Noname manuscript No.
(will be inserted by the editor)

Limited Second-Order Functionality in a First-Order Setting

Matt Kaufmann · J Strother Moore

the date of receipt and acceptance should be inserted later

Abstract We describe how we have defined in ACL2 a weak version of the Common Lisp functional `apply`, which takes a function and list of actuals and applies the function to the actuals. Our version, called `apply$`, does not operate on functions but on ordinary objects — symbols and lists representing lambda expressions – some of which are interpreted as functions. We define a syntactic notion of "tameness" to identify the interpretable objects. This makes our `apply$` weaker than a true second-order functional but we believe `apply$` is powerful enough for many uses in ACL2. To maintain soundness and the conservativity of our Definitional Principle we require that certain hypotheses, called "warrants", be present in any theorem relying on the behavior of `apply$` on non-primitives. Within these constraints we can define "functionals" such as `sum` and `foldr` which map tame "functions" over lists and accumulate the results. This allows the ACL2 user to avoid defining specialized recursive functions for each such application. We can prove and use general-purpose lemmas about these "functionals." We describe the formalization, explain how we keep the Definitional Principle conservative, show examples of useful functions using `apply$` and theorems about them, sketch the proof that there is a model of any extension of the system using the new primitives, discuss issues arising in making these functions executable, and show some preliminary performance results.

Keywords ACL2, theorem proving, apply, higher-order logic, functionals

1 About Lisp and ACL2

ACL2 is an untyped, first-order logic with induction up to $\epsilon_0 = \omega^{\omega^{\omega^{\cdots}}}$ and a conservative Definitional Principle [7] for recursive functions over an induc-

Department of Computer Science
University of Texas at Austin
E-mail: {kaufmann,moore}@cs.utexas.edu

tively constructed domain. It is also a programming language supported by an automatic theorem prover [20,21]. As a language, ACL2 extends a substantial applicative subset of ANSI standard Common Lisp [30,32].[1]

The Lisp programmer and the ACL2 user write `(`ψ α_1 ... α_n`)` where traditional mathematical notation would be $\psi(\alpha_1, \ldots, \alpha_n)$, denoting a call of the n-ary function ψ on the n values delivered by the "actual expressions" $\alpha_1, \ldots, \alpha_n$. Thus, `(mod x 2)` is how we write what might otherwise be written `mod`$(x, 2)$ or even x **mod** 2.

ACL2 provides numbers, characters, strings, symbols, and lists. For example `(LAMBDA (X) X)` is a list of length 3 whose elements are (i) the symbol `LAMBDA`, (ii) the list of length 1 whose only element is the symbol `X`, and (iii) the symbol `X`. Lists are represented by right-associated nests of ordered pairs terminating in `NIL`. The list `(1 2 3)` is the ordered pair $\langle$`1`, $\langle$`2`, $\langle$`3`,`NIL`$\rangle\rangle\rangle$.

The function `cons` constructs ordered pairs and the functions `car` and `cdr` return the left and right components respectively.

Any ACL2 object, x, can be written as a term whose value is that object by preceding the object with a quote mark, `'`x, or by writing `(quote` x`)`. So one way to write a term whose value is `(LAMBDA (X) X)` is `'(LAMBDA (X) X)`. Another way is `(cons 'LAMBDA '((X) X))`.

Newcomers to Lisp and ACL2 are sometimes confused by the mix of terms and values. For example, in Lisp, both `(mod x 2)` and `'(mod x 2)` are terms that can be evaluated (provided, in the former case, the variable `x` has a rational value). The former evaluates to a rational, in fact, to 0 or 1 if `x` is integral. The latter evaluates to a list of length 3 whose first element is the symbol `mod`, whose second element is the symbol `x` and whose third element is the number `2`.

ACL2 is case insensitive by default. So when we write symbols such as `mod` we could write variations like `Mod` or `MOD`.

In this paper when a symbol appears as a function symbol we use lowercase. We capitalize the symbol if it is the first symbol in a sentence. When a symbol appears as a variable symbol we use lowercase italics. And when a symbol appears inside a quoted constant we use uppercase. For example, a term denoting the concatenation of the list `(MON TUE WED)` onto the value of the variable x is `(append '(MON TUE WED)` x`)`. We could have equivalently written `(append '(mon tue wed) x)` but believe the former makes it clear that `append` is a function symbol, the symbols `MON`, `TUE`, and `WED` appear in a constant, and x is a variable.

ACL2 supports a Definitional Principle. To define a new function with name ψ, formals ν_1, ... ν_n, and body β one executes the form `(defun` ψ `(`$\nu_1, \ldots \nu_n$`)` β`)`.[2] To be admissible the definition must satisfy certain con-

[1] See [20] for a description of the logic and system. For installation instructions and many other resources, see the ACL2 home page [21]. Extensive web-based documentation is available there. In this paper we sometimes refer to documentation by writing "see :DOC x," which means "go to the ACL2 home page [21], click on The User's Manuals link, then click on the ACL2+Books Manual link and type x into the **Jump to** box."

[2] Optional `declare` forms may be placed before β but we ignore this for the moment.

straints including that there be some measure of the formals into the ordinals that can be proved to decrease in every recursive call. If admissible, the `defun` extends the logic by adding as an axiom the definitional equation $(\psi\ \nu_1 \ldots \nu_n) = \beta$. This extension is *conservative* in the following sense. Let Γ be the theory in which the `defun` is admitted. Let Γ' be Γ with the addition of the definitional equation above. Then if formula τ can be proved in Γ' and ψ is not ancestral in τ, then τ can be proved in Γ. We say a function is *ancestral* in a formula if the function is called in the formula, or is ancestral in the definitional equation of or constraints on a function called in the formula.

Conservativity of the Definitional Principle is important because it means the user can define concepts to structure a proof. For example to prove that $p \rightarrow q$ one might define some new concept r and then prove $p \rightarrow r$ and $r \rightarrow q$. Conservativity guarantees that $p \rightarrow q$ is a theorem in the original theory. New concepts like r are often needed in inductive proofs.

We are frequently interested in functions in which `apply$` is or is not ancestral. For ease of exposition, if `apply$` is ancestral in f then we say f is a *scion of* `apply$` or simply a *scion*. The dictionary defines a "scion" to be a descendant, especially a descendant of an influential family or person. Informally, if f is a scion then f is `apply$` or calls `apply$`, or calls a function that calls `apply$`, etc. Many scions would be called "functionals" in higher-order systems but we do not use that term since ACL2 is first-order. Furthermore, scions need not take "functions" as arguments. The function defined by `(defun silly (`x`) (apply$ 'CONS (list` x x`)))` is a scion but does not take a "function" as an argument.

If f is not a scion, i.e., `apply$` is not ancestral in f, then we say f is *independent of* `apply$`.

2 The Goal of This Work

Because ACL2 is first-order, functions are not objects in the ACL2 universe. Variables cannot take on functions as values and no function can take functions as arguments. There are no functionals in ACL2. Prior to this work, the user wishing to map over a list and sum the absolute values of the elements typically had to define a recursive function specialized for that purpose.

```
(defun sum-abs (lst)
  (if (endp lst)
      0
      (+ (abs (car lst)) (sum-abs (cdr lst)))))
```

If instead one wanted to sum the squares, a different function would be defined. But a very common construct in Common Lisp is the `loop` special form. Examples include

```
(loop for x in lst when (p x) sum (f x))
(loop for x in lst always (p x))
(loop for x from i to max by incr collect (f x))
```

We would like to give formal meaning to such expressions in ACL2. Our approach will be to formalize `apply$` as a "functional" that takes a "function" and a list of objects and applies the "function" to the objects. Intuitively `(apply$` f `(list` $a_1 \ldots a_n$`))` = `(`f $a_1 \ldots a_n$`)`.[3]

We put the words "functional" and "function" in quotes here because, in fact, `apply$` is not a functional: its first argument is not (and cannot be) a function; instead the "function" argument of `apply$` will be an ordinary ACL2 object, normally a symbol naming a function or a list expression representing a lambda expression. `Apply$` will give such objects meaning by interpreting them. But it is impossible to achieve this soundly for all definable functions in our first-order untyped logic. So our `apply$` is weak and can properly interpret only a subset of the "function" objects one might pass to it.

Given `apply$` we can define other "functionals." We might, for example, define `(sum` f lst`)` to sum the values of f over the elements of lst and `(filter` p lst`)` to collect those elements of lst satisfying p.

Then the first `loop` expression above could be expressed formally as `(sum 'f (filter 'p` lst`))`. Of course, this captures the semantics of the Common Lisp `loop` only if `f` and `p` fall into the subset of "functions" `apply$` handles properly.

The challenge is formalizing `apply$` so that for a useful class of functions, `(apply$` f `(list` a_1 ... a_n`))` is `(`f a_1 ... a_n`)` for each f in that class — while preserving the untyped, first-order nature of ACL2 and the conservativity of ACL2's Definitional Principle. Those aspects of ACL2 — an untyped language, a first-order logic, and a conservative Definitional Principle — are critical to many proof techniques in ACL2 not to mention its convenience, power, efficiency, and legacy library of verified results.

Formalizing `apply$` involves the kinds of considerations and compromises that are familiar to anybody trying to build an effective automatic theorem prover for a sufficiently powerful and convenient logic. Will the new feature be sound? Will it provide the power and convenience we seek? Can we support proofs of lemmas about it? Can we arrange for the lemmas we prove to be used automatically in other proofs?

This paper is about those considerations and compromises and the logical and engineering solutions we have found while formalizing `apply$` and extending the ACL2 theorem prover and system to support it.

[3] We use "`apply$`" instead of "`apply`" because the latter is the corresponding Common Lisp primitive which is a true functional. `Apply$` and `apply` are equal on "tame" "guard verified" function symbols when the actuals satisfy the "guard" of the function. See Section 11.

3 Overview of Our Solution

The following examples define "functionals" `sum` and `foldr`. Technically, these functions are scions of `apply$`.

```
(defun sum (fn x)
  (if (endp x)
      0
      (+ (apply$ fn (list (car x)))
         (sum fn (cdr x)))))
```

and

```
(defun foldr (lst fn init)
  (if (endp lst)
      init
      (apply$ fn
              (list (car lst)
                    (foldr (cdr lst) fn init)))))
```

The definition of `sum` above may be paraphrased as: If x is the empty list, then the value of `(sum` f x`)` is `0`; else, the value of `(sum` f x`)` is obtained by adding (a) the value of `apply$` on f and the singleton list containing the first element of x, written `(car` x`)`, and (b) the recursively obtained value of `sum` on f and the rest of x, written `(cdr` x`)`.

