The Deductive Synthesis of Imperative LISP Programs

Zohar Manna
Stanford University
Stanford, California

Richard Waldinger
SRI International
Menlo Park, California

Abstract

A framework is described for the automatic synthesis of imperative programs, which may alter data structures and produce destructive side effects as part of their intended behavior. A program meeting a given specification is extracted from the proof of a theorem in a variant of situational logic, in which the states of a computation are explicit objects. As an example, an in-place *reverse* program has been derived in an imperative LISP, which includes assignment and destructive list operations (*rplaca* and *rplacd*).

Introduction

For many years we have been working on the design of a system for *program synthesis*, i.e., the automatic derivation of a program from a given specification. For the most part, we have been concentrating on the synthesis of *applicative* programs, i.e., programs that return an output but produce no side effects (Manna and Waldinger [80], [87a]). Here we consider the synthesis of *imperative* programs, i.e., programs that alter data structures as part of their intended behavior. We adapt the same techniques that we have used for applicative programs.

We have developed a *deductive approach*, in which the construction of a program is regarded as a task in theorem proving. For applicative programs, we prove a theorem that establishes the existence of an output object meeting the specified conditions. The proof is restricted to be sufficiently constructive to indicate a computational method for finding the desired output. This method provides the basis for a program that is extracted from the proof.

The difficulty in adapting this deductive approach to imperative programs is that, if data structures are altered, a sentence that is true at a certain state of the computation of a program may become false at other states. In the logical theories in which we usually prove theorems, a sentence does not change its truth-value. A time-honored approach to this problem is to employ a *situational logic*, i.e., one in which states of the computation are explicit objects. Predicate and function symbols each have a state as one of their arguments, and the truth of a sentence may vary from one state to another.

In this paper, we adapt situational logic to the synthesis of imperative programs. We find that conventional situational logic is inadequate for this task, but formulate a new situational logic, called *imperative-program theory*, that overcomes this inadequacy. To be specific, we shall set down a theory of imperative LISP, which includes the destructive operators *rplaca* and *rplacd* and the assignment operator *setq*. We intend, however, that other versions of imperative-program theory shall be equally applicable to other languages.

Historical Notes

Situational logic was introduced into the computer science literature by McCarthy [63] and was also applied to describe imperative programs by Burstall [69]. It was used for the synthesis of imperative programs in the systems QA3 (Green [69]) and PROW (Waldinger and Lee [69]). We have used situational logic earlier to describe ALGOL-like programming languages (Manna and Waldinger [81]). Recently, we have adapted situational logic to be a framework for automatic planning (Manna and Waldinger [87b]).

Imperative LISP has recently been described (in terms of "memory structures") in the thesis of Mason [86]. We have translated many of Mason's notions into the situational logic framework. Mason applies his framework to proving properties of programs and to program transformation, but does not deal with synthesis from specifications. We also treat the assignment operation (*setq*), which Mason omits.

The Trouble with Conventional Situational Logic

To construct a program in conventional situational logic (e.g., the QA3 logic), one proves the existence of a final state in which the specified conditions will be true. One regards the initial state as the input and the final state as the output of the imperative program. In other words, one uses the same approach one would use for applicative programs, treating states as objects that can be passed around like numbers or lists.

The trouble with this approach is that one can construct programs that can perform more than one operation on the same state, contrary to the physical fact that, once an operation has been performed on a state, that state no longer exists. For example, it is possible to construct programs such as

This research was supported by the National Science Foundation under Grants DCR-82-14523 and DCR-85-12356, by the Defense Advanced Research Projects Agency under Contract N00039-84-C-0211, by the United States Air Force Office of Scientific Research under Contract AFOSR-85-0383, by the Office of Naval Research under Contract N00014-84-C-0706, by United States Army Research under Contract DAJA-45-84-C-0040, and by a contract from the International Business Machines Corporation.

$$bad(x) \Leftarrow \begin{cases} \textit{if } p(setq(x, square(x), s_0)) \\ \textit{then } setq(z, x, s_0) \\ \textit{else } setq(y, x, s_0). \end{cases}$$

According to this program, in our initial state s_0 we are to set x to x^2 and then test if condition p is true. If so, in our initial state s_0, we are to set z to x; otherwise, we are to set y to x. Unfortunately, once we have changed the value of x, we have destroyed our initial state and no longer have access to the initial value of x.

