Defining a LISP Interpreter in a Logic of Total Functions

Michael J.C. Gordon
The University of Cambridge Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
mjcg@cl.cam.ac.uk

ABSTRACT

We describe a preliminary study on embedding a Lisp interpreter in a logic that only admits total functions. The long-term goal is to verify an ACL2 interpreter in HOL. We hope the approach will scale to this task, but the details could turn out to be overwhelming. The current study uses pure Lisp to investigate the formulation of correctness. We define the semantics of Lisp evaluation operationally, using inductively defined relations. We then define a 'clocked' interpreter that is guaranteed to terminate and prove it equivalent to the operational semantics. This provides an executable Lisp evaluator that is a total function.

Keywords

LISP, APPLY, EVAL, HOL, ACL2

1. BACKGROUND

The following quote is from an earlier paper [9] on embedding the ACL2 logic [10] in higher order logic (HOL).

> Consider the ACL2 axiom `ASSOCIATIVITY-OF-*` occurring in the ACL2 source file `axioms.lisp`: `(EQUAL (* (* X Y) Z) (* X (* Y Z)))`. This can be viewed as an S-expression in ACL2's version of Lisp, or as a formula of first-order logic.
>
> Under the first view the axiom is valid because if `X`, `Y` and `Z` are replaced by any S-expressions, then the resulting instance of the axiom will evaluate to 'true', i.e., `T` in Common Lisp. Under the second view, the formula is an axiom that defines what it means for evaluation to be correct: it is a partial semantics of Lisp evaluation. Thus, in order to build a formal model of the ACL2 logic, we are faced with deciding whether to take Lisp evaluation or the ACL2 axioms as 'golden' – i.e., as the primary specification.

Recent work has developed the second view [8, 9, 14, 15]. In this paper we explore the first view, but just for pure Lisp [11]. Scaling to full ACL2 Lisp [1] is future work.

The pure Lisp interpreter is given by partial recursive functions *apply* and *eval* that operate on S-expressions representing Lisp programs [11, 12]. These functions can be defined using Scott domains [6], so an approach based in LCF [7] embedded in a total logic (e.g. HOLCF [13]) is a possibility. However, we explore a simpler operational method based on inductively defined relations. The basic idea is explained in the next section. We then apply the method to pure Lisp.

2. INDUCTIVE RELATION SEMANTICS

A standard way of representing partial recursive functions is as inductively defined relations. As an example consider the following recursive scheme defining function f, where p, q, h and k are given functions (p should return a truth-value).

$$f(x) = \texttt{if } p(x) \texttt{ then } q(x) \texttt{ else } h(x, f(k(x)))$$

For some choices of p, q, h and k there is no total function satisfying this equation (e.g. if $p(x) = \texttt{false}$, $k(x) = x$ and $h(x, y) = 1+y$ then the equation becomes $f(x) = 1+f(x)$). However, we can evaluate terms $f(a)$, for a given argument a, by repeatedly rewriting with the defining equation. Rewriting might not terminate (e.g. the example just mentioned), but when the function is defined it will compute the correct result. This kind of operational semantics can be formalised by defining the binary relation R_f such that $R_f(x, y)$ represents $y = f(x)$. R_f is the least relation satisfying:

$$\begin{array}{l}(\forall x.\ p(x) \Rightarrow R_f(x, q(x))) \\ \wedge \\ (\forall x\ y.\ \neg p(x) \wedge R_f(k(x), y) \Rightarrow R_f(x, h(x, y)))\end{array}$$

Many general theorem provers have support for making inductive definitions. In HOL4, for example, the user gives the formula above (with R_f as a free variable) to a tool (`HOL_reln`) which then generates the definition of the relation that is the least solution, together with reasoning support [5]. It is then easy to prove that if the equation defining f holds, then $R_f(x, y)$ entails $y = f(x)$:

$$\begin{array}{l}\vdash (\forall x.\ f(x) = \texttt{if } p(x) \texttt{ then } q(x) \texttt{ else } h(x, f(k(x)))) \\ \quad \wedge \\ \quad R_f(x, y) \\ \quad \Rightarrow \\ \quad y = f(x)\end{array}$$

