMES)
   (FOLDL-FN FOLDL-TIMES)
   (FN       TIMES)))
```

The last argument above is a functional substitution that makes explicit the correspondences between the old (`FN`-based) function symbols and the new (`TIMES`-based) symbols. This functional substitution is applied to the formula of `FOLDR-IS-FOLDL` and must produce the formula to be deduced. Since `FOLDR-IS-FOLDL` is

```
(EQUAL (FOLDR-FN X N)
       (FOLDL-FN (REVERSE X) N))
```

we have to replace `FOLDR-FN` by `FOLDR-TIMES` and `FOLDL-FN` by `FOLDL-TIMES` to produce the claimed relationship between the new functions. Observe that the substitution also replaces `FN` by `TIMES`, even though `FN` is not involved in the statement of the theorem we are instantiating. We explain below.

Roughly speaking, the proof obligation incurred in the use of functional instantiation is that every axiom about the old symbols must hold of the new symbols. Consider the axiom about `FOLDR-FN`, namely its definition:

```
(EQUAL (FOLDR-FN X N)
       (IF (LISTP X)
           (FN (CAR X) (FOLDR-FN (CDR X) N))
           N)).
```

We must apply the functional substitution to this axiom (producing a formula about the new symbols) and prove the result:

```
(EQUAL (FOLDR-TIMES X N)
       (IF (LISTP X)
           (TIMES (CAR X) (FOLDR-TIMES (CDR X) N))
           N)).
```

Observe that this is just the definition of `FOLDR-TIMES`, so the proof is immediate.[1] Had we not mapped `FN` to `TIMES` in our functional substitution, the formula above would call `FN` instead of `TIMES` and the result would have been unprovable. This is why the functional substitution must make explicit all the "ancestral" correspondences and not just those arising immediately in the theorem to be instantiated. By "ancestral" we mean to include all the functions reachable from the theorem to be instantiated by tracing back through definitions.

We characterized our proof obligation, above, as requiring the proof of the functional instance of every axiom about the old symbols. This can be weakened. Imagine that we had also used `FN` in some other definition, e.g., that of `MAP-FN`, not involved (ancestrally) in the theorem being instantiated. Since the axiom about `MAP-FN` mentions `FN`, one of the functions in our functional substitution,

[1] Note that it is not necessary that `FOLDR-TIMES` be syntactically analogous to `FOLDR-FN`, only that the functional instantiation of `FOLDR-FN` be provable.

a literal interpretation of our proof obligation would require us to instantiate the definition of `MAP-FN` and prove the result. But instantiating the definition of `MAP-FN` with the substitution above would change its call of `FN` to a call of `TIMES` and make no other changes. Clearly, the new equation would not be provable, since `MAP-FN` won't satisfy two different recurrence equations (one about `FN` and one about `TIMES`). To make such an instantiation provable we would have to define the analogue of `MAP-FN`, say `MAP-TIMES`, and include that pair in the substitution. We would then have to handle definitions involving `MAP-FN` since it is now in the substitution, etc. Fortunately, we show that we do not have to instantiate such irrelevant definitions; the program of extending the logic in the obvious way to accommodate such functions can be shown always to work.

One might be tempted to view the arguments presented in this example as involving "higher-order reasoning." In fact, our choice of the name `FUNCTIONALLY-INSTANTIATE` seems a deliberate provocation of that perception. Although we agree that such reasoning has a somewhat higher-order feel, we reiterate that functional instantiation is a derived rule of inference for first-order logic: any theorem that can be proved with `FUNCTIONALLY-INSTANTIATE` can be proved in first-order logic without it. No new higher-order axioms, no functional variables, no typed lambda calculus syntax have been added to the logic. We believe that a similar derived rule of inference could be attached to any first-order logic prover, e.g., a resolution based theorem-prover.

0.2.2 Sorting

We briefly present another example. In this example we define a "generic" insertion sort program that sorts according to an undefined ordering function `LT`. For the insertion sort to work (in the sense that it produce `LT`-ordered output), `LT` must be antisymmetric. No other properties of `LT` are required.

Therefore, it is feasible to constrain `LT` to be antisymmetric, define the sort function and the related sense of orderedness in terms of `LT` and prove that the sort function produces ordered output. This is attractive because it permits us to construct the proof of the sort program in isolation from the details of any particular ordering relation. In a mechanized setting, where details often overwhelm strategic heuristics, this offers more than mere mathematical elegance.

Below we reproduce the script necessary to justify the generic insertion sort. Observe that we can use a trivial witness for the introduction of `LT`.

```
(CONSTRAIN LT-INTRO (REWRITE)
  (IMPLIES (LT X Y)
           (NOT (LT Y X)))
  ((LT (LAMBDA (X Y) F))))
```

```
(DEFN ORDERED-LT (L)
  (IF (LISTP L)
      (IF (LISTP (CDR L))
          (IF (LT (CADR L) (CAR L))
              F
              (ORDERED-LT (CDR L)))
          T)
      T))

(DEFN INSERT-LT (X L)
  (IF (LISTP L)
      (IF (LT X (CAR L))
          (CONS X L)
          (CONS (CAR L) (INSERT-LT X (CDR L))))
      (LIST X)))

(DEFN SORT-LT (L)
  (IF (LISTP L)
      (INSERT-LT (CAR L) (SORT-LT (CDR L)))
      NIL))

(PROVE-LEMMA ORDERED-SORT-LT (REWRITE)
  (ORDERED-LT (SORT-LT L)))
```

The generic insertion sort routine is of no computational value – i.e., we cannot execute it – because `LT` is not defined.

But now suppose that in some application we need to sort lists of natural numbers according to the usual "less than" ordering on the naturals, `LESSP`. Because our logic is not higher-order, we have to define analogues of the functions above, only using `LESSP` in place of `LT`.

```
(DEFN ORDERED-LESSP
  (L)
  (IF (LISTP L)
      (IF (LISTP (CDR L))
          (IF (LESSP (CADR L) (CAR L))
              F
              (ORDERED-LESSP (CDR L)))
          T)
      T))
```

```
(DEFN INSERT-LESSP (X L)
  (IF (LISTP L)
      (IF (LESSP X (CAR L))
          (CONS X L)
          (CONS (CAR L) (INSERT-LESSP X (CDR L))))
      (LIST X)))

(DEFN SORT-LESSP (L)
  (IF (LISTP L)
      (INSERT-LESSP (CAR L) (SORT-LESSP (CDR L)))
      NIL))
```

While having virtually to repeat previous definitions may strike many readers as inelegant, three remarks can be made in its defense. First, generating the analogous definitions (with a good text editor such as Emacs[9]) is easy and represents almost no burden on the user. Second, keeping the logic first-order is extraordinarily valuable, especially in light of the desire to mechanize it. Finally, thanks to the new functional instantiation derived rule of inference, the properties of these new functions can be deduced at minimal cost from the theorems about the old functions. In particular, we can immediately conclude from the generic result, `ORDERED-SORT-LT`, that the particular program `SORT-LESSP` produces `ORDERED-LESSP` output, at the cost of establishing that `LESSP` is antisymmetric.

```
(FUNCTIONALLY-INSTANTIATE ORDERED-SORT-LESSP (REWRITE)
  (ORDERED-LESSP (SORT-LESSP L))
  ORDERED-SORT-LT
  ((LT         LESSP)
   (ORDERED-LT ORDERED-LESSP)
   (INSERT-LT  INSERT-LESSP)
   (SORT-LT    SORT-LESSP)))
```

0.3 Precise Description of the Derived Rules of Inference

In this section we give a precise description of two new derived rules of inference. This discussion is analogous to that in Chapter 4 of [1], "A Precise Description of the Logic," in which we describe the logic precisely without regard for its mechanization. In the next section we document the mechanization of the new rules, in a discussion analogous to Chapter 12 of [1], "Reference Guide." Our mechanization of functional instantiation is somewhat more efficient than its formal description would suggest. After presenting the logical description and the reference guide material, we derive the new rules and show that our mechanization is correct.

Henceforth the reader is assumed to be familiar with the logic of NQTHM described in [1], especially with the concept of "proof" described there. Roughly speaking, that notion of proof consists of a standard first-order logic [8] minus the rules concerning quantification but plus new principles of recursive definition and induction. Ignoring the infrequently used axioms about the function `V&C$`, this logic is very constructive, even computational, and is similar in power to that of [4].

Definition. A *functional substitution* is a function on a finite set of function symbols such that for each pair <f_1,f_2> in the substitution, either (a) f_2 is also a symbol and the arity of f_1 is the arity of f_2 or (b) f_2 has the form `(LAMBDA` (a_1 ... a_n) term`)` where the a_i are distinct variables, the arity of f_1 is n, and term is a term.

Definition. We recursively define the *functional instantiation of a term* t *under a functional substitution* fs. If t is a variable, the result is t. If t is the term (f t_1 ... t_n), let t_i' be the functional instantiation of t_i, for i from 1 to n inclusive, under fs. If, for some function symbol f′, the pair <f, f′> is in fs, the result is (f′ t_1' ... t_n'). If a pair <f, `(LAMBDA` (a_1 ... a_n) term`)`> is in fs, the result is term/{..., <a_i,t_i'>, ...}. Otherwise, the result is (f t_1' ... t_n').

Note. Recall from [1] that "term/σ" denotes the result of applying the ordinary (variable) substitution σ to term. If σ is the variable substitution {`<X, (FN A)>`, `<Y, B>`}, then `(PLUS X Y)`/σ is `(PLUS (FN A) B)`.

Example. The functional instantiation of the term