Imperative-program theory has been designed to overcome this sort of difficulty. In this theory, programs are denied access to explicit states, and always apply to the implicit current state.

Elements of Imperative-LISP Theory

In an imperative-program theory, we return the states and objects of situational logic and introduce a new sort of entity, called the *fluent*, which is best described in terms of what it does. We shall say that *evaluating* a fluent in a given state *produces* a new state and *returns* an object.

For example, evaluating the fluent $setq(x, 2)$ in a given state s produces a new state, similar to the given state except x has been set to 2. The evaluation also returns an object, the number 2.

We shall think of an imperative program as computing a function that, applied to a given input object (or objects), yields a fluent. For example, if we apply the imperative reverse program $nrev(a)$ to a list structure e, we obtain a fluent $nrev(e)$; evaluating this fluent in a given state will produce a new state (in which e is reversed) and return an object (the reversed list structure itself). Because we do not regard the state as an explicit input to the program, we cannot construct programs, such as $bad(x)$, that perform multiple operations on the same state.

To construct a program, we prove the existence of a fluent that, for a given initial state and input object, produces a final state and returns an output object satisfying the specified conditions. The actual program is then extracted from the proof. To be more precise, let us restrict ourselves to imperative LISP.

In imperative-LISP theory we distinguish among several sorts of objects.

- States. These are states of the computation.
- Locations. These may be thought of as machine locations or cells. We discriminate between
  - *Pairs.* These are conventional LISP cells. They "point" to two locations.
  - *Atoms.* These are identified with storage locations. They sometimes point to a single location, which must be a pair. The special atom *nil* cannot point to anything.

Atoms and pairs are assumed to be disjoint. Locations are the input and output objects of imperative-LISP programs.

- Abstract trees. These are finite or infinite binary trees, which may be *represented* by pair locations. We identify atoms with atomic abstract trees.
- Fluents. These may be thought of as functions mapping states into state-location pairs. We identify each atom with a fluent, which we describe later.

Note that the above sorts are not disjoint. In particular, atoms are included among the locations, abstract trees, and fluents. We can only identify atoms with storage locations because of the absence of simple aliasing in LISP; two distinct atoms are never bound to the same storage location.

Now let us describe the functions that apply to and relate these sorts.

Functions on Locations

If ℓ is a pair location and s a state,

$$:left(\ell, s) \quad \text{and} \quad :right(\ell, s)$$

are locations, called the *left* and *right* components of ℓ.

If a is an atom and s a state, and if a has some location stored in it (i.e., a is bound), then

$$:store(a, s)$$

is the location stored in a.

Functions on Abstract Trees

We assume that we have the usual functions on abstract trees: the tree constructor $t_1 \bullet t_2$, the functions $left(t)$ and $right(t)$, etc. Abstract lists are identified with abstract trees in the usual way: the list $(t_1, t_2, \ldots, t_n)$ is identified with the tree

$$t_1 \bullet (t_2 \bullet \cdots \bullet (t_n \bullet nil) \cdots).$$

The append function $t_1 \circ t_2$ and the reverse function $rev(t)$ are defined in the case in which t_1 and t are finite lists and t_2 is a list.

Functions on Fluents

To determine the state produced and the location returned by evaluating a fluent in a given state, we employ the *production* function ";" and the *return* function ":".

If s is a state and e a fluent,

$$s;e$$

is the state *produced* by evaluating e and

$$s:e$$

is the location *returned* by evaluating e (in state s).

We assume that the evaluation of atoms does not change the state and returns the location stored in the atom. This is expressed by the *atom* axioms,

$$w;u = w \qquad (production)$$

$$w:u = :store(u, w) \qquad (return)$$

for all states w and atoms u. (We use letters towards the end of the alphabet as variables. Variables in axioms have tacit universal quantification.) Evaluating the atom *nil* is assumed to return *nil* itself, i.e.,

$$:store(nil, w) = nil \qquad (nil)$$

for all states w.

If e_1 and e_2 are fluents,

$$e_1;;e_2$$

is the fluent obtained by *composing* e_1 and e_2. Evaluating $e_1;;e_2$ is the same as evaluating first e_1 and then e_2. This is expressed by the *composition* axioms,

$$w;(u_1;;u_2) = (w;u_1);u_2 \qquad (production)$$

$$w:(u_1;;u_2) = (w;u_1):u_2 \qquad (return)$$

for all states w and fluents u_1 and u_2.