With `sum`, for example, we could express `(loop for` v `in` *lst* `sum (cube (abs` v`)))` as `(sum '(LAMBDA (V) (CUBE (ABS V)))` *lst*`)`. But this requires that `sum` and thus `apply$` be able to handle "`LAMBDA` expressions" composed of the "functions" corresponding to those used in the `loop` statement. It is easy to make `apply$` interpret any finite set, $\mathcal{S}$, of function symbols independent of `apply$`: define `apply$` as a "big switch" to look for each symbol in $\mathcal{S}$ and call the corresponding function. Since each member of $\mathcal{S}$ is independent of `apply$` there is no recursion and thus no difficulty in admitting this definition of `apply$`. But this is too restrictive a solution. First, consider the nested `loop`

`(loop for` x `in` *lst* `sum (loop for` y `in` x `sum (abs` y`))).`

This expands to

`(sum '(LAMBDA (X) (SUM '(LAMBDA (Y) (ABS Y)) X))` *lst*`)`

So `apply$` must be able to interpret `SUM`, which cannot be in the set $\mathcal{S}$ described above. Second, after `apply$` is introduced the user may wish to define more functions and `apply$` should be able to handle them. So we need an extensible `apply$` that can handle at least some scions of `apply$`.

We also want to be able to execute ground instances of `sum` expressions. For example, if `square` has been defined to square its argument, then `(sum 'SQUARE '(1 2 3))` should evaluate to `(+ 1 4 9)` = `14`, and if the variable *lst* in

`(foldr` *lst*

```
       '(LAMBDA (X Y)
                (FOLDR Y 'CONS (CONS X 'NIL)))
       nil).
```

were replaced by `'(1 2 3)` the ground term should evaluate to `'(3 2 1)`, i.e., the reverse of *lst*.

We want to be able to prove and use theorems about these concepts. For example, `sum` distributes over list concatenation,

```
(sum f (append x y))
=
(+ (sum f x) (sum f y))
```

for any f, and

```
(foldr lst
       '(LAMBDA (X Y)
                (FOLDR Y 'CONS (CONS X 'NIL)))
       nil).
=
(reverse lst).
```
[4]

And we want all this to be sound. Indeed, we want the Definitional Principle to remain conservative.

Naively, the function `apply$` is understood as a predefined ACL2 primitive whose axiomatic behavior evolves as new functions are introduced. For example, if we define `f` with

```
(defun f (x) x)
```

then we might reasonably expect the equation

```
(equal (apply$ 'f (list 3))
       3)
```

to be provable, but if instead we define `f` with

```
(defun f (x) (+ 4 x))
```

then we might instead reasonably expect that this time, the equation

```
(equal (apply$ 'f (list 3))
       7)
```

is provable. Clearly the axiomatic behavior of `apply$` has evolved in different ways for these two examples.

This naive understanding of an evolving `apply$` is problematic because ACL2's Definitional Principle is conservative: the function symbol `f` is not ancestral in the term `(apply$ 'f (list 3))`, in spite of the occurrence of the constant symbol `'f`, since the function symbol `f` is not called in that

[4] This equation is not a theorem. It requires a hypothesis called a "warrant" to apply `'FOLDR`, which we discuss later.

term[5]; thus both equalities above should be provable before the definition of `f` is introduced, which is a contradiction! We address this problem later, but for the moment we stick with the naive understanding of `apply$` as a predefined primitive that evolves.

`Apply$` will handle `LAMBDA` objects by evaluating the body of the expression in an appropriately constructed environment binding the formals of the `LAMBDA` object. This means we must formalize an expression evaluator as part of our formalization of `apply$`. Our name for that evaluator will be `ev$`. Both `ev$` and its counterpart for evaluating lists of expressions, `ev$-list`, will be defined mutually recursively with `apply$`. But our discussion focuses on `apply$`.

The naive expectation is that if ψ is a function symbol naming a function of arity n in ACL2, then

Expected Behavior of `apply$ on` ψ
`(apply$ '`ψ *args*`)` = `(`ψ `(nth 0` *args*`)` ... `(nth` $n-1$ *args*`))`

It is clear we can achieve this expectation for a large number of ACL2 primitives, like `car`, `cdr`, `cons`, etc., in which `apply$` is not ancestral, by just using the big switch idea. We call that big switch function `apply$-prim` (for "`apply$` a primitive") and it handles about 800 ACL2 primitives.

We will strive to achieve the Expected Behavior for user-defined functions ψ, but given the first-order nature of ACL2 will not be able to do it for all ψ and *args*. Syntax alone suggests some restrictions. ACL2 supports multiple-value return, but each function, including `apply$`, must return a fixed number of values. If the Expected Behavior of `apply$` on `car` holds, then `apply$` must return one value on `car` and thus must return one value on every input. Thus this exact Expected Behavior cannot hold for a function that returns two or more values.[6]

But there are two major logical problems.

One, which we call the *local problem*, is that since ψ is not ancestral in `'`ψ, the "Expected Behavior" guideline violates conservativity. ACL2 has scoping mechanisms (see :DOC encapsulate and :DOC books) within which one can declare an event to be *local* (see :DOC local), which limits the effect of that event to within the scope. One can "export" theorems from a scope if the locally defined functions are not ancestral in the exported theorem. This causes problems for `apply$`. We might define ψ in some local event of a scope, prove some theorem about the constant `'`ψ without mentioning the function ψ, then pop out of the scope exporting the "theorem." But the exported formula is not necessarily a theorem outside of the scope since ψ could be defined differently now. Our solution to the local problem is to avoid adding axioms that extend

[5] Formally the symbol `F` is just obtained from the ASCII code for 'F', which is 70, by `(intern (coerce (list (code-char 70)) 'string) "ACL2")`. The function named `f` is nowhere to be found!

[6] This could be fixed by complicating our notion of "Expected Behavior" by making it return a list of values but since the vast majority of ACL2 functions return one value we settled for this simple form and await user complaints.

`apply$`'s behavior as new functions are introduced. Instead, when proving theorems involving `apply$` the formula must assume explicitly that `apply$` behaves as expected on the relevant functions. These assumptions are called "warrants" for the functions. Because ψ is ancestral in its warrant, theorems about locally defined functions cannot be exported.

The other problem is illustrated by

```
(defun russell (fn) (not (apply$ fn (list fn))))
```

This is (apparently) an admissible function if `apply$` is predefined since it does not call itself recursively. Furthermore, some applications of `russell` produce reasonable results in ACL2's type-free logic, as shown below.

```
(russell 'NOT)
=                                {definition of russell}
(not (apply$ 'NOT '(NOT)))
=                                {Expected Behavior of apply$ on NOT}
(not (not (nth 0 '(NOT))))
=                                {evaluation of nth}
(not (not 'NOT))
=                                {evaluation of not}
T
```

But some applications of `russell` result in inconsistency if `apply$` behaves as "expected" on `RUSSELL`!

```
(russell 'RUSSELL)
=                                {definition of russell}
(not (apply$ 'RUSSELL '(RUSSELL)))
=                                {Expected Behavior of apply$ on RUSSELL}
(not (russell (nth 0 '(RUSSELL))))
=                                {evaluation of nth}
(not (russell 'RUSSELL))
```

Contradiction!

We do not want to prevent `apply$` from operating on all scions of `apply$` because that would prevent us from using scions like `sum` inside `LAMBDA` objects. So we have to allow `apply$` some "discretion" in when it behaves as expected and when it does not.[7] That "discretion" is based on how the functions to be applied treat their arguments and what arguments are supplied to them.

We seek a syntactic characterization of the "function" argument to `apply$` that will allow many interesting, practical applications while maintaining conservativity (and thus consistency). Our syntactic characterization is called "tameness".

[7] Since `apply$` (and every other ACL2 function) is total, problematic uses like `(apply$ 'RUSSELL '(RUSSELL))` are legal terms and thus have some semantic values, but we will not specify what they are.

Intuitively, a tame function is one whose body is a tame expression. To a first approximation, a tame expression is any expression whose evaluation is independent of `apply$`. However, we can allow `apply$` and other scions to be used in tame expressions provided the "functions" being passed are themselves tame. Thus, `(foldr` x `'(LAMBDA (X Y) (FOLDR Y 'CONS (CONS X 'NIL))) nil)` is tame, even though `foldr` is a scion, because the `LAMBDA` object passed to `foldr` is tame. The `LAMBDA` object is tame, even though it involves the scion `FOLDR`, because the function passed there is `CONS` (the primitive for creating an ordered pair), which is tame. The basic idea is that we can syntactically confirm that we always get down to functions independent of `apply$`. This idea is fundamental to tameness.

To identify tame functions syntactically we cannot permit new "functions" to be created during evaluation as would happen if `apply$` were applied to a term that computes a `LAMBDA` object. So one condition a tame function must satisfy is that we can classify each formal parameter as being treated exclusively as a "function" (destined for `apply$`), an "expression" (destined for `ev$`), or ordinary (never reaching either `apply$` or `ev$`).[8] We insist that only quoted constants and variable symbols being treated as "functions" occupy slots being treated as "functions."

Some functions do not permit such classification. For example, `russell`, as defined above, is rejected because its parameter, *fn* is used as a "function" when it appears as the first argument of `apply$` but not when it is used in the `list` term.[9]

In our formalization, the "badge" of a symbol, if any, includes the characterization of each formal parameter, and badges are used to determine whether functions and expressions are tame. Like `apply$`, `badge` is a function that "evolves" as new functions are introduced. For primitives, `badge` uses a big switch function, called `badge-prim`, to return the badges of the primitives. For user-defined symbols, `badge` calls the undefined function `badge-userfn` on the symbol. Warrants, mentioned earlier in connection with the local problem, assert the badges of user-defined functions by stipulating the value of the `badge-userfn`.

Warrants raise a new question. Do theorems bearing warrants in their hypotheses mean anything? If all warrants were identically false, the system would be sound and we would still be able to prove the warranted theorems and well as any formula containing a warrant as a hypothesis! Is there a way to show that all the warrants are satisfiable? The short answer to that is yes, we could have defined `apply$` in a big switch mutually recursive clique with scions. A suitable measure will exist due to tameness and the restrictions

[8] Because `apply$` interprets a `LAMBDA` object by evaluating the body with `ev$` we have to impose restrictions on how `ev$` is used to enforce the restrictions on how "functions" are used.

[9] This syntactic rejection of `russell` can be skirted by defining it to be `(defun russell (fn x) (not (apply$ fn (list x x))))`. The first formal of the revised `russell` is classified as a "function." `(Russell 'EQUAL 'EQUAL)` behaves as expected because the supplied "functional" argument, `'EQUAL`, is a tame function. But `(russell 'RUSSELL 'RUSSELL)` does not, because the "functional" argument, `'RUSSELL`, is not a tame function.

enforced when badges are computed. We can then prove that all the warrants are valid. See Section 10.

A second question that is natural to ask of any logical feature of `ACL2` is: Can we execute ground instances of scions? Without special provisions we cannot, because the axioms leave `apply$` unspecified (equal to a call of an undefined but constrained function) on user-defined functions. In proofs, warrant hypotheses alone specify how `apply$` behaves on user-defined functions. But to evaluate ground instances of scions at the top-level of ACL2's read-eval-print loop we have extended ACL2's pre-existing notion of the *evaluation theory* [19], which is an extension of the logical theory in which undefined but constrained functions may be given semantic interpretations for purposes of testing. We arrange for the evaluation theory to automatically assume all available warrants.