```
(PLUS (FN X) (TIMES Y Z))
```

under the functional substitution

```
{<PLUS, DIFFERENCE>, <FN, (LAMBDA (V) (QUOTIENT V A))>}
```

is the term

```
(DIFFERENCE (QUOTIENT X A) (TIMES Y Z)).
```

Definition. We recursively define the *functional instantiation of a formula* ϕ *under a functional substitution* fs. If ϕ is $\phi_1 \vee \phi_2$, then the result is $\phi_1' \vee \phi_2'$, where ϕ_1' and ϕ_2' are the functional instantiations of ϕ_1 and ϕ_2 under fs. If ϕ is $\neg\phi_1$, then the result is $\neg\phi_1'$, where ϕ_1' is the functional instantiation of

ϕ_1 under fs. If ϕ is x = y, then the result is x′ = y′, where x′ and y′ are the functional instantiations of x and y under fs.

Definition. A variable v is said to be *free* in `(LAMBDA (`a_1 ... a_n`) term)` if and only if v is a variable of term and v is not among the a_i. A variable v is said to be free in a functional substitution if and only if it is free in a `LAMBDA` expression in the range of the substitution. A variable v is said to be *bound* in `(LAMBDA (`a_1 ... a_n`) term)` if and only if v is among the a_i.

Definition. The aspects of a `LAMBDA` expression. A `LAMBDA` expression is a triple of the form `(LAMBDA (`a_1 ... a_n`) body)`. For such a `LAMBDA` expression, we say its *arity* is n, its *argument list* is `(`a_1 ... a_n`)`, and its *body* is body.

Notation. We denote functional instantiation with \ to distinguish it from ordinary (variable) substitution, which is denoted with /.

Example. If ρ is the functional substitution {`<PLUS, (LAMBDA (U V) (ADD1 U))>`} then `(PLUS X Y)\`ρ is `(ADD1 X)`.

Derived Rule of Inference. Conservatively constraining new function symbols.

It is permissible to add the term ax as an axiom to extend a history h provided there exists a functional substitution fs such that

1. the domain of fs is a set of new function symbols,
2. each member of the range of fs is either an old function symbol or is a `LAMBDA` expression whose body is formed of variables and old function symbols,
3. no variable is free in any `LAMBDA` expression in the range of fs, and
4. ax\fs is a theorem of h.

Definition. A functional substitution fs is *tolerable* with respect to a history h provided that the domain of fs contains only function symbols introduced into h by the user on top of the `GROUND-ZERO` logic, via `CONSTRAIN`, `DCL`, or `DEFN`, but not `ADD-SHELL`.

Note. We do not want to consider functionally substituting for built-in function symbols or shell function symbols because the axioms about them are so diffuse in the implementation. We especially do not want to consider substituting for such function symbols as `ORD-LESSP`, because they are used in the principle of induction.

Derived Rule Of Inference. Functional Instantiation.

If h is a history, fs is a tolerable functional substitution, p is a proof of thm in h, no free variable of fs occurs in p, and the fs instance of every axiom of h can be proved in h, then thm\fs can be proved in h.

0.4 Reference Guide

0.4.1 CONSTRAIN

General Form:

```
(CONSTRAIN name types ax
           (...  (new_i old_i) ...)  &OPTIONAL hints)
```

Example Form:

```
(CONSTRAIN H-COMMUTATIVITY2 (REWRITE)
           (EQUAL (H X (H Y Z))
                  (H Y (H X Z)))
           ((H PLUS))
           ((USE (PLUS-COMMUTATIVITY2))))
```

`CONSTRAIN` creates a new event. It does not evaluate its arguments. `CONSTRAIN` checks that <name, ..., new_i, ... > is a sequence of distinct new names; that each old_i is an old function symbol or a `LAMBDA` expression in old function symbols, without free variables, but with the same arity as new_i; that types is a legitimate set of types for storing lemmas; that ax is a formula; and that ax\{..., <new_i,old_i>, ...} is a theorem of the current history. The result of a `CONSTRAIN` is to add ax as an axiom according to the types and to declare the arity of each new_i to be that of old_i.

Note. We sometimes refer to the "old_i" used in a `CONSTRAIN` event as *witnesses*.

Examples. In the Example Form shown above we introduce a dyadic function `H` that has what we call the "commutativity2" property, namely, `(H X (H Y Z))` = `(H Y (H X Z))`. We use the Peano `PLUS` function as the witness. The `USE` hint supplied to `CONSTRAIN` says that the proof that the witness satisfies the commutativity2 property follows from the previously proved lemma `PLUS-COMMUTATIVITY2`. On page 27 we show how such a constrained `H` might be used to state and use the fact that to apply such a function iteratively to the elements of a list one may proceed either "outside in" or "inside out."

Below we use `CONSTRAIN` to introduce three functions, `P`, `Q`, and `R`, each of one argument. The functions are unconstrained, i.e., the constraining axiom added is `T`. We use the identity function as the witness for each.