We shall write $w;u_1;u_2;\ldots;u_n$ and $w;u_1;\ldots;u_{n-1}:u_n$ as abbreviations for $((w;u_1);u_2);\ldots;u_n$ and $(\ldots(w;u_1);\ldots;u_{n-1}):u_n$, respectively. For any positive integer i, we shall write $\overline{u}_i$ as an ab-

breviation for $u_1;;u_2;;\ldots;;u_i$, where $\overline{u}_1$ is taken to be u_1 itself. Thus (by the *production composition* axiom),

$$w;\overline{u}_i = w;u_1;\ldots;u_i$$

and (by the *return composition* axiom)

$$w{:}\overline{u}_i = w;u_1;\ldots;u_{i-1}{:}u_i.$$

The Linkage Axioms

Each function on fluents induces corresponding functions on states and on locations. We begin with some definitions.

A *fluent function* $f(e_1, \ldots, e_n)$ applies to fluents $e_1, \ldots, e_n$ and yields a fluent. A *state function* $h(\ell_1, \ldots, \ell_n, s)$ applies to locations $\ell_1, \ldots, \ell_n$ and a state s and yields a state. A *location function* $g(\ell_1, \ldots, \ell_n, s)$ applies to locations $\ell_1, \ldots, \ell_n$ and a state s and yields a location.

For each fluent function $f(e_1, \ldots, e_n)$, we introduce a corresponding state function $;f(\ell_1, \ldots, \ell_n, s)$ and location function $:f(\ell_1, \ldots, \ell_n, s)$. If f uses ordinary LISP evaluation mode, the three functions are linked by the following *linkage* axioms:

$$w;f(u_1, \ldots, u_n) = ;f(w{:}\overline{u}_1, \ldots, w{:}\overline{u}_n, w;\overline{u}_n) \qquad (\textit{production})$$

$$w{:}f(u_1, \ldots, u_n) = :f(w{:}\overline{u}_1, \ldots, w{:}\overline{u}_n, w;\overline{u}_n) \qquad (\textit{return})$$

for all states w and fluents $u_1, \ldots, u_n$. In other words, the state and location functions $;f$ and $:f$ describe the effects of the fluent function f after its arguments have been evaluated. The function $;f$ yields the state produced, and the function $:f$ yields the location returned, by the evaluation of f.

For example, for the fluent function *setleft* [conventionally written *rplaca*], we have the *production linkage* axiom

$$w;\textit{setleft}(u_1, u_2) = ;\textit{setleft}(w{:}u_1,\ w;u_1{:}u_2,\ w;u_1;u_2)$$

and the *return linkage* axiom

$$w{:}\textit{setleft}(u_1, u_2) = :\textit{setleft}(w{:}u_1,\ w;u_1{:}u_2,\ w;u_1;u_2),$$

for all states w and fluents u_1 and u_2. That is, to find the state produced and location returned by evaluating $\textit{setleft}(u_1, u_2)$ in state w, first evaluate u_1 in state w, then evaluate u_2 in the resulting state, and finally apply the corresponding state and location functions $;\textit{setleft}$ and $:\textit{setleft}$ in the new resulting state. The axioms that describe $;\textit{setleft}$ and $:\textit{setleft}$ are given in the next section.

While the fluent function *setstore* [conventionally, *set*] does adhere to ordinary LISP evaluation mode, the fluent function *setstoreq* [conventionally, *setq*] does not; it requires that its first argument be an atom and does not evaluate it. For this function, the *production linkage* axiom is

$$w;\textit{setstoreq}(u, v) = ;\textit{setstore}(u,\ w{:}v,\ w;v)$$

and the *return linkage* axiom is

$$w{:}\textit{setstoreq}(u, v) = :\textit{setstore}(u,\ w{:}v,\ w;v),$$

for all states w, atoms u, and fluents v. These axioms take into account the fact that evaluating an atom has no side effects.

As an informal abbreviation, we shall sometimes use "$s{:};e$" as an abbreviation for the string "$s{:}e,\ s;e$", and "$:;f(e_1, \ldots, e_n)$" as an abbreviation for the string "$:f(e_1, \ldots, e_n),\ ;f(e_1, \ldots, e_n)$".

Describing LISP Operators

Although we regard LISP programs as computing functions on fluents, they are best described by providing axioms for the corresponding state and location functions.