4 Organization

It is impractical in one paper to cover fully the material introduced above. We could write separate papers covering the formalization of `apply$`, its practical applications and limitations, how to prove useful lemmas involving it, why warrants do not render the undertaking vacuous, and how we address execution problems. But we feel no single paper would be interesting in its own right since, unless all the issues are addressed, `apply$` is just a logically limited, possibly unsound and/or non-executable hack.

Furthermore, all the work is available in the ACL2 Community Books repository on GitHub[10] [22].

The rest of this paper is organized as follows. In the next section, Section 5, we describe the formal definition of `apply$` and its mutually recursive peers, `apply$-lambda`, `ev$`, and `ev$-list`. The behavior of these functions depends on that of two undefined functions, `apply$-userfn` and `badge-userfn`, whose values are stipulated by warrants.

After introducing the formal definition of `apply$` and its peers, we revert to a bottom-up presentation to explain, in Section 6, how badges are inferred from admitted function definitions. In Section 7 we describe the formalization

[10] An ACL2 "book" is a file of formal definitions and theorems. Many books contain extensive comments and documentation. Thousands of books contributed by the user community are available via GitHub. A standard way of installing ACL2 from GitHub sources will download the community books so they are available as the subdirectory `books/` of your local ACL2 directory. The top of the GitHub directory for ACL2 is `https://github.com/acl2/-acl2`. The `*.lisp` files at the top level of that directory are copies of the *ACL2 source files* from the ACL2 home page [21]. A particular book may be found by clicking your way down the directory hierarchy. For example, to find `books/projects/apply-model/apply.lisp` start on the GitHub page above and select `books`, then `projects`, etc. If the GitHub books were installed with your ACL2 system you can execute an `include-book` form to load the book into your session. For example, `(include-book "projects/apply-model/apply" :dir :system)` will load your local copy of `projects/apply-model/apply.lisp`. Once a book is loaded into your ACL2 session you may inspect definitions, e.g., with `(pe 'apply$)`, and otherwise experiment with it.

of tameness. In Section 8 we describe how warrants are synthesized to stipulate the value of the undefined `badge-userfn` and the tameness conditions required by the undefined `apply$-userfn`. In Section 9 we bring all this together to show some examples of theorems ACL2 has proved about scions of `apply$`.

In Section 10 we describe our basic approach to showing that the warrants are all satisfiable. Given an arbitrary set of badged user-defined functions, we show that it is possible to *define* `badge-userfn` and `apply$-userfn` so that all the warrants are valid. This is a meta-theoretic argument about the ACL2 system, not a mechanically checked proof. Full details of the construction, including the proof that a certain measure decreases, are given in a long comment entitled "`Essay on Admitting a Model for Apply$ and the Functions that Use It`" in the file `apply-raw.lisp` found in the ACL2 source files at the top of the GitHub page for ACL2.

In Section 11 we describe how we have arranged for ACL2's top-level read-eval-print loop to execute ground terms that ancestrally involve `apply$` — even though the connection of `apply$` to user-defined functions is through the undefined function `apply$-userfn` and warrant hypotheses. We also discuss the optimization of `LAMBDA` object application. In Section 12 we report the results of some preliminary performance comparisons.

In Section 13 we briefly point out some expressive limitations of `apply$` all stemming from the simple fact that `apply$` only behaves as expected on syntactically tame functions, even though some non-tame functions are perfectly well-behaved!

In Section 14 we briefly mention related work, focusing on that which adds some of the convenience of higher-order functionality to a first-order system without changing the underlying logic. In Section 15 we briefly describe some ongoing work to improve the convenience and execution efficiency of `apply$`. We conclude, in Section 16, with acknowledgments.

Some supporting material may be found in the ACL2 Community Books repository on GitHub. The "living" definition of `apply$` as implemented in the current ACL2 is defined in the ACL2 source files named `apply*.lisp` at the top level of the GitHub page for ACL2 [22]. Those files are liable to evolve with the rest of ACL2, e.g., we expect some future version of ACL2 to allow `LAMBDA` objects include `DECLARE` forms. However, for archival purposes the formulation of `apply$` described in this paper is posted among the ACL2 Community Books repository under the GitHub ACL2 page (or your local ACL2 directory) at `books/projects/apply-model/`. This paper is essentially a readable `README` file for that directory. The book `books/projects/apply-model/apply.lisp` contains the complete formal definition of `apply$` as reported here[11]. The file `report.lisp` contains the example theorems about

[11] Following Common Lisp conventions for dealing with global name conflicts, the archival definition of `apply$` in the book `books/projects/apply-model/apply.lisp` is in its own package (i.e., namespace), `"MODAPP"` ("model of `apply$`"). This accomplishes two things. First, it allows the "living" `apply$` (which is in the `"ACL2"` package) to evolve without conflict. Second, it shows that the archival `apply$`, interesting theorems about it, and example constructions of models of it via "doppelgängers" (described below) making warrants prov-

scions presented in Section 9 of this paper. Two subdirectories, `ex1/` and `ex2/`, contain examples of the meta-level construction of a model making all warrants valid. The file `ex1/user-defs.lisp` contains just a few scions to model and the file `ex2/user-defs.lisp` contains many. We think of the files in `ex1/` being "surveyable" in the sense that the reader can better understand the model construction technique with the small set of scions. We think of `ex2/` as demonstrating that the model construction (and in particular, the termination argument which is only sketched here) works for a very large and complex set of functions.

5 The Definition of Apply$

The following four definitions are all introduced together in a mutually recursive clique (see :DOC mutual-recursion). Each is accompanied by an explicit measure term, which we discuss later. The definitions below have been simplified slightly, e.g., we use `(nth 2` *args*`)` for clarity below, where the actual definition has `(caddr` *args*`)` for runtime efficiency.[12] But these definitions are equivalent to the actual ones. See `books/projects/apply-model/-apply.lisp`.

Three undefined functions are used below: `apply$-userfn`, the behavior of which is described by warrants, and `untame-apply$` and `untame-ev$`, which remain completely unconstrained. These last two can be thought of as generating "don't care" values when `apply$` and `ev$` are presented with "illegal" inputs. The definition below uses `cond`, which provides a common way to abbreviate calls of `if`: the first clause says that if *fn* is a `cons` pair then return `(apply$-lambda` *fn args*`)`; the second clause says, else if `(apply$-primp` *fn*`)` is true then return `(apply$-prim` *fn args*`)`; else ... and so on.

```
(defun apply$ (fn args)
  (cond
   ((consp fn)
    (apply$-lambda fn args))
   ((apply$-primp fn)
    (apply$-prim fn args))
   ((eq fn 'BADGE)
    (badge (nth 0 args)))
   ((eq fn 'TAMEP)
    (tamep (nth 0 args)))
   ((eq fn 'TAMEP-FUNCTIONP)
    (tamep-functionp (nth 0 args)))
   ((eq fn 'SUITABLY-TAMEP-LISTP)
    (suitably-tamep-listp (nth 0 args) (nth 1 args) (nth 2 args)))
```

ably valid *can all be admitted to ACL2 without relying on any built-in knowledge of the "living"* `apply$`.

[12] `(Caddr` *args*`)` is a Common Lisp abbreviation for `(car (cdr (cdr` *args*`)))`.

```
  ((eq fn 'APPLY$)
   (if (tamep-functionp (nth 0 args))
       (apply$ (nth 0 args) (nth 1 args))
       (untame-apply$ fn args)))
  ((eq fn 'EV$)
   (if (tamep (nth 0 args))
       (ev$ (nth 0 args) (nth 1 args))
       (untame-apply$ fn args)))
  (t (apply$-userfn fn args))))
```

When *fn* is a cons, it is presumed to be a `LAMBDA` object and is applied using `apply$-lambda` as discussed in a moment. If *fn* is one of the 800+ primitives, it is applied with `apply$-prim`. The next four clauses deal with tame functions that could have been treated as primitives but happen to be defined in `books/projects/apply-model/apply.lisp`, so are not identified as primitive. But each of these clauses just recognizes the name of one of these functions and unconditionally calls the appropriate tame function on the correct number of arguments. The clauses handling `APPLY$` and `EV$` are most instructive: `apply$` can apply itself provided the leading element of *args* is a tame function, and `apply$` can apply `ev$` provided the leading element of *args* is a tame expression; otherwise, `apply$` returns a "don't care" value when applied to itself or `ev$`. Finally, on all other symbols, `apply$`'s behavior is determined by `apply$-userfn` — whose behavior is determined by warrants.

`LAMBDA` objects are handled by evaluating the body of the `LAMBDA` object with an association list — a list of pairs `(cons` x y`)` — that binds each of the `LAMBDA` object's formals, x, to the corresponding element of *args*, y.

```
(defun apply$-lambda (fn args)
  (ev$ (lambda-body fn)
       (pairlis$ (lambda-formals fn) args)))
```

This brings us to `ev$`, the expression evaluator. If the object, x, to be evaluated is not tame, `ev$` returns a "don't care" value. If x is a variable symbol, it looks up its value y (that is, finds a pair `(cons` x y`)`) in the association list a. If x is a quoted form, `(QUOTE` κ`)`, `ev$` returns κ. If x is `(IF` α_1 α_2 α_3`)`, `ev$` recursively evaluates α_1 and, conditionally on the result, evaluates α_2 or α_3. The next two clauses deal with how to evaluate forms beginning with `APPLY$` and `EV$` which we discuss in a moment. Otherwise, x is of the form `(`ψ α_1 ... α_n`)` and `ev$` evaluates it by `apply$`ing ψ to the list obtained by evaluating the α_i.

```
(defun ev$ (x a)
  (cond
   ((not (tamep x))
    (untame-ev$ x a))
   ((variablep x)
```

```
    (cdr (assoc x a)))
   ((fquotep x)
    (nth 1 x))
   ((eq (car x) 'IF)
    (if (ev$ (nth 1 x) a)
        (ev$ (nth 2 x) a)
        (ev$ (nth 3 x) a)))
   ((eq (car x) 'APPLY$)
    (apply$ 'APPLY$
            (list (nth 1 (nth 1 x)) (ev$ (nth 2 x) a))))
   ((eq (car x) 'EV$)
    (apply$ 'EV$ (list (nth 1 (nth 1 x)) (ev$ (nth 2 x) a))))
   (t (apply$ (car x) (ev$-list (cdr x) a)))))
```

The clause above dealing with `APPLY$` is subtly different than one might expect. The expected clause is

```
   ((eq (car x) 'APPLY$)
    (apply$ 'APPLY$
            (list (ev$ (nth 1 x) a) (ev$ (nth 2 x) a))))
```

but the actual clause above does not mention the first recursive call of `ev$` and instead uses `(nth 1 (nth 1 x))`. The discussion later of tameness will reveal that if x is a tame expression (which we know by the first clause of this `cond`) beginning with `APPLY$`, then `(nth 1 x)` is a quoted tame function, i.e., `(QUOTE` ψ`)`. The recursive `ev$` of that form is ψ, i.e., `(nth 1 (nth 1 x))`. So what is written above in the definition of `ev$` is actually equal to what might have been expected, but its justification in terms of a decreasing measure is simpler. The same trick has been used in the clause about `EV$`.