```
(CONSTRAIN P-Q-R-INTRO (REWRITE) T
  ((P (LAMBDA (X) X))
   (Q (LAMBDA (X) X))
   (R (LAMBDA (X) X))))
```

0.4.2 FUNCTIONALLY-INSTANTIATE

General Form:
`(FUNCTIONALLY-INSTANTIATE` name types term old-name fs `&OPTIONAL` hints)

Example Form:
```
(FUNCTIONALLY-INSTANTIATE PR-TIMES-IS-AC-TIMES (REWRITE)
  (EQUAL (AC-TIMES L Z) (PR-TIMES L Z))
  PR-IS-AC
  ((H TIMES)
   (PR-H PR-TIMES)
   (AC-H (LAMBDA (X Y) (AC-TIMES X Y)))))
```

`FUNCTIONALLY-INSTANTIATE` is like `PROVE-LEMMA` in that it proves a theorem, term, and adds it to the database as a lemma with name name and types types. `FUNCTIONALLY-INSTANTIATE` requires that fs be a tolerable functional substitution, that old-name be a symbol that names some previously added event, and that term be the result of functionally instantiating the `FORMULA-OF` old-name with fs. (If term is the symbol `*AUTO*`, then term is automatically arranged to be just this instantiation.) To succeed `FUNCTIONALLY-INSTANTIATE` must prove the conjunction of instances under fs of some of the `DEFN`s, `CONSTRAIN`s, and `ADD-AXIOM`s, for which proof attempt the hints are used. The formulas that must be proved are the fs instantiations of each user `DEFN`, `CONSTRAIN`, and `ADD-AXIOM` that (a) uses as a function symbol some symbol in the domain of fs and (b) is either (i) an `ADD-AXIOM` or (ii) a `DEFN` or `CONSTRAIN` that introduces a function symbol ancestral[2] in the `FORMULA-OF` old-name or some `ADD-AXIOM`.

We wish to make it convenient to apply the same functional substitution to several different theorems in a sequence of `FUNCTIONALLY-INSTANTIATE` events, without having to prove the same constraints repeatedly. Therefore, `FUNCTIONALLY-INSTANTIATE` does not bother to prove ax\fs if any previous `FUNCTIONALLY-INSTANTIATE` did prove it. If you would like to limit the set of previous `FUNCTIONALLY-INSTANTIATE` events considered to some particular set {ev1, ..., evn}, then use (old-name ev1 ... evn) for old-name.

`FUNCTIONALLY-INSTANTIATE` aborts if any of the `DEFN`, `CONSTRAIN`, or `ADD-AXIOM` formulas to be instantiated and proved uses as a variable any variable that is free in fs. Such an abort can always be avoided by choosing new variable names.

Note. Observe that the mechanization of `FUNCTIONALLY-INSTANTIATE` does not require that we prove the fs instantiation of every axiom.

---
[2] The concept "ancestral" is defined in the next section, on p. 20.

0.5 Correctness

0.5.1 The Conservative Nature of Constrain

Suppose we embed our theory into a traditional first order logic, such as that of [8], turning the induction principle into a collection of axioms, admitting existential quantifiers and the existential-quantifier introduction-rule. Then the introduction of constrained functions, as defined above, results in a conservative extension of the previous theory. Proof. Suppose that we can satisfy the conditions for adding the constraint, ax, to a theory T with the functional substitution {..., <new$_i$,old$_i$>, ...}. Extend the current theory T to a new theory T$'$ by adding definitions (Shoenfield style definitions: no `SUBRP` axioms) that equate each new$_i$ with the corresponding old$_i$. Of course, T$'$ is a conservative extension of T since definition is proved by Shoenfield to produce conservative extensions. Note that ax is a theorem in T$'$ because we have assumed that ax\{..., <new$_i$,old$_i$>, ...} can be proved in T. Form T$''$ from T$'$ by throwing out the definitions of new$_i$ but adding ax as an axiom. T$''$ is a conservative extension of T because it is an extension of T and because any formula of T that can be proved in T$''$ can be proved in T$'$ and hence in T. Q.E.D.

Although NQTHM's definitional principle is not, strictly speaking, conservative (because of the `SUBRP` axioms), an NQTHM definitional extension of a theory T is a conservative extension of the extension of T produced by adding the `SUBRP` axioms. Also, an NQTHM definitional extension has what Shoenfield calls a translation property: If a formula is provable in the new theory, there is a syntactically very similar theorem provable in the old theory, extended by the `SUBRP` axioms, which "says the same thing" as the formula.

0.5.2 Functional Instantiation as a Derived Rule

We prove that functional instantiation is a correct derived rule of inference. We enter into great, nay tedious, detail to show that functional instantiation is correct for the actual logic of NQTHM, with its constructive character, and not merely for first-order logic, which is non-constructive. A proof for first-order logic alone would be somewhat shorter, since handling the quantifier axioms would require less work than handling our induction and definition principles. Essentially, the proof is inductive, showing that if we have proved a theorem thm and we wish to instantiate thm with a functional substitution fs, then we can, in effect, instantiate the entire proof of thm with fs to obtain a proof of the desired instance of thm. Because the inductive proof would otherwise be a little long, we first break out a few important, simple lemmas about the standard rules of inference.

Propositional axiom lemma. The functional instantiation of every propositional axiom is an axiom. Proof. $(\phi \lor \neg\phi)\backslash\mathrm{fs} = (\phi\backslash\mathrm{fs} \lor \neg(\phi\backslash\mathrm{fs}))$, which is a propositional axiom itself. Q.E.D.

Equality axiom lemma. If fs is a functional substitution, and eq is an equality axiom, then eq\fs is a theorem. Proof. Suppose the axiom is $x_1 = y_1 \land \ldots \land x_n = y_n \rightarrow (f\ x_1\ \ldots\ x_n) = (f\ y_1\ \ldots\ y_n)$. If f is not in the domain of fs, the instantiation does not change eq. If fs replaces f with function symbol f′, we note that the instance is another equality axiom, about f′. If fs replaces f with `(LAMBDA` $(a_1\ \ldots\ a_n)$ term$)$, then the instance is $x_1 = y_1 \land \ldots \land x_n = y_n \rightarrow \mathrm{term}/\{\ldots, <a_i,x_i>, \ldots\} = \mathrm{term}/\{\ldots, <a_i,y_i>, \ldots\}$. We now prove that for all terms, x_i, y_i, and a_i, that $x_1 = y_1 \land \ldots \land x_n = y_n \rightarrow \mathrm{term}/\{\ldots, <a_i,x_i>, \ldots\} = \mathrm{term}/\{\ldots, <a_i,y_i>, \ldots)\}$ is a theorem by induction on term. If term is a variable then we consider the cases. If term is one of the a_i, then the theorem in question has the form $x_1 = y_1 \land \ldots \land x_n = y_n \rightarrow x_i = y_i$, which is a tautology. If term is not one of the a_i, then the theorem in question has the form $x_1 = y_1 \land \ldots \land x_n = y_n \rightarrow \mathrm{term} = \mathrm{term}$, which follows from $x = x$. If term is not a variable, suppose it is $(f\ t_1\ \ldots\ t_n)$. By induction, we have $x_1 = y_1 \land \ldots \land x_n = y_n \rightarrow t_i/\{\ldots, <a_i,x_i>, \ldots\} = t_i/\{\ldots, <a_i,y_i>, \ldots\}$. Q.E.D.

Propositional Rule of Inference Lemmas. The propositional rules "commute" with functional instantiation. We show for each of the following four rules of inference that if α is a consequence of β (and perhaps γ), then $\alpha\backslash\mathrm{fs}$ is a consequence of $\beta\backslash\mathrm{fs}$ (and $\gamma\backslash\mathrm{fs}$).

> **Expansion.** $\beta \lor \alpha$ follows from α.
>
> $(\beta \lor \alpha)\backslash\mathrm{fs}$ follows from $\alpha\backslash\mathrm{fs}$ because $(\beta \lor \alpha)\backslash\mathrm{fs}$ is $\beta\backslash\mathrm{fs} \lor \alpha\backslash\mathrm{fs}$, which follows from $\alpha\backslash\mathrm{fs}$ by expansion.
>
> **Contraction.** α follows from $\alpha \lor \alpha$.
>
> $\alpha\backslash\mathrm{fs}$ follows from $(\alpha \lor \alpha)\backslash\mathrm{fs}$ because $(\alpha \lor \alpha)\backslash\mathrm{fs}$ is $\alpha\backslash\mathrm{fs} \lor \alpha\backslash\mathrm{fs}$.
>
> **Associativity.** $\alpha \lor \beta \lor \gamma$ follows from $(\alpha \lor \beta) \lor \gamma$.
>
> $(\alpha \lor \beta \lor \gamma)\backslash\mathrm{fs}$ follows from $((\alpha \lor \beta) \lor \gamma)\backslash\mathrm{fs}$ because $(\alpha\backslash\mathrm{fs} \lor \beta\backslash\mathrm{fs} \lor \gamma\backslash\mathrm{fs})$ follows from $((\alpha\backslash\mathrm{fs} \lor \beta\backslash\mathrm{fs}) \lor \gamma\backslash\mathrm{fs})$.
>
> **Cut.** $\beta \lor \gamma$ follows from $\alpha \lor \beta$ and $\neg\alpha \lor \gamma$.
>
> $(\beta \lor \gamma)\backslash\mathrm{fs}$ follows from $(\alpha \lor \beta)\backslash\mathrm{fs}$ and $(\neg\alpha \lor \gamma)\backslash\mathrm{fs}$ because $(\beta\backslash\mathrm{fs} \lor \gamma\backslash\mathrm{fs})$ follows from $(\alpha\backslash\mathrm{fs} \lor \beta\backslash\mathrm{fs})$ and $(\neg(\alpha\backslash\mathrm{fs}) \lor \gamma\backslash\mathrm{fs})$.

Notational Convention. a\b/c means (a\b)/c, and a/b\c means (a/b)\c.

Definition. Suppose that s is a substitution and fs is a functional substitution. Then fs%s is defined to be {<x,y\fs >: <x,y> is a member of s}. In other words, to obtain fs%s, we apply fs to each element of the range of s.

Commutativity Lemma. If t is a term, fs is a functional substitution, s is a substitution, and no variable free in fs occurs in the domain of s, then t\fs/(fs%s) = t/s\fs. Proof by induction on the structure of t. If t is a variable and for some v, <t,v> is in s, then t\fs/(fs%s) = t/(fs%s) = v\fs = t/s\fs; if t is a variable not in the domain of s, then t\fs/(fs%s) = t = t/s\fs. If t is not a variable but has the form (f ... t_i ...) and f is not in the domain of fs or is mapped by fs to a function symbol, then t\fs/(fs%s) = (f′ ... t_i\fs/(fs%s) ...) = (f′ ... t_i/s\fs ...) = t/s\fs, by induction, where f′ is f or its image under fs. Finally, if f is mapped to `(LAMBDA` (a_1 ... a_n) term) by fs, then t\fs/(fs%s) = (term/{..., <a_i, t_i\fs>, ...})/(fs%s) = term/{..., <a_i, t_i\fs/(fs%s)>, ...} because no free variable of `(LAMBDA` (a_1 ... a_n) term) is in the domain of fs%s since no such variable is in the domain of s; but term/{..., <a_i, t_i\fs/(fs%s)>, ...} = term/{..., <a_i, t_i/s\fs>, ...} by induction, and term/{..., <a_i, t_i/s\fs>, ...} = t/s\fs. Q.E.D.