A fluent function $f(e_1, \ldots, e_n)$ is said to be *applicative* if its evaluation produces no side-effects other than those produced by evaluating its arguments $e_1, \ldots, e_n$. This is expressed by the axiom

$$;f(x_1, \ldots, x_n, w) = w \qquad (\textit{applicative})$$

for all locations $x_1, \ldots, x_n$ and states w. It follows that

$$\begin{aligned} & w;f(u_1, \ldots, u_n) \\ &= ;f(w{:}\overline{u}_1, \ldots, w{:}\overline{u}_n, w;\overline{u}_n) \\ &\quad \text{(by the } \textit{production linkage} \text{ axiom for } f) \\ &= w;\overline{u}_n \\ &\quad \text{(by the } \textit{applicative} \text{ axiom for } f) \end{aligned}$$

for all fluents $u_1, \ldots, u_n$ and states w.

For example, the fluent functions *left* and *right* [conventionally written *car* and *cdr*, respectively] are applicative, that is,

$$;\textit{left}(x, w) = w \quad \text{and} \quad ;\textit{right}(x, w) = w$$

for all locations x and states w. It follows by the above reasoning that

$$w;\textit{left}(u) = w;u \quad \text{and} \quad w;\textit{right}(u) = w;u$$

for all fluents u and states w.

The fluent function *setleft* alters the left component of its first argument to contain its second argument. This is expressed precisely by the *primary production* axiom for *setleft*,

$$:\textit{left}\,(x_1,\ ;\textit{setleft}(x_1, x_2, w)) = x_2 \qquad (\textit{primary production})$$

for all pair locations x_1, locations x_2, and states w.

The function *setleft* returns its first argument; this is expressed by the *return* axiom for *setleft*,

$$:\textit{setleft}(x_1, x_2, w) = x_1 \qquad (\textit{return})$$

for all pair locations x_1, locations x_2, and states w.

We must also provide *frame* axioms indicating that the function *setleft* does not alter anything but the left component of its first argument; namely,

$$;\textit{setleft}(x_1, x_2, w){:}u = w{:}u \qquad (\textit{atom})$$

$$\begin{aligned} &\textit{if not } (x_1 = y) \\ &\textit{then } :\textit{left}(y, ;\textit{setleft}(x_1, x_2, w)) = :\textit{left}(y, w) \qquad (\textit{left frame}) \\ &:\textit{right}(y, ;\textit{setleft}(x_1, x_2, w)) = :\textit{right}(y, w) \qquad (\textit{right frame}) \end{aligned}$$

for all pair locations x_1 and y, locations x_2, atoms u, and states w.

The above axioms give properties of the state and location functions $;\textit{setleft}$ and $:\textit{setleft}$. Properties of the corresponding fluent function *setleft* can now be deduced from the *linkage* axioms.

The function *setright* [conventionally, *rplacd*] is treated analogously.

Locations and Abstract Trees

We think of each location as representing an abstract (finite or infinite) tree. While we describe LISP functions at the state and location level, it is often natural to express the specifications for LISP programs at the abstract tree level. In this section we explore the relationship between locations and abstract trees. A formalization of abstract trees is discussed in Mason [86].

We introduce a function $\textit{decode}(\ell, s)$ mapping each location into the abstract tree it represents. If u is an atom,

$$\textit{decode}(u, w) = u \qquad (\textit{atom})$$

for all states w. Thus each atom represents itself.

Furthermore, if x is a pair location,

$$decode(x, w) = decode(:left(x, w), w) \bullet decode(:right(x, w), w) \qquad (pair)$$

for all states w.

Specification constructs

Certain concepts are used repeatedly in the description of imperative-LISP programs. We give a few of these here, without defining them formally. We follow Mason [86] in our terminology.

- *spine*

The spine of a pair location ℓ in state s, written

$$spine(\ell, s),$$

is the set of locations we can reach from ℓ by following a trail of *right* pointers (not *left* or *store* pointers).

- *finiteness*

We say that the spine of location ℓ is finite in state s, written

$$finite(\ell, s),$$

if we can get from location ℓ to an atom by following a trail of *right* pointers (not *left* or *store* pointers). A location is finite if and only if its spine is a finite set.

- *list*

We say that the location ℓ represents a list in state s, written

$$list(\ell, s),$$

if either the spine of ℓ is infinite or we can get from ℓ to the atom *nil* by following a trail of *right* pointers.

- *accessibility*

A location ℓ is said to *access* a location m in state s, written

$$access(\ell, m, s),$$

if it is possible to get from ℓ to m by following a trail of *left* or *right* pointers (not *store* pointers). We shall also say that m is *accessible* from ℓ.