Although relations are a way to represent partial functions

in a logic of total functions, they are cumbersome and not immediately executable. A widely used trick with Boyer-Moore provers is to add an extra argument – sometimes called a 'clock' – that decreases on each recursive call and forces the function to terminate. We use n for the clock variable. A first clocked version of f, say f_c, is defined by:

$$f_c(n,x) = \texttt{if } n=0 \texttt{ then u else}\\ \qquad\texttt{if } p(x) \texttt{ then } q(x) \texttt{ else } h(x, f_c(n{-}1, k(x)))$$

where u is a value indicating 'timeout' (clock n has hit 0). As a 'sanity check' it is easy to verify

$$\vdash R_f(x,y) \Rightarrow \exists n.\ f_c(n,x) = y$$

However, one would like also to know that if the clocked function doesn't timeout, then it returns the correct result. This would enable us to compute y from x by running $f_c(n,x)$ with a large n and then checking the result isn't u. Thus one would like f_c to satisfy:

$$\neg(y = \mathsf{u}) \wedge (\exists n.\ f_c(n,x) = y) \Rightarrow R_f(x,y)$$

This isn't true for all choices of u, but it is easy to show that it holds if $h(x,\mathsf{u}) = \mathsf{u}$:

$$\vdash (\forall x.\ h(x,\mathsf{u}) = \mathsf{u})\\ \quad\Rightarrow\\ \quad\forall y.\ \neg(y=\mathsf{u}) \Rightarrow ((\exists n.\ f_c(n,x) = y) \Leftrightarrow R_f(x,y))$$

For more complex examples, finding a suitable condition like $\forall x.\ h(x,\mathsf{u}) = \mathsf{u}$ can be hard (I was unable to find such a condition for the recursive Lisp interpreter functions *eval* and *apply* described in Section 3). A general solution is to use a well-known idea that first appeared in the denotational semantics of exceptions and was subsequently popularised via the 'exception monad' [16]. This idea is to have the clocked function return a value marked as a 'success' only when it really succeeds; when the clock times out a 'failure value', which is propagated unchanged by other functions, is returned.

A value is marked as a success is by having the clocked function return $\mathsf{some}(v)$ instead of just v. The failure value returned on clock-timeout is a value none chosen so that $\mathsf{none} \neq \mathsf{some}(v)$ for any v. The some/none terminology comes from option types, e.g. in HOL and ML. We use the same simple example of f to illustrate the new approach. The definition of the new clocked version of f, which will still be called f_c, uses a case$\cdots$of notation.[1]

$$f_c(n,x) = \texttt{if } n=0 \texttt{ then none else}\\ \qquad\texttt{if } p(x) \texttt{ then some}(q(x))\\ \qquad\qquad\texttt{else case } f_c(n{-}1,k(x)) \texttt{ of}\\ \qquad\qquad\qquad\mathsf{none} \rightarrow \mathsf{none}\\ \qquad\qquad\qquad\mid \mathsf{some}(y) \rightarrow \mathsf{some}(h(x,y))$$

If the recursive call $f_c(n{-}1,k(x))$ exhausts the clock, none is returned immediately. If $f_c(n{-}1,k(x))$ returns $\mathsf{some}(y)$, then $\mathsf{some}(h(x,y))$ is returned. Notice that f_c can only return none or a value of the form $\mathsf{some}(v)$. This 'exit-or-propagate' semantics could perhaps be written more slickly with monad-style combinators.

With the revised definition of f_c it follows by induction that:

$$\vdash R_f(x,y) \Leftrightarrow \exists n.\ f_c(n,x) = \mathsf{some}(y)$$

The $\Rightarrow$-direction uses a fixed-point induction principle automatically generated from the definition of R_f [5] and the $\Leftarrow$-direction is by mathematical induction on n.