Instantiation Rule of Inference Lemma. If t is a term, fs is a functional substitution, s is an ordinary substitution, no variable in the domain of s is free in fs, and t\fs is a theorem, then so is t/s\fs. Proof. t/s\fs = t\fs/(fs%s) by the Commutativity Lemma. Hence t/s\fs is a theorem by instantiation of t\fs with (fs%s). Q.E.D.

Justification of Functional Instantiation.
Suppose
h is a history,
fs is a tolerable functional substitution,
p is a proof of thm with respect to h,
no variable free in fs occurs in p, and
for each axiom ax that results from a user **DEFN**, **ADD-AXIOM**, or **CONSTRAIN**, ax\fs is a theorem of h.

Then thm\fs is a theorem of h.

Proof by induction on the length of p.

Base Case. If the length is 1, then thm must be an axiom of h. If ax is a propositional or equality axiom, then thm\fs is also an axiom, as proved above. If thm is another sort of **GROUND-ZERO** axiom or the result of a user shell invocation, it mentions no function symbol in the domain of fs by the hypothesis that fs is tolerable, and hence thm\fs = thm. If thm is any other user axiom, it must come from an **ADD-AXIOM**, **CONSTRAIN**, **DCL**, or **DEFN**. **DCL** adds no axiom, and in the other three cases, thm\fs is a theorem of h by hypothesis.

Induction Step. Suppose that the theorem holds when the length of p is k or less and suppose thm has a proof of length k+1. The rules of inference are the propositional rules, instantiation, and induction. The Propositional and Instantiation Rule of Inference Lemmas handle everything except induction. Note that the hypothesis that no variable free in fs occurs in p yields the necessary condition for the Instantiation Rule of Inference that no variable in the domain of an ordinary substitution used is free in fs.

Suppose then that thm has been proved by induction. We check the conditions for the inductive proof of thm\fs, inside square brackets, as we walk through the conditions that were checked in the proof of thm.

thm is a term [but so is thm\fs]

m is a term [we will use m\fs]

q_1 ... q_n are terms [we will use q_1\fs ... q_n\fs, which are terms]

h_1 ... h_n are positive integers [same]

it is a theorem that (**ORDINALP** m)

[we need to check that (**ORDINALP** m\fs) but this follows by induction, provided we have defined "proof" so that inductive proofs have such theorems as parts of them]

for $1 \leq i \leq k$ and $1 \leq j \leq h_i$, $s_{i,j}$ is a substitution

[we will use the fs%$s_{i,j}$ as our new substitutions; note that because no variable free in fs occurs in p, the Commutativity Lemma can be applied to show that for any t, t/$s_{i,j}$\fs = t\fs/fs%$s_{i,j}$. This depends on the somewhat peculiar, Shankar[10] style definition of "proof" in such a way that substitutions are explicitly embedded in proofs.]

it is a theorem that

(**IMPLIES** q_i (**ORD-LESSP** m/$s_{i,j}$ m))

[we need to check that

(**IMPLIES** q_i\fs (**ORD-LESSP** m\fs/(fs%$s_{i,j}$) m\fs))

but this is

(**IMPLIES** q_i (**ORD-LESSP** m/$s_{i,j}$ m))\fs

by the Commutativity Lemma and we have

(**IMPLIES** q_i (**ORD-LESSP** m/$s_{i,j}$ m))\fs

by induction since the proof of

(IMPLIES q_i (ORD-LESSP $m/s_{i,j}$ m))

is part of the proof of thm and hence has a length less than that of p. Note: the fact that fs passes through calls to functions such as ORD-LESSP, IMPLIES, etc., follows from the fact that fs is tolerable.]

Then thm is a theorem [i.e. thm\fs will be a theorem] if

(IMPLIES (AND ... (NOT q_i) ...) thm)

is a theorem
[we need to check that

(IMPLIES (AND ... (NOT q_i\fs) ...) thm\fs)

is a theorem, but this follows by induction]

and for $1 \leq i \leq k$,

(IMPLIES (AND q_i thm/$s_{i,1}$ ... thm/s_{i,h_i}) thm)

[we need to check that

(IMPLIES (AND q_i\fs thm\fs/(fs%$s_{i,1}$) ... thm\fs/(fs%s_{i,h_i}))
thm\fs)

is a theorem but this, by the Commutativity Lemma, is the same as

(IMPLIES (AND q_i\fs thm/$s_{i,1}$\fs ... thm/s_{i,h_i}\fs) thm\fs)

which is the same as

(IMPLIES (AND q_i thm/$s_{i,1}$ ... thm/s_{i,h_i}) thm)\fs

which follows by induction.]

Q.E.D.

Note. The theorem we have just proved can be strengthened by weakening the hypothesis

for each axiom ax that results from a user DEFN, ADD-AXIOM, or CONSTRAIN, ax\fs is a theorem of h.

to

for each axiom ax that
(a) results from a user `DEFN`, `ADD-AXIOM`, or `CONSTRAIN`,
(b) uses some member of the domain of fs as a function symbol, and
(c) is not one of the `SUBRP` axioms added by `DEFN`,
ax\fs is a theorem of h.

because in each case in which we no longer bother proving that ax\fs is a theorem, it is the case that ax\fs = ax. In particular, it is the case that the function symbols in the `SUBRP` axioms added by a `DEFN` (e.g., `SUBRP`, `FORMALS`, and `BODY`) are not permitted in the domain of tolerable functional substitutions.

It is now our intention to develop a somewhat less obvious but more important strengthening of functional instantiation, a strengthening that permits us to ignore instantiating and proving "irrelevant" definitions. Let us say, roughly, that a definition of a function fn in a history h is irrelevant to a theorem thm of h provided that (a) fn is not involved in the statement of thm nor is any function whose definition uses fn, etc., and (b) fn is similarly not involved in any `ADD-AXIOM`. Intuitively speaking, if we (i) ignore the `SUBRP` axioms that are added when a `DEFN` occurs and (ii) we embed our logic into a standard first order logic, then it is not hard to see that we need not do functional instantiation and proof on irrelevant definitions when using functional instantiation, because the theorem can be proved in the history that results from dropping away the irrelevant definitions. However, for the actual logic of NQTHM, it is not possible to ignore the `SUBRP` axioms. Furthermore, we are interested in a constructive proof of the legitimacy of ignoring irrelevant definitions, a proof that does not rely upon the presence of existential quantification in our logic. Therefore, we are about to embark upon a rather tedious proof that we can ignore irrelevant definitions provided we are content with knowing that a functional instantiation thm\fs of a theorem thm of a history h is at least a theorem of a definitional extension of h. In preparation for proving the Justification of Functional Instantiation with Extension Lemma, which permits us to ignore irrelevant definition, we first lay some groundwork.

Theorem. The generality of `LAMBDA`. Without loss of generality, we may assume that every element of the range of a functional substitution is a `LAMBDA` expression. Proof. Let fs = $\{..., \langle x_i, y_i \rangle, ...\}$. Let fs′ be the functional substitution obtained by replacing each y_i in the range of fs′ that is a function symbol with `(LAMBDA (`a_1 ... a_n`) (`y_i a_1 ... a_n`))`, where the a_j are distinct variables and n is the arity of y_i. We now prove by induction on the structure of the term t that t\fs = t\fs′. If t is a variable, both sides are t. So suppose t = (f ... t_i ...). If f is not in the domain of fs, then t\fs = (f ... t_i\fs ...), which, by induction is (f ... t_i\fs′ ...) = t\fs′. But if f is replaced with f′ by fs then t\fs = (f′ ... t_i\fs ...) = (f′ ... t_i\fs′ ...) = (f′ ... a_i ...)/$\{..., \langle a_i, t_i\backslash fs' \rangle, ...\}$ = t\fs′.

Q.E.D.

Definition. The *composition* of two functional substitutions fs_1 and fs_2, denoted fs_1:fs_2, is defined as follows, provided that no free variable of fs_2 occurs bound in fs_1. Without loss of generality, assume that each member of the range of fs_1 is a `LAMBDA` expression. Let fs_1 = {..., <x_i, `(LAMBDA` (... a_j ...) $term_i$`)`>, ...}. Let fs_2' be the restriction of fs_2 to the complement of the domain of fs_1. Then fs_1:fs_2 = {..., <x_i, `(LAMBDA` (... a_j ...) $term_i$\fs_2`)`>, ...}∪fs_2'. (This is strictly analogous to the composition of ordinary substitutions.)

Theorem. The composition of functional substitutions. If no free variable of fs_2 occurs bound in fs_1, then t\$fs_1$\$fs_2$ = t\(fs_1:fs_2). Proof by induction on the structure of t. If t is a variable, then both sides equal t. If t has the form (f ... t_i ...), assume inductively that t_i\$fs_1$\$fs_2$ = t_i\(fs_1:fs_2).

Suppose that <f, `(LAMBDA` (... a_i ...) fterm`)`> is a member of fs_1. Then

t\$fs_1$\$fs_2$
  by the definition of functional substitution
= (fterm/{..., <a_i, t_i\$fs_1$>, ...})\$fs_2$
  by the Commutativity Lemma, checking that no a_i is free in fs_2
= (fterm\$fs_2$)/($fs_2$`%`{..., <$a_i$, t_i\$fs_1$>, ...})
  by the definition of `%`
= (fterm\$fs_2$)/{..., <$a_i$, t_i\$fs_1$\$fs_2$>, ...})
  by induction
= (fterm\fs_2)/{..., <a_i, t_i\(fs_1:fs_2)>, ...}
= t\(fs_1:fs_2).

On the other hand, suppose that f is not a member of the domain of fs_1 but <f, `(LAMBDA` (... a_i ...) fterm`)`> is a member of fs_2. Then

t\$fs_1$\$fs_2$
= (f ... t_i\$fs_1$ ...)\$fs_2$
= fterm/{..., <a_i, t_i\$fs_1$\$fs_2$>, ...}
= fterm/{..., <a_i, t_i\(fs_1:fs_2)>, ...}
= t\(fs_1:fs_2)

Finally, if f is a member of neither the domain of fs_1 nor of fs_2,

t\$fs_1$\$fs_2$
= (f ... t_i\$fs_1$ ...)\$fs_2$
= (f ... t_i\$fs_1$\$fs_2$ ...)

= $t\backslash(fs_1{:}fs_2)$

Q.E.D.

Definition. A function symbol f_1 is an *ancestor* of a function symbol f_2 iff f_2 is introduced by a `DEFN` or `CONSTRAIN` and either f_1 is one of the function symbols introduced with f_2 (including f_2 itself) or f_1 is an ancestor of a symbol that is used as a function symbol in the axiom(s) added by the introduction of f_2.

Definition. A function symbol f is *ancestral* in a term t if and only if f is an ancestor of some symbol used as a function symbol in t.

Note. It is possible that when we do functional instantiation, we pick up new "governors." For example, if we consider the definitional equation