- *spine accessibility*

We say that the *spine of* location ℓ *accesses the spine of* location m in state s, written

$$spine\text{-}access(\ell, m, s),$$

if some element in the spine of ℓ accesses some element in the spine of m in state s.

- *purity*

We say that the location ℓ is *pure* in state s, written

$$pure(\ell, s),$$

if no element of the spine of ℓ is accessible from the left component of some element of the spine of ℓ itself. Otherwise, the location ℓ is said to be *ingrown*.

Abstract Properties of LISP Operators

While LISP functions are most concisely defined by giving their effects on locations, their most useful properties often describe their effects on the abstract lists and trees represented by these locations. For example, we can establish the *abstract* property of the function *cons*,

$$decode(:;cons(x, y, w)) = decode(x, w) \bullet decode(y, w) \qquad (abstract)$$

which relates *cons* to the abstract tree constructor $\bullet$.

Often, the properties we expect do not hold unless certain stringent requirements are met. For example, the *abstract* property of the function *setright* is

$$\begin{aligned} &if \quad pair(x) \ and \\ &\quad\ not\ access(:left(x, w), x, w) \ and \\ &\quad\ not\ access(y, x, w) \\ &then \ decode(:;setright(x, y, w)) = \\ &\quad\ left(decode(x, w)) \bullet decode(y, w) \end{aligned} \qquad (abstract)$$

where the relation *pair* characterizes the pair locations. That is, the function *setright* "normally" returns the result of a *left* operation followed by a tree construction. However, we require that x must be inaccessible from $:left(x, w)$; otherwise, in altering the right component of x, we inadvertently alter the abstract tree represented by the left component of x. Also, x must be inaccessible from y; otherwise, in altering the right component of x, we inadvertently alter the abstract tree represented by y.

Many errors in imperative programming occur because people assume that the *abstract* properties hold but forget the conditions they require.

Specification of Programs

Each LISP program is applied to an initial input location ℓ_0. The resulting fluent is evaluated in an initial state s_0, produces a new state s_f, and returns an output location ℓ_f. The specification for a LISP program may thus be expressed as a sentence

$$\mathcal{Q}[\ell_0, s_0, \ell_f, s_f].$$

The program to be constructed computes a fluent function, which does not apply to locations directly; it applies to an *input parameter* a, an atom that (in normal evaluation mode) contains the input location, that is, $s_0{:}a = \ell_0$. When the program is evaluated, its actual argument, which is a fluent, is evaluated first. The location it returns is stored in the parameter a. (This is easily extended for programs which take more than one argument.)

Furthermore, the computed function does not yield a state or location itself, but a fluent z. Evaluating z in the initial state s_0 produces the final state and returns the output location, that is,

$$s_0{:}z = \ell_f \quad \text{and} \quad s_0{;}z = s_f.$$

To construct the program, therefore, we prove the theorem

$$(\forall a)(\forall s_0)(\exists z)\mathcal{Q}[s_0{:}a,\ s_0,\ s_0{:}z,\ s_0{;}z].$$

In other words, we prove the existence of a fluent z that, when evaluated in the initial state, will produce a final state and yield an output location meeting the specified conditions.

For example, suppose we want to specify a destructive list-reversing program. In terms of its principle *abstract* property, we may specify the desired program by the sentence

$$\mathcal{Q}[s_0, \ell_0, s_f, \ell_f]\colon \quad decode(\ell_f, s_f) = rev(decode(\ell_0, s_0)).$$

In other words, the list represented by the location ℓ_f after evaluation of the program is to be the reverse of the list represented by ℓ_0 before. For a moment, we forget about the conditions required to make this possible.

The theorem we must prove is accordingly

$$(\forall a)(\forall s_0)(\exists z)[decode(s_0{:}z,\ s_0{;}z) = rev(decode(s_0{:}a,\ s_0))].$$

The Deductive System

The system we employ to prove our theorems is an adaptation of the deductive-tableau system we use to derive applicative programs (Manna and Waldinger [80], [87a]). The adaptation to imperative

programs mimics our development of a deductive system for automatic planning (Manna and Waldinger [87b]). Because we shall only informally present a segment of the program derivation in this paper, we do not describe the system in detail. A complete description appears in the report version of this paper.