[1]Define the function the so that $\mathsf{the}(\mathsf{some}(v)) = v$ holds and let $e[u/v]$ be the result of substituting u for v in e, then "case e of none $\rightarrow e_1$ | some$(v) \rightarrow e_2$" abbreviates "if $e = \mathsf{none}$ then e_1 else $e_2[\mathsf{the}(e)/v]$".

3. RELATIONAL PURE LISP SEMANTICS

Pure Lisp was introduced by McCarthy in 1960 in the paper *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I* [11]. Lisp computes on S-expressions (symbolic expressions), e.g.: NIL, X, 42, (A.B), (FOO (A B)). Functions on S-expressions are defined by recursion equations written in a notation called M-expressions (meta-expressions). McCarthy specified the Lisp semantics with a "Universal LISP Function" *apply* defined via a translation from M-expression function definitions to S-expressions. If fn^* is the translation of fn then *apply* is defined so that $apply[fn^*; [arg_1; \cdots; arg_n]; a]$ evaluates to $fn[arg_1; \cdots; arg_n]$, where the third argument a is an 'alist' that specifies values of variables occurring in the body of fn. In the early descriptions of Lisp interpreters there is no formal treatment of the semantics of M-expression recursive definitions. Perhaps it was assumed the semantics was clear by analogy with the theory of partial recursive functions over numbers. Before giving the relational definition of the pure Lisp interpreter we review S-expressions and M-expressions.

An S-expression is an atom or a dotted pair $(S_1.S_2)$ where S_1 and S_2 are S-expressions. An atom is either NIL, or is a number or a symbol. A proper list is NIL or an S-expression of the form $(S_1.(S_2.(\ \cdots\ (S_n.\texttt{NIL})\cdots)))$ which may be written $(S_1\ S_2\ \cdots\ S_n)$. An example of an S-expression that is a proper list is (FOO.(X.(Y.(5.NIL)))), where FOO, X and Y are symbols. This may be written (FOO X Y 5).

An M-expression term is either an S-expression constant c, or a variable v or the application $fn[e_1; \cdots; e_n]$ of a function fn to argument terms e_1,...,e_n, or a conditional term $[p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n]$, where p_i, e_i $(1 \leq i \leq n)$ are terms.

An M-expression function is either a primitive function constant k (where k is *car*, *cdr*, *cons*, *eq* or *atom*), or a function variable f, or a lambda $\lambda[[v_1; \ldots; v_n]; e]$, where $v_1, \ldots, v_n$ are term variables and e is a term, or a recursive definition $label[f; fn]$, where f is a function variable and fn a function. The *label*-operator is Lisp's syntax for fixed-points. For example, the recursive definition:

$ff[x] = [atom[x] \rightarrow x; T \rightarrow ff[car[x]]]$

defines the function that is denoted using M-expressions as:

$label[ff; \lambda[[x]; [atom[x] \rightarrow x; T \rightarrow ff[car[x]]]]]$

It is assumed that $k[arg_1; \ldots; arg_n]$ is defined for each function constant k and argument S-expressions $arg_1, \ldots, arg_n$. For example, $cons[x; y] = (x.y)$ (*cons* also needs to be defined when there are fewer or more than two arguments).

We define R_{ap} so $R_{ap}(fn, [arg_1; \cdots; arg_n], \rho, s)$ means that $fn[arg_1; \cdots; arg_n] = s$ if the free variables in fn have values specified by an environment ρ. We will model environments as maps from variables to S-expressions or M-expression functions, so if ρ is defined on a term variable v then $\rho(v)$ is an S-expression and if ρ is defined on a function variable f, then $\rho(f)$ will be an M-expression function. The notation $\rho[a/b]$ denotes ρ updated so variable b is mapped to a b can be a term or a function variable); $\rho[[a_1; \ldots; a_n]/[b_1; \ldots; b_n]]$ means $\rho[a_1/b_1] \cdots [a_n/b_n]$.