```
(F1 X Y)
   =
(IF (NLISTP X)
    NIL
    (G1 (F1 (CDR X) Y) Y)))
```

and consider the functional instantiation

{`<F1, F2>, <G1, (LAMBDA (X Y) (IF (G2 Y) X 3))>`}

the resulting instantiated equation is

```
(F2 X Y)
   =
(IF (NLISTP X)
    NIL
    (IF (G2 Y)
        (F2 (CDR X) Y)
        3))
```

Note that the recursive call of `F2` is now governed by the additional condition `(G2 Y)`. The crucial point for arguing the termination of the instantiated function is that we do not lose any governors.

Theorem. The Governor's Lemma. If fs is a functional substitution, f is not in the domain of fs, f* is not in fs or term nor equal to f, fs′ = fs∪{<f, `(LAMBDA` $(x_1 \ldots x_m)$ $(f^* x_1 \ldots x_n\ a_1 \ldots a_n)$`)>`}, and fs′ is tolerable, then the governors of an occurrence of a term o whose function symbol is f* in term\fs′ include the fs′ instances of the governors of a term n in term, with function symbol f, such that n\fs′ = o. Proof by induction on the size of term. If term is a variable, nothing governs anything. Case 1. Suppose term has the form `(IF` x y z). Consider occurrences of a term o in `(IF x y z)\fs′` with function symbol f*. Case 1.1. o is not `(IF x y z)\fs′` itself because fs′ is tolerable. Case 1.2. o occurs in the first argument of `(IF x y z)\fs′`. The governors of o here are the governors of o in x, so by induction those governors include the fs′ instances of the governors of a term n in x, with function symbol f, such that n\fs′ = o. Case 1.3. o occurs in the second argument of `(IF x y z)\fs′`. The one new governor that an occurrence of any term in y\fs′ obtains, when that occurrence is viewed as an occurrence in the second argument of `(IF x y z)\fs′`, is x\fs′, which is an fs′ instance of x. Case 1.4. Analogous to 1.3. Case 2. term has the form (f ... t_i ...), so term\fs′ has the form (f* ... t_i\fs′ ...). The governors of (f* ... t_i\fs′ ...) in (f* ... t_i\fs′ ...) is the empty set, which is the governors of t in t. The governors of an occurrence of term o with function symbol f* in a t_i\fs′ correspond to the governors of f terms in t_i by induction, and no new governors arise. Case 3. term has the form (g ... t_i ...), where g is not f. If g is not in the domain of fs′, then induction does the job. If <g, **(LAMBDA** (... a_i ...) gterm)> is a member of fs′, then term\fs′ = gterm\{..., <a_i, t_i\fs′>, ...}. Because f* does not occur in fs, it does not occur in gterm. Hence the only occurrences of o terms with function symbol f* we must consider are those in the t_i\fs′, which are covered by the induction hypothesis. They may pick up additional governors from gterm, but they do not lose any. Q.E.D.

We now show that certain functional substitutions can be extended to include **CONSTRAIN**s and **DEFN**s not supplied.

Definition. A functional substitution fs and a history h are said to be *extensible* provided that no variable free in fs occurs in h, and for each user **DEFN**, **ADD-AXIOM**, or **CONSTRAIN** axiom ax of h either (a) ax\fs is a theorem of h or (b) ax arises from a **DEFN** or **CONSTRAIN** and none of the function symbols there introduced are ancestors of any function symbol in the domain of fs or are ancestral in any **ADD-AXIOM** of h.

Convention. We take the attitude that embedded within a history h we have the proofs checking the acceptability of the **DEFN**s and **CONSTRAIN**s. We adopt this only to be able to obtain variables and function symbols not in those proofs.

Note. The concept introduced next, "obvious extension," is the key to the generation of the definitional extensions we will need in the proof of the Justification of Functional Instantiation with Extension Lemma. With obvious extensions, we can introduce for each irrelevant definition, (fn args) = body, another definition, very roughly (fn* args) = body/fs∪{<fn, fn*>), whose axiom will provide an automatic proof for the fs instance of fn, which we prefer not to consider.

Note. The following definition is several pages long because we prove the well-formedness of the definition as we present it.

Definition. The *obvious extensions* of an extensible functional substitution fs and history h are a new functional substitution fs′ and history h′ defined as follows.

If there is no axiom ax of h such that ax\fs is not a theorem of h, then fs′ is fs and h′ is h.

Otherwise, let ax be the first axiom in h such that ax\fs is not a theorem. Because fs is extensible, ax is a **DEFN** or **CONSTRAIN**, and the function symbols there introduced are not in the domain of fs. There are two cases to consider, depending upon whether the event in question is a **DEFN** or a **CONSTRAIN**.

DEFN Case. If the event is a **DEFN** of a function symbol f, then let f* be a function symbol new in h, one that is used in no **DEFN** or **CONSTRAIN** proofs of h. Let a_1, ..., a_n be the free variables of fs. Let the arity of f* be the arity of f plus n. Let a_1*, ..., a_n* be distinct variables not in fs or h. Let A be the substitution {..., <a_i,a_i*>, ...}. If the definitional axiom added for f is (f x_1 ... x_m) = body, then fs′ is fs∪{<f, (**LAMBDA** (x_1 ... x_m) (f* x_1 ... x_m a_1 ... a_n))>} and h′ is the history obtained by extending h with the definition (f* x_1 ... x_m a_1* ... a_n*) = body\fs′/A.

Before proceeding to the **CONSTRAIN** case we check that the definition added to h to build h′ is admissible. A new function symbol is being introduced, the argument list consists of distinct variables, and the new body is a term that mentions no variable not in the argument list. We now check condition (d) of the principle of definition. We review the argument that led to the introduction of f; the argument for f* is closely analogous, with the analogues noted in square brackets.

there is a term m [we take m\fs′/A]

such that (a) (**ORDINALP** m) can be proved directly in h

[we need to check that (**ORDINALP** m\fs′/A) can be be proved directly in h. Let p be the proof of (**ORDINALP** m) used to justify the introduction of f. Every axiom used in p is true under fs, from the definition of extensible because ax is the first axiom in h such that ax\fs is not a theorem. Because f was new at the time of its definition, the only axiom in p that could mention f as a function symbol would be the equality axiom for f. Hence for every axiom used in p, ax\fs′ is also a theorem. Hence by the Justification of Functional Substitution, (**ORDINALP** m)\fs′ is a theorem of h. That ax\fs′ can be proved "directly" follows from inspecting the proof constructed in the Justification Lemma, noting that no new functions are defined. (**ORDINALP** m\fs′/A) is an instance, indeed a variant, of (**ORDINALP** m\fs′).]

and (b) for each occurrence of a subterm of the form (f y_1 ... y_m) in body and the terms t_1 ... t_k governing it, the following formula can be proved directly in h:

```
(IMPLIES (AND t1 ... tk)
         (ORD-LESSP m/s m))
```

where s is the substitution {..., <x_i,y_i>, ...}.
[Observe that, by the Governor's Lemma, for each occurrence n of a subterm of the form (f* y_1' ... y_m' z_1 ... z_n) in the new body, there is an occurrence o of a subterm of the form (f y_1 ... y_m) in the original body such that $y_i' = y_i$\fs'/A, $z_i = a_i$*, and such that if g governs o in the old body, then g\fs'/A governs n in the new body.]

We need to check that for each subterm of the form (f* y_1' ... y_{m+n}') in body\fs'/A, and the terms v_i governing the occurrence, the following formula can be proved directly in h:

```
(IMPLIES (AND ... vi ...)
         (ORD-LESSP m\fs'/A/s' m\fs'/A))
```

where s' is the substitution {..., <x_i, y_i\fs'/A>, ...} (we can ignore the pairs <a_i*, a_i*>). We will prove the stronger theorem

```
(IMPLIES (AND t1\fs'/A ... tk\fs'/A)
         (ORD-LESSP m\fs'/A/s' m\fs'/A)),
```

which is stronger because each t_i\fs'/A is one of the v_i, by the Governor's Lemma.

Note that for all terms t, t/A/s' = t/(fs'%s)/A, so we need to prove

```
(IMPLIES (AND t1\fs'/A ... tk\fs'/A)
         (ORD-LESSP m\fs'/fs'%s/A m\fs'/A)),
```

which by the Commutativity Lemma is

```
(IMPLIES (AND t1\fs'/A ...tk\fs'/A)
         (ORD-LESSP m/s\fs'/A m\fs'/A)),
```

which by the definition of substitution is

```
(IMPLIES (AND t1 ...tk)
         (ORD-LESSP m/s m))\fs'/A,
```

which is a theorem by the Justification Lemma and instantiation because every axiom used in the proof of

```
(IMPLIES (AND t1 ... tk)
         (ORD-LESSP m/s m))
```

is a theorem under fs$'$.

CONSTRAIN Case. If the event was a **CONSTRAIN** with axiom ax and justifying functional substitution $\{\ldots, <f_i,$ **(LAMBDA** $(x_{i,1} \ldots x_{i,k_i})\ w_i)>, \ldots\}$, let $a_1, \ldots, a_n$ be the free variables of fs, let f_i* be distinct function symbols, new for h, that do not occur in fs, and that have the same arities as the f_i, plus n more arguments. Let $a_1*, \ldots, a_n*$ be distinct variables not in fs or h. Let A be the substitution $\{\ldots, <a_i,a_i*>, \ldots\}$. Then

fs$'$ =
fs$\cup\{\ldots, <f_i,$ **(LAMBDA**$(x_{i,1} \ldots x_{i,k_i})$
$(f_i* \ x_{i,1} \ldots x_{i,k_i}\ a_1 \ldots a_n))>, \ldots\}$,

and h$'$ is the extension of h with the **CONSTRAIN** event ax\fs$'$/A and justifying functional substitution $\{\ldots, <f_i*, w_i'>, \ldots\}$, where

w_i' = **(LAMBDA** $(x_{i,1} \ldots x_{i,k_i}\ a_1* \ldots a_n*)\ w_i$\fs$'$/A).

We now check that the **CONSTRAIN** event added to h to build h$'$ is admissible. Note that ax$\backslash\{\ldots, <f_i,$**(LAMBDA** $(x_{i,1} \ldots x_{i,k_i})\ w_i)>, \ldots\}$ was a theorem of h, checked when ax was added. Let p be the proof used in the introduction of ax. Because every axiom used in p has a proof in h under fs (since ax is the first axiom not a theorem under fs), we have that

ax$\backslash\{\ldots, <f_i,$**(LAMBDA** $(x_{i,1} \ldots x_{i,k_i})\ w_i)>, \ldots\}$\fs

is a theorem of h. But

ax\fs$'$/A$\backslash\{\ldots, <f_i*,w_i'>, \ldots\}$,

which is what we must prove to show the admissibility of the new **CONSTRAIN**, is, because no variable is free in a w_i', and by using the Commutativity Lemma,

$$ax\backslash fs'\backslash\{\ldots, <f_i*,w_i'>, \ldots\}/(\{\ldots, <f_i*, w_i'>, \ldots\}\%A),$$

which, by the definition of A, equals

$$ax\backslash fs'\backslash\{\ldots, <f_i*,w_i'>, \ldots\}/A,$$

which is, by the definition of fs′,

$$ax\backslash(fs\cup\{\ldots, <f_i, (\texttt{LAMBDA}\ (x_{i,1} \ldots x_{i,k_i})\ (f_i*\ x_{i,1} \ldots x_{i,k_i}\ a_1 \ldots a_n))>, \ldots\})\ \backslash\{\ldots, <f_i*,w_i'>, \ldots\}\ /A,$$

which, because no f_i* occurs in fs or ax, because no w_i' has a free variable bound in fs′, and because of the theorem on composition of functional substitutions is

$$ax\backslash(fs\cup\{\ldots, <f_i, (\texttt{LAMBDA}\ (x_{i,1} \ldots x_{i,k_i})\ (w_i\backslash fs'/A)/\{<x_{i,j}, x_{i,j}>, <a_i, a_i>\})>, \ldots\}/A,$$

which, because of the trivial substitutions, is

$$ax\backslash(fs\cup\{\ldots, <f_i, (\texttt{LAMBDA}\ (x_{i,1} \ldots x_{i,k_i})\ (w_i\backslash fs'/A))>, \ldots\}/A,$$

which, because f_i occurs in no w_i and a_i occurs in no w_i nor in fs due to the extensibility of fs, is

$$ax\backslash(fs\cup\{\ldots, <f_i, (\texttt{LAMBDA}\ (x_{i,1} \ldots x_{i,k_i})\ (w_i\backslash fs))>, \ldots\}/A,$$

which, by the composition of functional substitutions, and the observation that no a_i occurs in any w_i, is

$$ax\backslash\{\ldots, <f_i,(\texttt{LAMBDA}\ (x_{i,1} \ldots x_{i,k_i})\ w_i)>, \ldots\}\backslash fs/A$$

Hence what we must prove is only a variant of what we proved when we introduced the original `CONSTRAIN`.

End of the definition of obvious extensions.

Theorem. The obvious extensions of an extensible functional substitution and history are themselves extensible. Proof. To check that in both cases fs′ and h′ are extensible, we must show that if ax is any axiom of h′ such that ax\fs′ is not a theorem of h′, ax must arise from a `DEFN` or `CONSTRAIN` that introduces functions symbols, none of which is an ancestor of any function symbol in the domain of fs′ or ancestral in any `ADD-AXIOM` of h′. Suppose that ax is an axiom of h′ such that ax\fs′ is not a theorem of h′. Then ax is not the old `DEFN` or `CONSTRAIN` of h just analogized because ax\fs′ is a variant of the axiom just added to h to form h′ and hence is a now a theorem of h′ with a proof of length 2. If ax is the newly introduced `DEFN` or `CONSTRAIN` axiom, we note that no f* (the function or functions introduced there) is an ancestor of any function symbol in the domain of fs′ nor is it ancestral in any

`ADD-AXIOM` of h′. But ax is not an `ADD-AXIOM` because for every `ADD-AXIOM` axiom, ax_1, ax_1\fs = ax_1\fs′ since the function symbols just introduced and added to make fs′ are ancestral in no `ADD-AXIOM`. So ax must be introduced by a `DEFN` or `CONSTRAIN`, which occurs in h after the `DEFN` or `CONSTRAIN` just analogized. Because fs was extensible with respect to h, none of the function symbols g_1, ..., g_n introduced with ax is an ancestor of any function symbol in the domain of fs nor ancestral in an `ADD-AXIOM` of h; but none of the g_1, ..., g_n introduced with ax is then an ancestor of any function symbol of the domain of fs′, since function symbols introduced with later `DEFN`s and `CONSTRAIN`s cannot be ancestors of earlier ones. Furthermore, the g_1, ..., g_n introduced with ax are not ancestral in any `ADD-AXIOM`s of h′ because the `ADD-AXIOM`s of h′ are those of h. Q.E.D.

Lemma. Justification of Functional Instantiation with Extension.

Suppose
h is a history,
fs is a tolerable functional substitution,
p is a proof of thm with respect to h,
no variable free in fs occurs in p,
and <h, fs> is extensible, and, furthermore, for each `DEFN` or `CONSTRAIN` of h whose instance under fs is not a theorem of h, none of the function symbols introduced by the `DEFN` or `CONSTRAIN` is ancestral in thm.

Then thm\fs is a theorem in a `DEFN/CONSTRAIN` extension of h.

Proof. Because fs and h are extensible, we can keep obviously extending them to fs′ and h′ such that for each axiom ax used in p, ax\fs′ is a theorem of h′. Hence thm\fs′ will be a theorem of h′. But no function symbol ancestral in thm will be added to the domain of fs to form fs′, hence thm\fs = thm\fs′. Thus thm\fs is a theorem of h′, a `DEFN/CONSTRAIN` extension of h. Q.E.D.

Note. Justification of the Implementation of `FUNCTIONALLY-INSTANTIATE`. The implementation of the new event `FUNCTIONALLY-INSTANTIATE` differs in a minor way from that which is suggested by the Justification of Functional Instantiation with Extension Lemma, namely (i) we permit the free variables of fs to occur in thm and (ii) we cause a simple error if any of the free variables of fs occur in any of the `DEFN`, `CONSTRAIN`, or `ADD-AXIOM` axioms that need instantiation and proof. We now justify (i). If a user wishes to functionally instantiate a theorem thm with a functional substitution fs, let p be any proof of thm, let s be a 1:1 substitution that maps the free variables of fs to variables that occur nowhere in p or fs, and let fs′ be the set of pairs <f_1, f_2> such that either <f_1, f_2> occurs in fs and f_2 is a symbol or for some term and (a_1 ... a_n) it is the case that <f_1, `(LAMBDA` (a_1 ... a_n) term)> occurs in fs and f_2 = `(LAMBDA` (a_1 ... a_n) term)/s where `(LAMBDA` (a_1 ... a_n) term)/s is `(LAMBDA` (a_1 ... a_n) term/s_1) where s_1 is the result of removing from s every pair whose first element is one of the a_i. Because no variable free in fs′ occurs in p, thm\fs′ will be a theorem provided that we check that each relevant ax\fs′ is a theorem. Given thm\fs′, then, how do we derive the desired thm\fs ? The answer is that thm\fs = thm\fs′/s_2, where s_2 is the inverse of s.

0.6 Examples

0.6.1 Inside-Outside

Here we introduce a dyadic function `H` with the property we call "commutativity2," and show that one can "map" with such a function in a primitive recursive way (using `PR-H`, which computes "inside-out") or in an accumulator-using way (using `AC-H`, which computes "outside-in"), getting the same result either way. The equivalence of these two methods of applying `H` to a list is formally stated in `PR-IS-AC` below.