To see how `ev$` performs its "evaluation", consider the example `(ev$ '(FOLDR '(1) 'CONS 'NIL) nil)`. Since the `FOLDR` expression is tame, the `ev$` is equal to `(apply$ 'FOLDR '((1) CONS NIL))`. By the definition of `apply$`, `(apply$ 'FOLDR '((1) CONS NIL))` is `(apply$-userfn 'FOLDR '((1) CONS NIL))`, whose value is determined (if at all) by a warrant. If we have the warrant on `FOLDR` saying that we get the expected behavior when the "functional" argument (here, `CONS`) is a tame function, then this is `(foldr '(1) 'CONS 'nil)` = `'(1)`.

Finally, to evaluate a list of expressions we map `ev$` over the list argument, x, returning the empty list if x is empty and otherwise returning the list whose head and tail are constructed by calling `ev$` and (recursively) `ev$-list` on the head and tail of x, respectively.

```
(defun ev$-list (x a)
  (cond
   ((atom x) nil)
   (t (cons (ev$ (car x) a)
            (ev$-list (cdr x) a))))))
```

It remains to discuss the measures used to admit the `apply$` clique. Colloquially, why does `apply$` always terminate? To ensure admission, we must exhibit measures of the incoming arguments of each of the four functions in the clique so that whenever one of the functions calls itself or one of the others, the callee's measure of its incoming arguments is strictly smaller than the caller's measure of its incoming arguments. The measure we use is a lexicographic combination of two measures. The most significant component for the `ev$` and `ev$-list` functions is the size of the expression or list of expressions, measured with the built-in ACL2 function `acl2-count`. For `apply$`'s call of itself, that component is one greater than the size of the tame function to which it is being applied, i.e., `(+ 1 (acl2-count (car` *args*`)))`. The second component of the measures is always 0 except for `apply$-lambda`, where it is 1. See `books/projects/apply-model/apply.lisp` for details.

6 Badges

Consider

```
(defun foldr (lst fn init)
  (if (endp lst)
      init
      (apply$ fn
              (list (car lst)
                    (foldr (cdr lst) fn init)))))
```

`Foldr` treats its second argument, *fn*, exclusively as a function because it is passed into the "functional" slot of `apply$` and otherwise is only passed down in recursion. `Foldr` treats its first and third arguments, *lst* and *init*, exclusively as ordinary ACL2 objects — they are never used as functions. This information is gleaned from the definition of `foldr` when `(def-warrant FOLDR)` is executed. The warrant created for `foldr` stipulates

```
(badge-userfn 'FOLDR) = '(APPLY$-BADGE T 3 NIL :FN NIL).
```

(The warrant also constrains `apply$-userfn` on `'FOLDR`, but we discuss that later.)

The symbol `APPLY$-BADGE` is just a tag identifying this object as a badge. The `T` tells us that `foldr` returns a single value[13], the `3` is the arity of `foldr`, and the last three elements are the "ilks" of the respective formal parameters

[13] Functions that satisfy all the requirements imposed by `apply$` except for the requirement that they return single values have badges in which this component is `NIL`. Even though `apply$` cannot behave as expected on such symbols, the corresponding multiple-valued function might be used in the definition of a single-valued function and we need to track the argument use of the multiple-valued function to determine whether the single-valued function is acceptable. We could allow `apply$` to handle multiple-valued functions by complicating the "Expected Behavior" guideline but we have chosen not to.

and encode the above information about how `foldr` treats its arguments. As previously noted, not every admitted function definition can have a badge because some definitions violate our restrictions.

The *ilk* of a parameter with respect to a definition is one of the tokens `:FN`, `:EXPR`, or `NIL`, if it is possible to assign an ilk at all. We informally say these tokens denote that the parameter is "treated as a function," "treated as an expression," or "treated as an ordinary object." We extend the notion of ilk from just parameters to the corresponding argument positions or slots of calls of a function, ψ. So if the i^{th} parameter of ψ has a given ilk then we say the i^{th} actual in a call of ψ, e.g., α_i in `(`ψ $\alpha_1 \ldots \alpha_n$`)`, is in a slot of that same ilk.

We infer the ilks (and thus the badge) of ψ from its definition `(defun` ψ `(`$v_1 \ldots v_n$`)` β`)`. The process is akin to type inference and the process signals an error if the definition is found to violate certain restrictions. We do not give the algorithm here; for details see the definition of `badger` in `books/-projects/apply-model/apply.lisp`. If ψ is assigned a badge the following conditions are guaranteed.

- The function ψ does not take the ACL2 `state` (see :DOC state) or single-threaded objects (see :DOC stobj) among its inputs, is not defined as part of a mutually recursive clique, has a natural-number valued termination measure proved to decrease on each recursive call, and `apply$` is not ancestral in the measure.
- Every function called in the body β, except recursive calls of ψ, has a badge.
- Any v_i assigned an ilk of `:FN` or `:EXPR` only occurs in slots of that same ilk. Furthermore, there is at least one such occurrence.
- Every slot of ilk `:FN` or `:EXPR` is occupied by either a v_i of the appropriate ilk or by a quoted term. This implies that `cons`, for example, may not be used to create new `LAMBDA` objects in the definitions of badged functions.
- If v_i is assigned an ilk of `:FN` or `:EXPR`, then in every recursive call of ψ, the i^{th} actual is v_i. That is, formals of ilk `:FN` and `:EXPR` must be passed identically in recursion.
- If a quoted term, `(QUOTE` x`)`, occurs in a slot of ilk `:FN` then x is either a previously defined function symbol that is tame (see below), or a well-formed `LAMBDA` object with a tame body. This implies that every function symbol used in the body of a quoted `LAMBDA` object in a `:FN` slot is defined and badged, which in turn implies that no recursion through quoted `:FN` slots is allowed. An analogous restriction is imposed on quoted objects in `:EXPR` slots.

7 Tameness

Recall that a tame function is one whose body is a tame expression and that, to a first approximation, a tame expression is any expression that does not involve a scion of `apply$` unless the functions being passed into the `:FN` slots are tame.

To determine whether a function is tame we have to explore its definition, which here we treat as an object: tameness is a concept concerned with ACL2 objects used as data by `apply$` and `ev$`. We speak of tame symbols or `LAMBDA` objects (seen by `apply$`) and tame expressions (seen by `ev$`).

A symbol ψ is a *tame function symbol* iff it has a badge and the ilk of each parameter is `NIL`. All of the 800+ primitives recognized by `apply$-primp` are tame. But so is the function defined by

```
(defun fold-version-of-reverse (x)
  (foldr x '(LAMBDA (X Y) (FOLDR Y 'CONS (CONS X 'NIL))) nil)).
```

Even though it involves a call of `foldr`, the `LAMBDA` object is tame, as defined below.

A list is a *tame* `LAMBDA` *function* iff it is of the form `(LAMBDA` α β`)` where α is a list of n distinct symbols and β is a tame expression (see below).[14] We may extend the notion of badge (which actually is just defined for some symbols) to tame `LAMBDA` functions and let the arity field of the badge be n and each ilk be `NIL`. The object `(LAMBDA (X Y) (FOLDR Y 'CONS (CONS X 'NIL)))` is tame because even though it involves a call of `FOLDR` the function provided, `CONS`, is tame.

An object β is a *tame expression* iff it is (a) a symbol, (b) a list of the form `(QUOTE` κ`)`, or (c) a list of the form `(`ψ α_1 ... α_n`)` where ψ is either a symbol with a badge or a tame `LAMBDA` function, n is the arity in the badge of ψ, and each α_i is *suitably tame* with respect to the i^{th} ilk of ψ — that is, α_i is a tame function or expression if the i^{th} ilk is `:FN` or `:EXPR`, respectively.

As illustrated by `foldr-version-of-reverse` above, tame functions can be defined in terms of non-tame ones. In the `defun` below, `natp` is the function that returns `T` or `NIL` according to whether its argument is a natural number. If we define `square` to square its input, `collect` to apply its `:FN` argument to every element of its list argument, and `filter` to collect those elements of its list argument that satisfy its `:FN` argument, then

```
(defun sqnats (x)
  (collect 'SQUARE (filter 'NATP x)))
```

is tame and `(sqnats` x`)` collects the squares of the natural numbers in x.

In `books/projects/apply-model/apply.lisp` we formalize these various notions of tameness as ACL2 functions: `tamep-functionp` for recognizing tame function symbols and tame `LAMBDA` functions, `tamep` for recognizing tame expressions, and `suitably-tamep-listp` for recognizing suitably tame objects with respect to a list of ilks.

8 Warrants

Merriam-Webster defines the word "warrant" to mean a "commission or document giving authority to do something." We introduce the notion of warrants

[14] ACL2 imposes additional restrictions on its `lambda` objects, but the `LAMBDA` objects we are dealing with here are just objects being used as data by `apply$` and `ev$`.

for function symbols. A warrant for the symbol ψ gives `apply$` permission to apply the function of that name and tells it how to do so. Technically, an ACL2 warrant is a nullary predicate (which in ACL2 means a constant function of no arguments that returns `T` or `NIL`). We explain by example.

Consider the function `foldr` defined above. Its warrant is introduced by the command `(def-warrant foldr)` which defines `apply$-warrant-foldr` so that

Theorem.

```
(apply$-warrant-foldr)
↔
((badge-userfn 'FOLDR) = '(APPLY$-BADGE T 3 NIL :FN NIL))
 ∧
 [∀ args .
   (tamep-functionp (nth 1 args))
   →
   (apply$-userfn 'FOLDR args)
   =
   (foldr (nth 0 args) (nth 1 args) (nth 2 args))])
```

Note that `foldr` is ancestral in its warrant.

Using ACL2's `defun-sk` ("define Skolem function") to introduce the nullary function `apply$-warrant-foldr`, `def-warrant` proves (conditional) rewrite rules that make ACL2 behave as follows when `(apply$-warrant-foldr)` is among the hypotheses of a conjecture

- `(badge 'FOLDR)` is replaced by the constant `'(APPLY$-BADGE T 3 NIL :FN NIL)`, and
- `(apply$ 'FOLDR (list` a b c`))` is rewritten to `(foldr` a b c`)`, provided ACL2 can prove `(tamep-functionp` b`)`.

The rewrite rules for `(badge '`ψ`)` and `(apply$ '`ψ ...`)` are designed to "force" the assumption of the warrant for ψ. What this means is that the rewrites are carried out even if no warrant is available, but a separate subgoal is set aside to establish the warrant. A missing warrant will result in the corresponding subgoal's being unprovable. That unprovable subgoal is usually presented as a "checkpoint" at the end of the failed proof attempt, alerting the user to the need for the warrant.