We define relation R_{ap} by mutual recursion with R_{ev} and R_{evl}. These relations correspond to the Lisp interpreter functions *eval* and *evlis*: $R_{ev}(e, \rho, s)$ holds if term e evaluates to the S-expression s with respect to environment ρ, and $R_{evl}(el, \rho, sl)$ holds if list el of M-expression terms evaluates to the list sl of S-expressions with respect to ρ. The relations R_{ap}, R_{ev} and R_{evl} are defined by cases on their first argument. They are the least relations such that:

$$
\begin{array}{l}
R_{ap}(k, args, \rho, k\ args) \\
\wedge \\
R_{ap}(\rho(f), args, \rho, s) \Rightarrow R_{ap}(f, args, \rho, s) \\
\wedge \\
R_{ev}(e, \rho[args/vars], s) \Rightarrow R_{ap}(\lambda[[vars]; e], args, \rho, s) \\
\wedge \\
R_{ap}(fn, args, \rho[fn/x], s) \Rightarrow R_{ap}(label[[x]; fn], args, \rho, s)
\end{array}
$$

and

$$
\begin{array}{l}
R_{ev}(c, \rho, c) \\
\wedge \\
R_{ev}(v, \rho, \rho(v)) \\
\wedge \\
R_{ev}([\,], \rho, \mathtt{NIL}) \\
\wedge \\
R_{ev}(p, \rho, \mathtt{NIL}) \wedge R_{ev}([gl], \rho, s) \Rightarrow R_{ev}([p \rightarrow e; gl], \rho, s) \\
\wedge \\
R_{ev}(p, \rho, x) \wedge x \neq \mathtt{NIL} \wedge R_{ev}(e, \rho, s) \Rightarrow R_{ev}([p \rightarrow e; gl], \rho, s) \\
\wedge \\
R_{evl}([el], \rho, args) \wedge R_{ap}(fn, args, \rho, s) \Rightarrow R_{ev}(fn[el], \rho, s)
\end{array}
$$

and

$$
\begin{array}{l}
R_{evl}([\,], \rho, [\,]) \\
\wedge \\
R_{ev}(e, \rho, s) \wedge R_{evl}([el], \rho, sl) \Rightarrow R_{evl}([e; el], \rho, [s; sl])
\end{array}
$$

Traditionally the recursion down a conditional is delegated to a separate function *evcon* [11, 12], but for simplicity we have folded this into the definition of R_{ev}.

The three relations defined above are analogous to R_f in Section 2. Now we define a clocked interpreter functions $apply_c$, $eval_c$ and $evlis_c$ analogous to f_c. The first argument is the clock, we use `case` $\cdots$ `of` notation and define $\mathsf{List}[s_1; \ldots; s_n] = (s_1 \cdots s_n)$ and $\mathsf{Split}((s_1 \cdots s_n)) = [s_1; \ldots; s_n]$.