```
(CONSTRAIN INTRO-H
  (REWRITE)
  (EQUAL (H X (H Y Z))
         (H Y (H X Z)))
  ((H PLUS)))

(DEFN PR-H (L Z)
  (IF (NLISTP L)
      Z
      (H (CAR L) (PR-H (CDR L) Z))))

(DEFN AC-H (L Z)
  (IF (NLISTP L)
      Z
      (AC-H (CDR L) (H (CAR L) Z))))

(PROVE-LEMMA PR-IS-AC (REWRITE)
  (EQUAL (AC-H L Z) (PR-H L Z))
  ((INDUCT (AC-H L Z))))
```

Since `H` is unconstrained except for having the commutativity2 property, the intuitive force of `PR-IS-AC`, above, is that it should hold for any function with the commutativity2 property. More precisely, given any function with the commutativity2 property, the two analogues of `PR-H` and `AC-H` are equal. We show how, with functional instantiation, `PR-IS-AC` can be used to draw this conclusion about two analogous functions that map with `TIMES`.

```
(DEFN PR-TIMES (L Z)
  (IF (NLISTP L)
      Z
      (TIMES (CAR L) (PR-TIMES (CDR L) Z))))

(DEFN AC-TIMES (L Z)
  (IF (NLISTP L)
      Z
      (AC-TIMES (CDR L) (TIMES (CAR L) Z))))

(FUNCTIONALLY-INSTANTIATE PR-TIMES-IS-AC-TIMES (REWRITE)
  (EQUAL (AC-TIMES L Z) (PR-TIMES L Z))
  PR-IS-AC
  ((H TIMES)
   (PR-H PR-TIMES)
   (AC-H (LAMBDA (X Y) (AC-TIMES X Y)))))
```

0.6.2 Map-Append

We here define the familiar `MAP` function that collects the results of applying an arbitrary unary function `FN` to every element of a list. We show that `MAP` distributes over the list concatenation function, `APPEND`, and instantiate the result so that we map with a function that takes two arguments instead of one.

```
(CONSTRAIN FN-INTRO () T ((FN ADD1)))

(DEFN MAP-FN (X)
  (IF (NLISTP X)
      NIL
      (CONS (FN (CAR X)) (MAP-FN (CDR X)))))

(PROVE-LEMMA MAP-DISTRIBUTES-OVER-APPEND (REWRITE)
  (EQUAL (MAP-FN (APPEND U V))
         (APPEND (MAP-FN U) (MAP-FN V))))

(DEFN MAP-PLUS-Y (X Y)
```

```
  (IF (NLISTP X)
      NIL
      (CONS (PLUS (CAR X) Y) (MAP-PLUS-Y (CDR X) Y))))

(FUNCTIONALLY-INSTANTIATE MAP-PLUS-Y-DISTRIBUTES-OVER-APPEND (REWRITE)
  (EQUAL (MAP-PLUS-Y (APPEND U V) Z)
         (APPEND (MAP-PLUS-Y U Z) (MAP-PLUS-Y V Z)))
  MAP-DISTRIBUTES-OVER-APPEND
  ((FN (LAMBDA (X) (PLUS X Z)))
   (MAP-FN (LAMBDA (X) (MAP-PLUS-Y X Z)))))
```

0.6.3 Properties of the Generic Interpreter

NQTHM is often used to formalize programming languages or general computing systems. The typical formalization involves defining an interpreter for the language or system. This interpreter generally takes the form of a function of a "state" and a "clock" and repeatedly applies a "step" function to the state until "time" has run out.

For example, in an assembly-level language[6,7], the state might include a "program counter" and some "program space" and "data space." Often the notion of state is further refined to include just "good states," i.e., states whose components stand in certain invariant relations to one another e.g., the program counter points to a legal address in program space, program space contains well-formed instructions, etc. Stepping generally involves determining from the initial state some transformation to be performed. For example, if the program counter points to an `(ADD a b)` instruction in program space, then the step is to compute the sum of the contents of data locations `a` and `b`, store that into data location `a`, and increment the program counter by 1. For realistic languages, the formal definitions of "good state" and "step" often run to a hundred pages.

But if an interpreter is just the iterated application of a step function to an initial state, then many properties of the formal language can be proved without regard for the details. Below we introduce the generic notions of a "good state" and of "stepping" from one good state to another. Then we define the generic interpreter.

```
(CONSTRAIN STATEP-AND-STEP-INTRO (REWRITE)
  (IMPLIES (STATEP S) (STATEP (STEP S)))
  ((STEP (LAMBDA (X) X))
   (STATEP (LAMBDA (X) F))))
```

Observe that `STEP` is constrained to preserve `STATEP`.

```
(DEFN INTERP (S N)
  (IF (ZEROP N)
      S
      (INTERP (STEP S) (SUB1 N))))
```

We can then prove two important lemmas about this generic interpreter. The first is that if it is started in a good state then it ends in a good state:

```
(PROVE-LEMMA STATEP-INTERP (REWRITE)
  (IMPLIES (STATEP S) (STATEP (INTERP S N))))
```

The second is that to run it `I+J` steps starting from some state `S` is the same as running it `I` steps from `S` and then running it `J` steps more from there.

```
(PROVE-LEMMA SEQUENTIAL-INTERP (REWRITE)
  (EQUAL (INTERP S (PLUS I J))
         (INTERP (INTERP S I) J)))
```

If it is desired to conclude these facts about a particular, realistic interpreter, they can be inferred by functional instantiation at the cost only of proving that the constraint on `STATEP` and `STEP` is satisfied, i.e., proving that the realistic step function preserves realistic good states.

0.6.4 The Associativity of APPEND without Induction

Here we follow the lead of Goodstein in [4] and of McCarthy with his recursion induction [5]. We show, using `FUNCTIONALLY-INSTANTIATE`, that the associativity of `APPEND` can be proved without explicit appeal to induction. Of course there are inductions hidden all over the place, e.g., in the type-set analysis for `TRUE-REC` and in the proof of the metatheorem that justifies `FUNCTIONALLY-INSTANTIATE`. Still, this is a startling development to those who regard the associativity of `APPEND` as the first theorem requiring an inductive proof.

In this example we actually define and use the function `APP` in place of `APPEND`, which is predefined in NQTHM, so that the entire development is explicit.

```
(DEFN TRUE-REC (X)
  (IF (NLISTP X)
      T
      (TRUE-REC (CDR X))))

(PROVE-LEMMA TRUE-REC-IS-TRUE (REWRITE) (TRUE-REC X))
```

```
(DEFN APP (X Y)
  (IF (NLISTP X)
      Y
      (CONS (CAR X) (APP (CDR X) Y))))

(FUNCTIONALLY-INSTANTIATE ASSOC-OF-APP (REWRITE)
  (EQUAL (APP (APP X Y) Z) (APP X (APP Y Z)))
  TRUE-REC-IS-TRUE
  ((TRUE-REC (LAMBDA (X) (EQUAL (APP (APP X Y) Z) (APP X (APP Y Z)))))))
```

0.6.5 Faking Quantifiers

We next illustrate the use of `CONSTRAIN` and `FUNCTIONALLY-INSTANTIATE` in theorems that resemble proofs in first-order predicate calculus with quantifiers. We first introduce an unconstrained unary function `P`. We then constrain `(ALL-X-P-X)` so that its truth implies that `(P X)` is true for all `X`. We give the analogous constrained meaning to `(SOME-X-P-X)`. Then we prove that the former implies the latter.