If `def-warrant` is unable to determine a badge for the function, an error is caused.

The file `books/projects/apply-model/apply.lisp` defines `def-warrant` and introduces `(defun$` ψ ...`)` as an abbreviation for a sequence of two events, `(defun` ψ ...`)` followed by `(def-warrant` ψ`)`. The notation `(warrant` $\psi_1 \ldots$ ψ_n`)` is a convenient abbreviation for the conjunction of the warrants of the ψ_i, `(and (apply$-warrant-`ψ_1`) ... (apply$-warrant-`ψ_n`))`.

9 Examples

In this section we show a few scions and prove some relations connecting them. None of these results will surprise the reader familiar with higher-order languages and theorem provers. Furthermore, it should come as no surprise that limited second-order functionality can be supported in a first-order setting. What may be informative is that so much can be done with such expressive and theorem-proving ease, especially when one realizes that these proofs could all be done by ACL2 before any support for `apply$` was added (as demonstrated by the certification of the books in the disjoint namespace set up by `books/projects/apply-model/`).

The book `projects/apply-model/report.lisp` provides the ACL2 input for all the examples in this section. The theorem names, e.g., **Theorem** `T1`, used in our displays are the same as in `report.lisp`. At the end of this section we note some additional example books.

Assume we have already defined the following functions, none of which are scions of `apply$`, and assume all have warrants except for `append`, `natp`, and `evenp`, which are ACL2 primitives and hence do not need warrants.

- (`append` x y) - concatenation of lists x and y
- (`cube` x) - x^3
- (`flatten` x) - linear list of all atoms in the binary tree x
- (`natp` x) - `T` if x is a natural number, `NIL` otherwise
- (`nats` n) - the list of naturals from n down to 1
- (`rev` x) - reverse of the list x
- (`square` x) - x^2

Assume the following scions have been defined and warranted so that the formals named *fn* have ilk `:FN` and all other formals have ilk `NIL`. Also, let the successive elements of list *lst* be $e_1 \ldots e_n$.

- (`sum` *fn lst*): (`+` (*fn* e_1) ... (*fn* e_n))
- (`filter` *fn lst*): list of those e_i such that (*fn* e_i)
- (`foldr` *fn lst*): (*fn* e_1 (*fn* e_2 ... (*fn* e_n *init*)...))
- (`foldt` *fn lst*): a version of `foldr` for binary trees instead of linear lists

The formal definitions are

```
(defun$ sum (fn lst)
  (cond ((endp lst) 0)
        (t (+ (apply$ fn (list (car lst)))
              (sum fn (cdr lst))))))
(defun$ filter (fn lst)
  (cond ((endp lst) nil)
        ((apply$ fn (list (car lst)))
         (cons (car lst) (filter fn (cdr lst))))
        (t (filter fn (cdr lst)))))
(defun$ foldr (lst fn init)
  (if (endp lst)
```

```
      init
      (apply$ fn
              (list (car lst)
                    (foldr (cdr lst) fn init)))))
(defun$ foldt (x fn init)
  (if (atom x)
      (apply$ fn (list x init))
      (apply$ fn (list x (foldt (car x)
                                fn
                                (foldt (cdr x) fn init))))))
```

Without `apply$`, the ACL2 user wishing to sum the squares of the elements of *lst* would have to write `(sum-squares lst)` after introducing a recursive function to do the job

```
(defun$ sum-squares (lst)
  (if (endp lst)
      0
      (+ (square (car lst))
         (sum-squares (cdr lst)))))
```

If the theorem-proving task required the knowledge that this function distributes over `append`, that fact would have to be proved

`(sum-squares (append a b)) = (+ (sum-squares a) (sum-squares b)).`

Furthermore, if the user then needed to sum the cubes or the absolute values, etc., additional specialized recursive functions would first have to be introduced, and the corresponding distribution laws proved about each.[15]

But `sum` does the job for all such functions. For example, ACL2 can prove

Theorem `T1`
`(warrant square)` → `(sum-squares lst) = (sum 'SQUARE lst)`

and that `sum` distributes over list concatenation no matter what function is being applied

Theorem `T2`
`(sum fn (append a b)) = (+ (sum fn a) (sum fn b)).`

Notice that no warrant is required for *fn* here because this follows from the definition of `sum` regardless of how `apply$` is defined.

ACL2 can also reason about specific uses of `sum`, e.g., by proving:

Theorem `T3`
`((warrant square) ∧ (natp n))`
→
`(sum 'SQUARE (nats n)) = (/ (* n (+ n 1) (+ (* 2 n) 1)) 6).`

[15] These kinds of problems can be partially addressed with macros that introduce the required recursive functions and prove the indicated lemmas about them; for example see :DOC deflist and :DOC defdata. In our opinion this is not as elegant as introducing the scion itself as a function that can be directly named and used.

This theorem requires that the standard arithmetic book be loaded and a `hint` has to be provided. The hint forces `(sum 'SQUARE '(1))` to expand symbolically: ACL2's "natural" proof technique for this ground term is to evaluate it and that is impossible because `apply$-userfn` is undefined. We anticipate changing the prover to eliminate the need for a hint but so far we have resisted changes in the prover to support `apply$`. Note also that the warrant for `square` is required because `'SQUARE` is being applied. If the warrant is omitted the proof fails with the goal `(apply$-warrant-square)`, which is a clear indication the warrant is needed.

`Filter` is useful when one wishes to apply a function only to certain elements of a list. For example, we can define `sum-squares-of-evens` as the recursive function that sums each element of its argument that satisfies `evenp` and prove that it could have been defined with `sum` and `filter`.

Theorem `T4`

```
(warrant square)
→
(sum-squares-of-evens lst) = (sum 'SQUARE (filter 'EVENP lst)).
```

We do not need a warrant for `evenp` because it is an ACL2 primitive.

Since filter distributes over concatenation,

Theorem `T5`.

`(filter` *fn* `(append` *a b*`)) = (append (filter` *fn a*`) (filter` *fn b*`)),`

the user enjoys more generic reasoning than is possible with specialized functions.

`Foldr` is more flexible than the other scions used above. For example,

Theorem `T6`.

```
 (warrant square)
→
 (foldr lst
        '(LAMBDA (I A)
                 (IF (EVENP I)
                     (+ (SQUARE I) A)
                     A))
        0)
 =
 (sum 'SQUARE (filter 'EVENP lst))
```

is proved without assistance by ACL2.[16]

Theorem `T8`

```
 (warrant foldr)
```

[16] Inspection of the script in `projects/apply-model/report.lisp` reveals that we do not use the macro "`+`" in the `LAMBDA` object but use its expansion into a function of two arguments, `binary-+`, because `ev$` cannot handle macros. We similarly abbreviate the `LAMBDA` objects in `T9` and `T11` below.

```
→
 (foldr x
        '(LAMBDA (X Y)
                 (FOLDR Y 'CONS (CONS X 'NIL)))
        nil)
 =
 (rev x)
```

is particularly interesting since the `LAMBDA` object being passed to `foldr` calls `foldr` itself. The theorem above is proved automatically provided ACL2 has first proved

Theorem `T7`
`(foldr x 'CONS y) = (append x y).`

Still more generally, we can prove that for almost all *fn*, both `(sum` *fn lst*`)` and `(filter` *fn lst*`)` can be computed by appropriate uses of `foldr`. Below, we use the "backquote" notation of Common Lisp to describe `LAMBDA` objects obtained by substituting the value of *fn* for *fn* in the "near constants" shown.

Theorem `T9`

```
 (ok-fnp fn)
→
 (foldr lst `(LAMBDA (X Y) (+ (,fn X) Y)) 0)
 =
 (sum fn lst)
```

Theorem `T10`

```
 (ok-fnp fn)
→
 (foldr lst `(LAMBDA (X Y) (IF (,fn X) (CONS X Y) Y)) nil)
 =
 (filter fn lst).
```

By "almost all *fn*" we mean for *fn* satisfying `ok-fnp`, which checks that *fn* is not `QUOTE` and is a tame function of arity 1.

```
(defun$ ok-fnp (fn)
  (and (not (equal fn 'QUOTE))
       (tamep `(,fn X))))
```

Theorem `T11`

```
 ((ok-fnp f) ∧ (ok-fnp g))
→
 (foldr lst
        `(LAMBDA (X Y) (IF (,f X) (+ (,g X) Y) Y))
        0)
 =
 (sum g (filter f lst))
```

is potentially useful for converting an efficient computation into an easier-to-reason-about compositional one. The `foldr` makes one pass through *lst* whereas the compositional version essentially makes two.

`Foldt`, defined above, is like `foldr` but recurs over a binary tree accumulating the values of *fn* on the subtrees. Here is an interesting theorem about it

Theorem `T12-lemma`

```
(foldt x
       '(LAMBDA (X Y)
                (IF (CONSP X)
                    Y
                    (CONS X Y)))
       z)
=
(append (flatten x) z)
```

from which it follows (**Theorem** `T12`) that for the `LAMBDA` object above, `(foldt x '(LAMBDA ...) nil)` is `(flatten x)`.

Despite the power of generalized scions like `foldr` to express these and many other linear scions, we tend to prefer such functions as `sum` and `filter` because we find them easier to compose. Other useful scions we have defined and used include

- `(foldl` *lst fn ans*`)`: like `foldr` but tail-recursive
- `(prod` *fn lst*`)`: product of the values of *fn* on the elements of *lst*
- `(collect` *fn lst*`)`: list of the values of *fn* on the elements of *lst*
- `(collect-from-to` *fn start end incr*`)`: list of the values of *fn* on the numbers between *start* and *end* in steps of *incr*
- `(collect-on` *fn lst*`)`: list of the values of *fn* on the tails of *lst*
- `(collect-as` *fn lst1 lst2*`)`: list of the values of *fn* on corresponding elements of *lst1* and *lst2*
- `(all` *fn lst*`)`: `t` or `nil` according to whether every element of *lst* satisfies *fn*
- `(xists` *fn lst*`)`: `t` or `nil` according to whether some element of *lst* satisfies *fn*
- `(until` *fn lst*`)`: initial sublist of *lst* up to first element satisfying *fn*
- `(maxlist` *fn lst*`)`: maximum value of *fn* on elements of *lst*

In addition, it is sometimes useful to have versions of these functions that allow additional parameters to be passed to the `:FN` argument[17]

`(defun$ sum-with-params (`*fn lst params*`)`

[17] One might be tempted to allow `LAMBDA` objects containing free variables whose values are determined by the lexical environment. But then `apply$` would not be definable as a function. Consider `(apply$ '(LAMBDA (X) PARAMS) '(1))`. The arguments to `apply$` are constants, so if `apply$` is a function, the value must be a constant regardless of the lexical environment.

```
(if (endp lst)
    0
    (+ (apply$ fn (cons (car lst) params))
       (sum-with-params fn (cdr lst) params))))
```

or that map an expression over a domain, successively binding some variable to the next element:

```
(defun$ sum-expr (x lst a)
  (if (endp lst)
      0
      (+ (ev$ x (cons (cons 'X (car lst)) a))
         (sum-expr x (cdr lst) a))))
```

Note that the ilks of `sum-expr` are `(:EXPR NIL NIL)`. These two examples suggest ways that all the scions mentioned could be generalized to be more useful.