$$
\begin{array}{l}
(apply_c(n, fn, args, \rho) = \\
\quad \mathtt{if}\ n = 0 \\
\quad \mathtt{then\ none} \\
\quad \mathtt{else\ case}\ fn\ \mathtt{of} \\
\qquad\qquad k \rightarrow \mathsf{some}(k\ args) \\
\qquad\quad \mid f \rightarrow apply_c(n{-}1, \rho(f), args, \rho) \\
\qquad\quad \mid lambda[vars; e] \rightarrow eval_c(n{-}1, e, \rho[args/vars]) \\
\qquad\quad \mid label[f; fn] \rightarrow apply_c(n{-}1, fn, args, \rho[fn/f])) \\
\wedge \\
(eval_c(n, e, \rho) = \\
\quad \mathtt{if}\ n = 0 \\
\quad \mathtt{then\ none} \\
\quad \mathtt{else\ case}\ e\ \mathtt{of} \\
\qquad\qquad c \rightarrow \mathsf{some}(c) \\
\qquad\quad \mid v \rightarrow \mathsf{some}(\rho(v)) \\
\qquad\quad \mid [\,] \rightarrow \mathsf{some}(\mathtt{NIL}) \\
\qquad\quad \mid [p \rightarrow e; gl] \rightarrow (\mathtt{case}\ eval_c(n{-}1, p, \rho)\ \mathtt{of} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{none} \rightarrow \mathsf{none} \\
\qquad\qquad\qquad\qquad \mid \mathsf{some}(s) \\
\qquad\qquad\qquad\qquad\qquad \rightarrow (\mathtt{if}\ s = \mathtt{NIL} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{then}\ eval_c(n{-}1, [gl], \rho) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{else}\ eval_c(n{-}1, e, \rho)) \\
\qquad\quad \mid fn[el] \rightarrow \mathtt{case}\ evlis_c(n{-}1, el, \rho)\ \mathtt{of} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{none} \rightarrow \mathsf{none} \\
\qquad\qquad\qquad\qquad \mid \mathsf{some}(sl) \rightarrow apply_c(n{-}1, fn, \mathsf{Split}(sl), \rho)) \\
\wedge \\
(evlis_c(n, el, \rho) = \\
\quad \mathtt{if}\ n = 0 \\
\quad \mathtt{then\ none} \\
\quad \mathtt{else\ case}\ el\ \mathtt{of} \\
\qquad\qquad [\,] \rightarrow \mathsf{some}(\mathtt{NIL}) \\
\qquad\quad \mid [e; el'] \rightarrow \mathtt{case}\ eval_c(n{-}1, e, \rho)\ \mathtt{of} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{none} \rightarrow \mathsf{none} \\
\qquad\qquad\qquad\qquad \mid \mathsf{some}(s) \rightarrow \mathtt{case}\ evlis_c(n{-}1, el', \rho)\ \mathtt{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{none} \rightarrow \mathsf{none} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{some}(sl) \rightarrow \mathsf{some}((s.sl)))
\end{array}
$$

The use of exception monad notation [16] might substantially simplify these definitions.

The connection between the relational and clocked functional semantics is similar to the connection between R_f and f_c in Section 2, and the proof is similar.

$$
\begin{array}{l}
\vdash (R_{ap}(fn, args, \rho, s) = \exists n.\ \mathsf{some}(s){=}apply_c(n, fn, args, \rho)) \\
\quad \wedge \\
\quad (R_{ev}(e, \rho, s) = \exists n.\ \mathsf{some}(s){=}eval_c(n, e, \rho)) \\
\quad \wedge \\
\quad (R_{evl}(el, \rho, sl) = \exists n.\ \mathsf{some}(\mathsf{List}(sl)){=}evlis_c(n, el, \rho))
\end{array}
$$

This function definition is expressed in HOL[2], but it could be coded in ACL2 Lisp to give a sound and complete interpreter inside ACL2. It is not clear how useful this might be though (see the discussion below).

4. DISCUSSION

In Section 1 two views of the semantics of Lisp were discussed: a programming language view versus a logic view. We have explored the first view here and shown how to define a Lisp interpreter in a classical logic of total functions. This interpreter could be used to validate the axioms

[2]See `http://www.cl.cam.ac.uk/~mjcg/papers/acl207/` for actual HOL formalisations and proofs.

of the logic. For example, the 'axiom' $car[cons[x;y]] = x$ can be validated by evaluating $car[cons[x;y]]$, i.e. proving $\vdash R_{ev}(car[cons[x;y]], \rho, x)$. Although we have not attempted to validate a Lisp theory, we have built a tool that can apply a function fn to arguments $args$ in a given environment ρ by running the clocked interpreter with a large clock and then, if a result $\mathsf{some}(s)$ is returned, deducing $\vdash R_{ap}(fn, args, \rho, s)$. This tool is a simple derived rule coded in ML that combines the first conjunct of the theorem at the end of the preceding section with the result of the clocked evaluation.[3]