```
(CONSTRAIN P-INTRO () T ((P LISTP)))

(CONSTRAIN ALL-X-P-X-INTRO (REWRITE)
  (IMPLIES (ALL-X-P-X) (P X))
  ((ALL-X-P-X FALSE)))

(CONSTRAIN SOME-X-P-X-INTRO (REWRITE)
  (IMPLIES (P X) (SOME-X-P-X))
  ((SOME-X-P-X TRUE)))

(PROVE-LEMMA ALL-IMPLIES-SOME ()
  (IMPLIES (ALL-X-P-X) (SOME-X-P-X))
  ((USE (ALL-X-P-X-INTRO))))
```

0.6.6 Fairness

We illustrate a `CONSTRAIN` that expresses that a function is "fair" in the sense that it is infinitely often true and false. This sort of constraint is used in Goldschlag's NQTHM formalization of Unity[3].

```
(DEFN EVEN (X)
```

```
  (IF (ZEROP X)
      T
      (IF (EQUAL X 1) F (NOT (EVEN (SUB1 X))))))

(CONSTRAIN FAIR-INTRO (REWRITE)
  (AND (FAIR (FAIR-TRUE-WITNESS N))
       (NOT (FAIR (FAIR-FALSE-WITNESS N)))
       (NOT (LESSP (FAIR-TRUE-WITNESS N) N))
       (NOT (LESSP (FAIR-FALSE-WITNESS N) N)))
  ((FAIR EVEN)
   (FAIR-TRUE-WITNESS (LAMBDA (X) (IF (EVEN X) X (ADD1 X))))
   (FAIR-FALSE-WITNESS (LAMBDA (X) (IF (EVEN X) (ADD1 X) X)))))
```

0.6.7 Tracking ADD-AXIOMs

In this section we illustrate a functional instantiation that fails. Suppose we constrain `P` to be an arbitrary unary function:

```
(CONSTRAIN P-INTRO (REWRITE) T ((P (LAMBDA (X) 0))))
```

Suppose we define the function `P-ALIAS` just to be another name for `P`.

```
(DEFN P-ALIAS (X) (P X))
```

but we then "constrain" `P-ALIAS` to be even by adding an axiom

```
(ADD-AXIOM EVEN-P-ALIAS (REWRITE)
  (EVEN (P-ALIAS X)))
```

This implicitly constrains `P` to be even, as we can now prove:

```
(PROVE-LEMMA EVEN-P (REWRITE) (EVEN (P X))
  ((USE (EVEN-P-ALIAS)))).
```

Now a certain flawed line of reasoning goes like this: `P` was introduced by an unconstrained `CONSTRAIN` and we have proved `P` to be even. Therefore, we ought to be able to conclude by functional instantiation that any function, e.g., `ADD1`, is even.

```
(FUNCTIONALLY-INSTANTIATE EVEN-ADD1 ()
  (EVEN (ADD1 X))
  EVEN-P
  ((P ADD1)))
```

Why doesn't this work? At first glance, one is tempted to say "you can't prove the functional instance of the `ADD-AXIOM EVEN-P-ALIAS`." But this is false, we can prove it: the functional substitution does not include `P-ALIAS` in its domain and so the instance of the axiom in question is just the axiom itself. So what proof obligation can't we establish?

Consider what the Reference Guide for `FUNCTIONALLY-INSTANTIATE` requries:

> The formulas that must be proved are the fs instantiations of each user `DEFN`, `CONSTRAIN`, and `ADD-AXIOM` that (a) uses as a function symbol some symbol in the domain of fs and (b) is either (i) an `ADD-AXIOM` or (ii) a `DEFN` or `CONSTRAIN` that introduces a function symbol ancestral in the `FORMULA-OF` old-name or some `ADD-AXIOM`.

Consider the `DEFN` of `P-ALIAS`. It is a `DEFN` that (a) uses a function symbol in the domain of our fs, namely `P`, and (b) introduces a function symbol, namely `P-ALIAS`, that is ancestral in some `ADD-AXIOM`, namely `EVEN-P-ALIAS`. Thus, the functional instance of the definition of `P-ALIAS` under the substitution {`<P, ADD1>`} must be proved. This produces the goal `(P-ALIAS X) = (ADD1 X)`, which is unprovable.

We could attempt to remedy this situation by providing an instantiation of `P-ALIAS`, namely, `ADD1`, in our functional substitution. But if we did that, the previously considered instance of the `ADD-AXIOM EVEN-P-ALIAS` would no longer be provable.

0.6.8 Tracking Free Variables

It is necessary for soundness that we check that the variables in the constraints do not intersect the free variables in the `FUNCTIONALLY-INSTANTIATE` substitutions. The example in this section illustrates this.

Suppose we constrain the constant function `Z` to be `0`, but we use the variable `X` in the constraint

```
(CONSTRAIN Z-INTRO (REWRITE)
  (IMPLIES (EQUAL X 0)
           (EQUAL (Z) X))
  ((Z (LAMBDA () 0))))
```

because we think we see how to compromise the system with free variable confusion.

We can prove that `Z` is `0`:

```
(PROVE-LEMMA Z-IS-0 (REWRITE) (EQUAL (Z) 0))
```

Now let us try to prove `(EQUAL X 0)`, i.e., everything is `0`, by functionally instantiating `(Z)` to be `X`, using the substitution {`<Z, (LAMBDA () X)>`}.

```
(FUNCTIONALLY-INSTANTIATE EVERY-X-IS-0 ()
  (EQUAL X 0)
  Z-IS-0
  ((Z (LAMBDA () X))))
```

The faulty reasoning goes as follows: To prove `EVERY-X-IS-0` we have to prove the functional instance of the constraint on `Z`, namely,

```
  (IMPLIES (EQUAL X 0)
           (EQUAL (Z) X))
```

under the functional substitution {`<Z, (LAMBDA () X)>`}. But that instance is the trivial:

```
  (IMPLIES (EQUAL X 0)
           (EQUAL X X)).
```

That reasoning is correct, as far as it goes. However, the Reference Guide for `FUNCTIONALLY-INSTANTIATE` goes on to say

> `FUNCTIONALLY-INSTANTIATE` aborts if any of the `DEFN`, `CONSTRAIN`, or `ADD-AXIOM` formulas to be instantiated and proved uses as a variable any variable that is free in fs. Such an abort can always be avoided by choosing new variable names.

Thus, the above `FUNCTIONALLY-INSTANTIATE` event is rejected. If we avoid the abort by choosing a different variable, e.g., `V`, we must prove a non-theorem, e.g.,

```
  (IMPLIES (EQUAL X 0)
           (EQUAL V X)),
```

which is equivalent to having to prove `(EQUAL V 0)`.

Acknowledgements

We thank Bill Bevier for suggestions about the design of the two new events. We thank Don Simon and others at Ross Overbeek's theorem-proving seminar at the University of Texas in the Spring of 1989 for finding and fixing some bugs in the proofs. Joe Goguen (private communication) has observed that the legitimacy of functional instantiation is a consequence of the "theorem on constants." In fact, the idea of functional instantiation is related to ideas in Clear[2]. We thank Dianne King for editorial assistance. And we thank the Defense Advanced Research Projects Agency, which has supported in part the research reported here. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the U.S. Government, or Computational Logic, Inc.

Bibliography

[1] Robert S. Boyer and J Strother Moore (1988): A Computational Logic Handbook. Academic Press.

[2] R. M. Burstall and J. A. Goguen (1981): An Informal Introduction to Specification using Clear. In: Robert S. Boyer and J Strother Moore, eds.: The Correctness Problem in Computer Science. Academic Press.

[3] D. Goldschlag (1990): Mechanizing Unity. In: M. Broy and C. B. Jones, eds.: Programming Concepts and Methods. North-Holland Publishing Co.

[4] R. L. Goodstein (1964): Recursive Number Theory. North-Holland Publishing Company.

[5] John McCarthy (1963): A Basis for a Mathematical Theory of Computation. In: P. Braffort and D. Hershberg, eds.: Computer Programming and Formal Systems. North-Holland Publishing Company.

[6] J S. Moore (1988): Piton: A Verified Assembly Level Language. Technical Report 22, Computational Logic, Inc., 1717 West Sixth Street, Suite 290, Austin, TX 78703.

[7] J S. Moore (1989): A Mechanically Verified Language Implementation. Journal of Automated Reasoning 5(4), 461–492.

[8] J. R. Shoenfield (1967): Mathematical Logic. Addison-Wesley.

[9] Richard M. Stallman (1987): GNU Emacs Manual, Sixth Edition, Version 18. Free Software Foundation, 1000 Massachusetts Avenue, Cambridge, MA 02138.

[10] N. Shankar (1986): Proof Checking Metamathematics. Ph. D. Thesis. University of Texas at Austin.

[11] D. A. Turner (1979): A New Implementation Technique for Applicative Languages. Software – Practice and Experience 9, 31–49.