We can define

```
(defun$ russell (fn x)
  (not (apply$ fn (list x x))))
```

and prove (**Theorem** `T13`) that `(russell 'EQUAL 'EQUAL)` = `NIL`. But we cannot prove anything interesting about `(russell 'RUSSELL 'RUSSELL)`: by the definitions of `russell` and `apply$`, this is equal to `(not (apply$-userfn 'RUSSELL '(RUSSELL RUSSELL))`, but that term is undefined except by the warrant on `russell`. But the warrant tells us that `(apply$-userfn 'RUSSELL` *args*`)` behaves as expected provided `(tamep-functionp (car` *args*`))`. And (car *args*) is `'RUSSELL`, which is not tame. So the warrant is useless to us.

See the two books `books/projects/apply-model/ex2/user-defs.lisp` and `books/projects/apply-model/ex2/user-thms.lisp` for more examples, where we define a wide variety of scions and prove some theorems about some of the useful ones. Many of the scions in that `user-defs.lisp` book are quite artificial and were added to test and demonstrate our method for building a model of all warrants. They do, however, show a wide variety of recursions and intertwining of scions.[18]

10 A Model

Warrants were introduced to allow `apply$` to evolve as new functions are introduced, while simultaneously maintaining the conservativity of the Definitional Principle. Warrants restrict the behavior of the undefined functions `badge-userfn` and `apply$-userfn`. But if they over-restrict those functions,

[18] The reader should be aware that our various example definition files are not always compatible. For example, we may define a scion named `sum` in one file and place its `:FN` formal as the first formal, and then in another example file define `sum` so that its `:FN` formal is the second one.

our example theorems could be vacuously valid. In this section we briefly describe a meta-level proof that our theorems are not vacuous by virtue of all warrants being valid in a suitable *evaluation theory*.

Consider any ACL2 history, i.e., a sequence of admissible axiomatic events including in particular uses of `defun` and `def-warrant`. We show that there is a model of `badge-userfn` and `apply$-userfn` that satisfies all the axioms and that makes all the warrants valid. We stress that our construction of a model is not something that is or needs to be mechanized. The ACL2 system does not actually build a model. We offer this construction as a means of showing that a model satisfying all the warrants always exists for any admissible history. That said, we exhibit examples of this model construction in the two books named `user-defs.lisp` in subdirectories `ex1/` and `ex2/` of `books/projects/apply-model/`; more on that below.

A key idea is the notion of the *doppelgänger* of a function, f, which is a function with a different name, $f!$, and possibly a different definition, but which satisfies the constraints on f. This makes $f!$ and f equal if f is defined as opposed to merely constrained. But most importantly, for each function f that has been replaced in the construction by a doppelgänger $f!$, we can *define* f to be exactly $f!$; the resulting theory, the *evaluation theory*, thus extends the original theory, as desired.

The first step in the construction of the model is to define the doppelgänger for `badge-userfn`, named `badge-userfn!`. It is defined as a case split that recognizes each user-defined function f with a badge and returns that badge constant. We then define `badge!` just like `badge` except that it calls `badge-userfn!` instead of `badge-userfn`. We similarly define the functions `tamep!`, `tamep--functionp!`, and `suitably-tamep-listp!`, each defined like its counterpart but using the doppelgänger names.

Next, partition the badged user-defined functions into two groups, letting G1 contain those that are ancestrally independent of `apply$`, and letting G2 contain those that are ancestrally dependent on `apply$`, i.e., G2 is the set of scions.

Generally speaking, G1 functions do not need doppelgängers. We define each G1 function just as the user did, using the same name.[19]

The remaining badged functions are those in the `apply$` clique and the G2 functions. We define a doppelgänger for each and put them all in the same mutually recursive clique. The doppelgängers of all these functions are essentially derived by the standard renaming of functions to their doppelgängers. Full details of the construction are given in a long comment entitled "`Essay on Admitting a Model for Apply$ and the Functions that Use It`" in the file `apply-raw.lisp` found in the ACL2 sources.

The doppelgänger of `apply$-userfn`, aka `apply$-userfn!`, is *defined* as a big case split that recognizes each function name in G1 or G2. For G1 functions,

[19] Actually, if a G1 function, g, is ancestrally dependent on `badge` or the `tamep` functions, we do introduce its doppelgänger, $g!$, that instead calls doppelgängers. To see an example of this, look at `ok-fnp` in `books/projects/apply-model/ex2/user-defs.lisp` and its doppelgänger, `ok-fnp!` in `books/projects/apply-model/ex2/doppelgängers.lisp`.

it calls the corresponding function[20]. But for G2 functions it first checks the (doppelgänger versions of the) tameness conditions used in the warrant and if the appropriate arguments are tame it calls the doppelgänger of the G2 function.

Finding a measure that justifies this mutual recursion in the `apply$!` clique is the crux of the proof. The proof that the measure decreases in a well-founded sense across every inter-clique call establishes the existence of the doppelgängers defined in the clique.

Once all the doppelgängers are admitted, it is straightforward to prove by recursion induction that each doppelgänger satisfies the definitional equation of its counterpart, under the functional instantiation of replacing the constrained functions `badge-userfn` and `apply$-userfn` by their doppelgängers.

It is then straightforward to prove from the definition of `apply$-userfn!` and the equalities above that every warrant is valid under that same functional instantiation.

We return to the crux, which is that we can admit the clique under the Definitional Principle by exhibiting a suitable measure. First, we eliminate `apply$-lambda!` from the `apply$!` clique by inlining it and defining it later. Then to admit the simplified clique we assign a measure below ω^5 to each function in the clique. Think of each measure as a 5-tuple, $\langle a, b, c, d, e\rangle$, of natural numbers compared lexicographically. The measure for the doppelgänger of a user-defined function, f, has the following components listed from most significant to least (where the notion of "weights" is defined below).

- a: 0 if the tameness conditions for f are satisfied, else 1
- b: the maximum of (1) the weights of the `:FN` and `:EXPR` arguments, (2) the weights of every user-defined function g called in f other than f itself, and (3) the weights of every quoted constant used in a `:FN` or `:EXPR` slot in the body of f. The three parts, (1)–(3), are called the *internals* of a call of `f`.
- c: the position at which f got its warrant in the user's history, where the first function warranted gets position 2. (Earlier positions are given to `apply$`, `apply$-lambda`, `ev$`, and `ev$-list`.)
- d: the original measure justifying f
- e: 1

The *weight* of a cons is one greater than the sum of the weights of the car and cdr.

The *weight* of a user-defined function symbol f is the weight of its body at the time the `(def-warrant` f`)` occurred, where occurrences of the symbol in recursive calls are given weight 0. The *weight* of all other symbols is 0.

The measure for `(apply$!` *fn args*`)` is $\langle 0, b, 0, 0, 1\rangle$, where b is is the maximum of the weight of *fn* and, if *fn* has `:FN` or `:EXPR` arguments, then one more than the maximum weights of those arguments in *args*, else 0.

[20] For G1 functions with doppelgängers, i.e., those dependent on `badge`, `tamep`, etc, it calls the doppelgänger.

The measure for `(apply$-userfn!` *fn args*`)` is the same as for `apply$!` except that the last component is 0 instead of 1.

The measures for both `(ev$!` *x a*`)` and `(ev$-list!` *x a*`)` are $\langle 0, b, 1, 0, 1\rangle$, where b is the weight of x.

Proof Sketch: We must prove that these measures decrease in every inter-clique call. We do not give the proof here. But `apply$!` can call `ev$!` on the body of a `LAMBDA` object because the body has smaller weight than the `LAMBDA` object. `Apply$!` can call `apply$-userfn!` because the first four components are equal and the last component decreases from 1 to 0. `(Apply$-userfn` *fn args*`)` can call every user-defined scion because the second component of the caller's measure is strictly larger than the second component of the callee's measure, since the former measures the entire weight of the callee symbol and its `:FN` and `:EXPR` actuals but the latter only measures the internals.

As for calls from G2 functions, the key observations are (i) when a G2 function calls another G2 function, the weight of the caller is no smaller than that of the callee because the caller's internals include the callee, but if the weights are equal, the chronological positions settle the question appropriately, (ii) when a G2 function uses a quoted tame constant in a `:FN` or `:EXPR` slot, every "function" in the constant was defined earlier or else the constant would not be tame and the caller would not have gotten a badge, (iii) and when a G2 function calls itself the `:FN` and `:EXPR` arguments are passed down unchanged allowing the original measure to justify the call. **End Sketch**

Full details of the proof for any admissible user history are given the aforementioned "`Essay on Admitting a Model for Apply$ and the Functions that Use It`" in the file `apply-raw.lisp` found in the ACL2 sources.

The `books/projects/apply-model/` directory contains two examples of this modeling process and verification of the warrants. See the `README` file in that directory.

11 Execution

ACL2 supports (an extension of) a substantial subset of applicative Common Lisp and is written in Common Lisp.[21] ACL2 presents itself to the user as a Lisp read-eval-print loop: it reads an ACL2 term and, if the term is well-formed and ground, it evaluates the term according to the ACL2 axioms, and prints the result.

One might naively think that when the user defines an admissible ACL2 function, that same function is defined in the underlying Common Lisp and then the evaluation of ground terms is accomplished just by passing the term to the underlying Common Lisp engine. However, many Common Lisp functions are defined only on their *intended domains*, e.g., `(car 7)` is not specified by the Common Lisp semantics [32]. But the ACL2 axioms "complete" those semantics so that all primitive Common Lisp functions are fully specified,

[21] Indeed, except for a relatively small amount of Common Lisp boot-strapping code, ACL2 is written in itself.

e.g., `(car 7)` is `NIL` under the ACL2 axioms. The preconditions imposed by Common Lisp are formalized in ACL2 as *guards* (see :DOC guards) and ACL2 has a way to check, via theorem proving (see :DOC verify-guards), that the user's guards on a defined function are sufficient to imply that all Common Lisp functions involved are called only on objects in their intended domains [16]. When a function definition is admitted, ACL2 defines two versions of it in the underlying Common Lisp: one version is exactly what the user wrote and is here called the *fast version*; the other version, called the *executable counterpart*, calls the fast version if the user's guards have been verified to imply all the guards in the body and the arguments satisfy the user's guards, otherwise it runs code that implements the completion schemes adopted in the axioms. Thus, executable counterparts always return answers in accordance with the ACL2 axioms and can use the fast versions of any sub-computations known to be equivalent. When ground terms are evaluated, the executable counterpart is run. ACL2 computation agrees with the Common Lisp semantics when guards are verified and hold on the inputs.

Another difference between ACL2 execution and Common Lisp is that ACL2 permits new functions to be constrained but not defined (see :DOC encapsulate). When such a function is encountered in execution, an error is signaled.