Once we have a Lisp interpreter defined formally we can use it to interpret definitions of *apply* and *eval* coded as S-expressions [12, Section 1.6]. One should be able to prove, for example, that $R_{ev}(e, \rho, s) \Leftrightarrow R_{ap}(eval, [e^*; \rho^*], \rho_{init}, s)$, where e^* is than translation of e, ρ^* translates ρ to an alist S-expression and ρ_{init} contains the definitions of the Lisp interpreter functions (*assoc*, *pairlis*, *apply*, *eval* etc.) used by *eval*. We hope to perform this proof for pure Lisp to see how tricky it is and to get an estimate of the feasibility for ACL2, but have not attempted this yet.

Will the approach described here scale to ACL2? I am pretty confident that one *could* define an evaluator for ACL2 inside HOL and prove that it was consistent with the ACL2 logical theory [10]. This would integrate nicely with the prior work on modelling ACL2 in HOL [8]. A second trickier question is whether a formally defined and verified interpreter would be worth the effort? Are there any uses for verified guaranteed-total clocked *apply* and *eval* functions inside ACL2? One possibility is to provide support in ACL2 for bounded quantifiers. In Nqthm, such quantifiers are supported by `V&C$` [3, 4]. Investigating the relationship between the `V&C$` way of adding an interpreter to a logic of total functions and relational semantics would be interesting further research. Another possible application could be some kind of reflection of Lisp code verified inside ACL2 into the ACL2 trusted kernel [2], though I don't have a particular idea to suggest.

5. ACKNOWLEDGEMENTS

Many thanks to the anonymous referees for corrections, comments and pointers to related work.

6. REFERENCES

[1] The ACL2 System. http://www.cs.utexas.edu/users/moore/acl2/.

[2] R. Boyer and J. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In R. Boyer and J. Moore, editors, *The correctness problem in computer science*, pages 103–184. Academic Press, 1981.

[3] R. S. Boyer and J. S. Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. *J. Autom. Reason.*, 4(2):117–172, 1988.

[4] R. S. Boyer and J. S. Moore. *A computational logic handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988.

[3]See the definition of `apply` at the end of `http://www.cl.cam.ac.uk/~mjcg/papers/acl207/PureLisp.ml`

[5] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, August 1992.

[6] M. J. C. Gordon. Models of Pure Lisp. Technical report, Department of Machine Intelligence, School of Artificial Intelligence, University of Edinburgh, 1973. (Author's PhD Dissertation).

[7] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[8] M. J. C. Gordon, J. Warren A. Hunt, M. Kaufmann, and J. Reynolds. An embedding of the ACL2 logic in HOL. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 40–46, New York, NY, USA, 2006. ACM Press.

[9] M. J. C. Gordon, J. Warren A. Hunt, M. Kaufmann, and J. Reynolds. An integration of HOL and ACL2. In A. Gupta and P. Manolios, editors, *FMCAD '06: Proceedings of Formal Methods in Computer-Aided Design*, pages 153–160. IEEE Computer Society Press, November 2006.

[10] M. Kaufmann and J. S. Moore. A precise description of the ACL2 logic. Technical report, Department of Computer Sciences, University of Texas at Austin, 1997.

[11] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM*, 3(4):184–195, 1960. http://doi.acm.org/10.1145/367177.367199, http://www-formal.stanford.edu/jmc/recursive.html.

[12] J. McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.

[13] O. Müller, T. Nipkow, D. v. Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[14] J. Reynolds. Automatically translating type and function definitions from HOL to ACL2. In *TPHOLs 2007: Proceedings of the 20th International Conference on Theorem proving in Higher Order Logics*. Springer LNCS, September 2007. To be published.

[15] K. Slind and M. Kaufmann. Proof pearl: Wellfounded induction on the ordinals up to epsilon-0. In *TPHOLs 2007: Proceedings of the 20th International Conference on Theorem proving in Higher Order Logics*. Springer LNCS, September 2007. To be published.

[16] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.