But it is common for ACL2 users to employ undefined but constrained functions in complex system models and prototypes, to formalize unspecified behaviors. And it is common to test such complex models by running them on interesting examples, e.g., an x86 model [13,12] might be tested by running it on a binary image produced by `gcc`. To permit this, it must be possible for the user to temporarily attach executable functions to constrained functions for the purpose of testing such models. This attachment mechanism is called `defattach` (see :DOC defattach) and the set of attachments defines an extension of the user's logical theory. That extension, which preserves consistency, is called the *evaluation theory*. The evaluation theory is not available for reasoning during proofs; for all intents and purposes it is only used in the evaluation of forms entered by the user at the top-level of ACL2 read-eval-print loop.

Since `badge-userfn` and `apply$-userfn` are undefined but constrained functions, they block most calls of `apply$` and `ev$`. However, we have arranged for the evaluation theory to automatically include the attachments of the (virtual) doppelgängers to these functions. We stress that we do not actually define doppelgängers; because of the equivalence between functions and their doppelgängers, we can use the executable counterparts of the user's original functions.

`LAMBDA` object application is also somewhat optimized. Logically the execution of tame `LAMBDA` application proceeds by calling `ev$` on the body under an appropriate association list binding the variable symbols to values. However, when the `LAMBDA` object is well formed and has a verified guard of `T`, we can run compiled code for the corresponding Lisp function using the Common Lisp primitive `apply`. Furthermore, because tame `LAMBDA` objects are not formed on

the fly in function bodies, we can cache their suitability for this optimization and their compiled code.

Our `apply$` is Common Lisp's `apply` on tame guard verified function symbols when the actuals satisfy the guard of the function. This claim extends to tame guard verified `LAMBDA` objects.

For details of the handling of attachments to the functions `badge-userfn` and `apply$-userfn` and the checking, compiling, and caching of `LAMBDA` objects, see the ACL2 source file `apply-raw.lisp`.

12 Some Preliminary Performance Comparisons

In this experiment we will time runs of variations of

```
(sum '(LAMBDA (X) (BINARY-+ '3 (BINARY-* '2 (FIX X))))
     *million*)
```

where `*million*` is a list constant containing the naturals from 1 to one million, and `sum` is defined in Section 3. This is our proposed "formal semantics" for the Common Lisp expression

```
(loop for x in *million* sum (+ 3 (* 2 (fix x)))).
```

The arithmetic expression was arbitrarily chosen. `Fix` is the identity on numbers and returns 0 on non-numbers. Its presence in the expression means that the `LAMBDA` object can be legally executed on any input. If we knew this `LAMBDA` object would only be applied to numbers we would not need `fix`.

Here are some variations on this computation.

Variation 1: `(variation1 *million*)`, where

```
(defun variation1 (lst)
  (declare (xargs :guard t))
  (cond ((atom lst) 0)
        (t (+ (+ 3 (* 2 (fix (car lst))))
              (variation1 (cdr lst))))))
```

Variation 2: `(sum 'VARIATION2 *million*)`, where

```
(defun variation2 (x)
  (+ 3 (* 2 (fix x))))
```

Because no guard was declared, ACL2 does not consider `variation2` to be a "guard verified" function; the executable counterpart never runs the fast version.

Variation 3: `(sum 'VARIATION3 *million*)`, where

```
(defun variation3 (x)
  (declare (xargs :guard t))
  (+ 3 (* 2 (fix x))))
```

This ACL2 function is logically equivalent to `variation2` but has had its guards verified and so the fast function is run.

Variation 4: `(sum *variation4* *million*)`, where

```
(defconst *variation4*
  '(LAMBDA (X)
     (BINARY-+ '3 (BINARY-* '2 (FIX X)))))
```

but in this test we will disable the optimized handling of `APPLY$-LAMBDA` so that the body of this `LAMBDA` object is interpreted a million times by `ev$`.

Variation 5: `(sum *variation5* *million*)`, where

```
(defconst *variation5*
  '(LAMBDA (X)
     (BINARY-+ '3 (BINARY-* '2 (FIX X)))))
```

The two `LAMBDA` objects, `*variation4*` and `*variation5*`, are identical. However, before we run `*variation5*` we will install our optimized handling of `APPLY$-LAMBDA` so that it will confirm that the `LAMBDA` object can be applied to any input, compile the `LAMBDA`, and cache that compiled code, so the guard confirmation and compilation are not done again and the compiled code is run each time.

Variation 6: `(loop for x in *million* sum (+ 3 (* 2 x)))` in Common Lisp.

All of the timing is done with ACL2 Version 7.3 running CCL with the high optimization settings provided by ACL2, on a 2.6 GHz Intel Core i7 with 16 GB of 1600 MHz DDR3. Three successive runs of each variation were done and the times shown are the averages of the three.

variation	*description*	*seconds*
1	special purpose function	0.016
2	sum of fn symb with unverified guards	0.436
3	sum of fn symb with verified guards	0.333
4	interpreted `LAMBDA` object	6.490
5	compiled `LAMBDA` object	0.123
6	Common Lisp `loop`	0.012

Summarizing this table, variation **1**, the special-purpose recursive function, is the fastest way to do this sum purely in ACL2. However, next best is variation **5** where we use the `sum` scion with a `LAMBDA` object whose guards can be verified. We see that compiling and caching is well worthwhile, beating variation **4** where we interpret the `LAMBDA` object with `ev$`, by a factor of over 50. It was surprising to us that variation **3**, mapping with a function symbol whose guards have been verified is about 3 times slower than with the comparable `LAMBDA` object, **5**. This is mainly due to overhead associated with linking from the function name to its attachment in the evaluation theory and checking various conditions allowing the attachment to be run. Finally, we see that raw Common Lisp, variation **6**, beats the best scion expression, **5**, by a factor of about 10.

Overall we find these results encouraging but indicative that we must work harder to improve execution performance if we want industrial users of ACL2 to rely routinely on scions. One possibility is to support limited use of the Common Lisp `loop` construct in ACL2, which logically would expand to calls of scions but "under the hood" would be presented as `loop` to the host Common Lisp compiler in guard verified functions.

13 Limitations

The logic of ACL2 is first-order and untyped, with support for recursive definition over an inductively constructed domain. It remains so with the introduction of `apply$`. The addition of `apply$` does not turn ACL2 into Haskell [17] or ML [28] or HOL [15] or even Common Lisp [32]!

Many limitations of this work immediately come to mind: we cannot deal with functions of order three or higher, polymorphic typing, non-terminating functions, infinite lists, continuations, Currying, functions that create other functions, etc., all of which are common attributes of virtually every higher-order language.

Nevertheless, as the examples in the previous section show, `apply$` allows the convenient expression of many concepts and the convenient mechanical proof of many relationships connecting them. In particular, as hinted by the collection of scions enumerated in Section 9, we can handle many `loop` statements, provided the expressions occurring in them are tame. This was a goal of our work on `apply$`.

But within the context of that goal there are unfortunate limitations. As described in Section 6, we do not assign badges to functions that operate on `state` or single-threaded objects, are defined in mutually recursive cliques, or are justified with measures that are ancestrally dependent on `apply$` or return ordinals other than natural numbers. These limitations stem more from our desire to avoid complicating the project than from deep logical difficulties. We believe these limitations could be removed or at least made less restrictive — for example, we could more fully handle multiple-valued functions by having `apply$` always return a list of results rather than a fixed-length vector of results — but we await such requests from users.

The syntactic analysis that is used to infer tameness is overly restrictive. For example, `(sum 'SQUARE` *lst*`)` and `(sum 'CUBE` *lst*`)` are both recognized as tame expressions, but `(sum (if` *p* `'SQUARE 'CUBE)` *lst*`)` is not.

In addition, ACL2 does not rewrite constants, which means the rewriter cannot simplify or normalize the bodies of `LAMBDA` objects or even substitute one constant for a functionally equivalent one, e.g., `'IDENTITY` for `'(LAMBDA (X) X)`. In partial support of work in this direction, the `books/projects/apply-model/apply.lisp` book defines the predicate `fn-equal` using `defun-sk` so that it tests that two objects are functionally equivalent when used by `apply$`. The book proves that `fn-equal` is an equivalence relation and, ev-

ery time a scion is warranted, the system automatically generates and proves congruence rules [9] for its `:FN` arguments, e.g.,

`(fn-equal` *fn1 fn2*`)` → `(sum` *fn1 lst*`)` = `(sum` *fn2 lst*`).`

But because ACL2 does not rewrite constants, these rules are not used automatically.[22]

Another, less troubling, limitation is that `apply$` cannot apply an untame function even if, by "looking further ahead," it could determine in some cases that it will ultimately reach a tame function. For example, since we can prove that `(russell 'EQUAL 'EQUAL)` is `NIL` one might expect we can prove that `(apply$ 'APPLY$ '(RUSSELL (EQUAL EQUAL)))` is `NIL`. But that is not provable. `Apply$` can apply itself only to tame functions and `RUSSELL` is not tame. But if we looked one level deeper we would see that the untame `RUSSELL` is being fed the tame `EQUAL` and so could be expanded as expected.

14 Related Work

In this section we survey some related work, focusing especially on that which adds some of the convenience of higher-order functionality to a first-order system, *without modifying the underlying logic*. The italicized restriction is critical to this work and we explain why in the next few paragraphs, before comparing this work to similar work. Adding `apply$` to ACL2 did not change the term structure of the language, the rules of inference, the extension principles, or the logical foundations. This was crucial to the practicality of the undertaking since it eliminated the need to inspect and possibly modify hundreds of megabytes of prover code and previously verified libraries. Furthermore, the commitment to not changing the logic was the major constraint on any solutions we considered. Had we ignored this constraint the logical problems could have been trivially solved by adopting a higher order logic but the practical problems would have been insurmountable.

After wading through the myriad details of this paper, readers may question whether we really left the underlying logic unchanged! The answer is yes, because before we added `apply$` to the definitions in our source code we simply *defined* it, acting as ordinary users, under ACL2's standard, conservative Definitional Principle. All the theorems of Section 9 were proved by that earlier version of ACL2. Indeed, the theorems in `books/projects/apply-model/report.lisp` are all proved about the definition of `apply$` in a disjoint namespace (the symbol package `"MODAPP"`), demonstrating that none of now-built-in handling of `apply$` is necessary for these formulas to be theorems. The bulk of this paper describes the ramifications of our definition of `apply$`, `tamep`, warrants, etc., but the logical foundations were never threatened.

One might then ask "then why bother with all that talk, in Section 10, of the model of `apply$`, doppelgängers, warrants, etc.?" The answer is that

[22] In future work, we may explore the use of conditional rewrite rules instead.

the model establishes that theorems governed by warrant hypotheses are not vacuously valid.

But to support efficient execution of `apply$` in the evaluation theory we had to modify ACL2's source code, and to do that soundly we had to add the definitions of `apply$`, `tamep`, etc., to the source definitions to "tie down" the meanings of those function symbols.

Our commitment to not modifying ACL2's first order logic is so important to us that this section focuses on work undertaken with comparable restrictions: adding some second-order functionality to a first-order prover without modifying the underlying logic.

There is a long and distinguished history of developing set theory in first-order settings, including the set theories of Ackermann, Zermelo-Fraenkel, Von Neumann-Bernays-Gödel, and Morse-Kelly. We regard that work to be in a completely different sphere than this; we are not trying to support higher-order logic, just to make a first-order interactive theorem prover a little more convenient.

In the same sense, work on implemented higher-order systems is incomparable, including programming languages (e.g., Haskell [17], ML [28], and Common Lisp [32]), interactive theorem provers (e.g., HOL [15], Isabelle [29], and Coq [4]), and more automatic theorem provers (e.g., TPS [1], Satallax [10] and Leo-II [3]). ACL2 with `apply$` is still first-order, and our "functionals" and the "functions" they handle are much more limited than the true function objects in such systems.

Higher-order interactive theorem provers can take advantage of fully automatic theorem provers, by translating from higher-order logic to first-order logic [26]: in essence, translate $f(x)$ to $apply(f, x)$. That approach is used, for example, by the popular Sledgehammer tool [5]. Such work differs from ours: it supports a higher-order interactive prover using first-order tools, while ours enhances a first-order interactive prover to provide some of the benefits of higher-order systems.

There have been previous approaches that include features into a first-order setting that can be viewed as higher-order in nature. Some of these extend first-order systems to support proofs by induction (already supported by ACL2). Indeed, Peano arithmetic includes schematized induction, so we do not view induction as particularly higher-order. A second approach is to allow a function symbol to represent more than one function. Kŭncar [23] deals with such overloading for Isabelle, and points to an example from Obua that is reminiscent of the `RUSSELL` example given in Section 3 above. But overloading alone does not support data objects that represent functions. Indeed, the `encapsulate` and functional instantiation features [8] already provided by ACL2 and its predecessor, Nqthm [7], support forms of generic functions that are, in a sense, extensible; but they are not adequate for the kinds of examples discussed above in Section 9, nor do they support the use of `loop` constructs (see Section 2). A third approach, implemented in Beeson's Otter-λ [2], is based on an extension of first-order syntax to include lambda terms that de-

note functions. This extends the power of the automatic first-order prover, Otter [25], with "the power of lambda unification."

By contrast, ACL2 is an interactive prover whose logic does not have lambda terms. Rather, the first argument of `apply$` may be a *list* whose first element is the *symbol*, `LAMBDA`. Our use of first-order data items to represent functions is by no means new; indeed, this idea is present in ZF set theory and, more recently, goes back to McCarthy's early LISP work [24] and is explored in the use of *defunctionalization* by Reynolds [31] to formalize the operational semantics of a higher-order language using a first-order interpreter (where the lambdas may have free variables, which we do not allow in our lambda constants). Our contribution is not to the creation of interpreters, nor does it pertain to logics that augment the inference rules of traditional first-order logic, such as rewriting logic [27]. Our contribution, rather, is to extend an interactive theorem prover to provide convenient support for a higher-order style that can serve as a basis for `loop` constructs, while staying firmly within traditional first-order logic and extending automation already provided by the system to work nicely with the new features.

Previous work of Boyer and Moore (a co-author of this paper) dealt with an effort to support some higher-order functionality in a mechanized theorem prover for a first-order system: the work on `V&C$` in Nqthm [6]. In that work, `V&C$` was axiomatized to be an uncomputable but first-order universal evaluator that returned the "*V*alue and *C*ost" of an expression as a pair, or else returned `nil` indicating that no cost was sufficient. Every time a new definition was added, axioms were added to extend the functions `formals` and `body`, used by `V&C$`. Because Nqthm did not have local scopes, this was not the problem it would be in ACL2. Many commonly used examples of Lisp-style `loop` expressions were formalized and properties were proved.

Nqthm was strictly more powerful than ACL2 with `apply$` in the following sense: `V&C$` allows one to formalize and prove that some expressions could not be evaluated. However, overall we feel that ACL2 is a superior system, as witnessed by the scale of its industrial applications, its flexibility as a system programming language, its speed of execution, and the number and difficulty of its applications. See [18] for a close-up look at several uses of ACL2 at one company, or inspect the ACL2 Community Books at GitHub.

It has been noted that with just `foldr` (sometimes called `lit` [14]), one could define explicitly many first-order functions one would normally define recursively. In order to reason about these functions, the only inductive reasoning necessary would thus be to prove a few inductive properties of `foldr`.

We considered a similar observation when contemplating our proof of the existence of a model for warrants. Had we restricted scions to those easily recognizable as instances of `foldr` and `foldl`, we could simply show that those two functions could be defined as part of the `apply$` clique and introduce all other scions as instances. However, we preferred to prove our more general scheme consistent because of its directness and convenience to the user.

15 Note on Ongoing Work

The paper was originally submitted to this special issue of the **Journal of Automated Reasoning** in April, 2017. Reviewers accepted the paper in August, 2018, but made several helpful suggestions and requests to which we responded in October, 2018. However, between April, 2017 and October, 2018, we have continued to work on `apply$`. These changes are not reflected in this paper because they do not change the logical story. However, we anticipate that future releases of ACL2 will expand the support for `apply$`. For example, we can see how to support macroexpansion in `LAMBDA` object bodies and how to allow `LAMBDA` objects containing declarations (as ACL2's `defun` has always done). The latter change will allow the user to specify guards on quoted `LAMBDA` expressions (a feature of `LAMBDA` objects not otherwise mentioned in this paper) and the compiler to optimize the compilation of such expressions. We also anticipate a much more sophisticated cache implementation supporting general guards on `LAMBDA` objects. We anticipate supporting a subset of Common Lisp `loop` via formal translation to scions. We still anticipate working on performance, particularly for the scions of `apply$` involved in the semantics of Common Lisp `loop` statements. Optimal performance may require exploiting "output contracts" [11] or Jared Davis' "return specifiers" as described in the book `books/std/util/define.lisp`. If and when these new features are adopted they will be described in ACL2's online documentation.

16 Acknowledgments

We thank the reviewers for their helpful suggestions. We thank ForrestHunt, Inc., for its support of the ACL2 project in general and this work in particular. We also would like to acknowledge Mike Gordon, whose support for higher-order logic never diminished his enthusiasm for our first-order provers and with whom we shared many hours of interesting and enlightening conversations.

References

1. Peter B. Andrews and Chad E. Brown. TPS: A hybrid automatic-interactive system for developing proofs. *Journal of Applied Logic*, 4(4):367 – 395, 2006. Towards Computer Aided Mathematics.
2. Michael Beeson. Otter-lambda, a theorem-prover with untyped lambda-unification. In Geoff Sutcliffe, Stephan Schulz, and Tanel Tammet, editors, *Proceedings of the ESFOR workshop at IJCAR 2004*, 2004.
3. Christoph Benzmüller, Nik Sultana, Lawrence C. Paulson, and Frank Theiß. The higher-order prover LEO-II. *Journal of Automated Reasoning*, 55(4):389–404, 2015.
4. Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010.
5. Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. Hammering towards QED. *J. Formalized Reasoning*, 9(1):101–148, 2016.

6. R. Boyer and J S. Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. *Journal of Automated Reasoning*, 4(2):117–172, 1988.
7. R. S. Boyer and J S. Moore. *A Computational Logic Handbook, Second Edition*. Academic Press, New York, 1997.
8. R.S. Boyer, D.M. Goldschlag, M. Kaufmann, and J S. Moore. Functional instantiation in first-order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.
9. B. Brock, M. Kaufmann, and J S. Moore. Rewriting with equivalence relations in ACL2. *Journal of Automated Reasoning*, 40(4):293–306, 2008.
10. Chad E. Brown. Satallax: An automatic higher-order prover. In *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, pages 111–117, 2012.
11. Harsh Chamarthi, Peter C. Dillinger, and Panagiotis Manolios. Data definitions in the ACL2 sedan. In *ACL2 '14*, pages 27–48. EPTCS, 2014.
12. S. Goel, W.A. Hunt, and M. Kaufmann. Simulation and formal verification of x86 machine-code programs that make system calls. In K. Claessen and V. Kuncak, editors, *FMCAD'14: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, pages 91–98. EPFL, Switzerland, 2014.
13. Shilpi Goel. *Formal Verification of Application and System Programs Based on a Validated x86 ISA Model*. PhD thesis, University of Texas at Austin, 2016.
14. M. J. C. Gordon. On the power of list iteration. *The Computer Journal*, 22(4):376–379, 1979. Based on the author's 1973 technical report, `http://www.brics.dk/~danvy/MIP-R-101.pdf`.
15. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY, USA, 1993.
16. D. Greve, M. Kaufmann, P. Manolios, J S. Moore, S. Ray, J. L. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient execution in an automated reasoning environment. *Journal of Functional Programming*, 18(01), January 2008.
17. The Haskell home page, Accessed: 2017. `https://www.haskell.org`.
18. W. A. Hunt, Jr., M. Kaufmann, J S. Moore, and A. Slobodova. Industrial hardware and software verification with ACL2. In *Verified Trustworthy Software Systems*, volume 375. The Royal Society, 2017 (to appear). (Article Number 20150399).
19. M. Kaufmann. Trusted extension of ACL2 system code: Towards an open architecture. In *Workshop on Trusted Extensions of Interactive Theorem Provers*, 2010. See `http://www.cs.utexas.edu/users/kaufmann/itp-trusted-extensions-aug-2010/`.
20. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press, Boston, MA., 2000.
21. M. Kaufmann and J S. Moore. The ACL2 home page. In *http://www.cs.utexas.edu/users/moore/acl2/*. Dept. of Computer Sciences, University of Texas at Austin, 2018.
22. M. Kaufmann, J S. Moore, and ACL2 User Community. ACL2 sources and ACL2 community books on GitHub. In *https://github.com/acl2/acl2*. GitHub, 2018.
23. Ondřej Kunčar. Correctness of Isabelle's cyclicity checker: Implementability of overloading in proof assistants. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, CPP '15, pages 85–94, New York, NY, USA, 2015. ACM.
24. J. McCarthy. Recursive functions of symbolic expressions and their computation by machine (part I). *CACM*, 3(4):184–195, 1960.
25. W. McCune. Otter 3.0 Reference Manual and Guide. Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, IL, 1994. See also URL `http://www.mcs.anl.gov/AR/otter/`.
26. Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.
27. Jos Meseguer. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming*, 81(7):721 – 781, 2012. Rewriting Logic and its Applications.
28. Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, New York, NY, USA, 1991.

29. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.
30. Kent Pitman. The Common Lisp HyperSpec. See `http://www.lispworks.com/documentation/common-lisp.html`.
31. John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher Order Symbol. Comput.*, 11(4):363–397, December 1998.
32. G. L. Steele, Jr. *Common Lisp The Language, Second Edition*. Digital Press, 30 North Avenue, Burlington, MA. 01803, 1990.