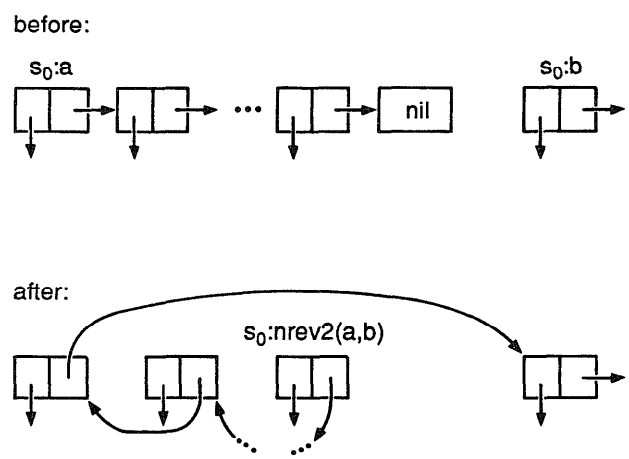
The In-place Reverse Program

At the risk of spoiling the suspense, let us present the final program we shall obtain from the derivation:

$$nrev(a) \Leftarrow nrev2(a, nil)$$

$$nrev2(a, b) \Leftarrow \begin{cases} if\ null(a) \\ then\ b \\ else\ nrev2(right(a), setright(a, b)). \end{cases}$$

This is an in-place reverse, used as an example by Mason [86]. The program *nrev* is defined in terms of a more general program *nrev2*, which has the effect of reversing the list a and appending the result to the list b. The consequence of applying the program *nrev2* is illustrated in the following figure:



Note that the pointers in the spine of $s_0{:}a$ have been reversed.

The principal condition in the specification for *nrev2* is

$$decode(\ell_f, s_f) = rev(decode(\ell_0, s_0)) \,\square\, decode(m_0, s_0),$$

where ℓ_0 and m_0 are the two input locations, In other words, the abstract list represented by the location ℓ_f after the evaluation is to be the abstract list obtained by reversing the list represented initially by ℓ_0 and appending the result to the list represented initially by m_0.

The program *nrev2* we derive does not satisfy the above specification in all cases. We must require several input conditions that ensure that our given lists are reasonably well behaved. We impose

- the *list* conditions
  $list(\ell_0, s_0)$ and $list(m_0, s_0)$,
  that ℓ_0 and m_0 initially represent abstract lists.
- the *finiteness* condition
  $finite(\ell_0, s_0)$,
  i.e., that the list ℓ_0 initially represents is finite; otherwise, *nrev2* would not terminate.
- the *purity* condition
  $pure(\ell_0, s_0)$,
  i.e., that the spine of ℓ_0 is not initially accessible from any of the left components of spine elements; otherwise, in altering the pointers in the spine, we would inadvertently be altering the elements of the list represented by ℓ_0.
- the *isolation* condition
  $not\ spine\text{-}access(m_0, \ell_0, s_0)$,
  i.e., that the spine of ℓ_0 is not initially accessible from (the spine of) m_0; otherwise, in altering the spine of ℓ_0, we would inadvertently be altering the list represented by m_0.

These are reasonable enough conditions, but to complete the derivation we must make them explicit. (Similarly, we must impose the *list* condition for ℓ_0, and the *finiteness* and *purity* conditions, on the specification for *nrev* itself.)

The full specification for *nrev2* is thus

$$\begin{aligned} if\ & list(\ell_0, s_0)\ and\ list(m_0, s_0)\ and \\ & finite(\ell_0, s_0)\ and \\ & pure(\ell_0, s_0)\ and \\ & not\ spine\text{-}access(m_0, \ell_0, s_0) \\ then\ & decode(\ell_f, s_f) = \\ & rev(decode(\ell_0, s_0)) \,\square\, decode(m_0, s_0), \end{aligned}$$

and the theorem we must prove is

$$\begin{aligned} if\ & list(s_0{:}a, s_0)\ and\ list(s_0{:}b, s_0)\ and \\ & finite(s_0{:}a, s_0)\ and \\ & pure(s_0{:}a, s_0)\ and \\ & not\ spine\text{-}access(s_0{:}b, s_0{:}a, s_0) \\ then\ & decode(s_0{:};z) = \\ & rev(decode(s_0{:};a)) \,\square\, decode(s_0{:};b). \end{aligned}$$

(Here we have dropped quantifiers by skolemization.)

We do not have time to present the full derivation of the program *nrev* here, so let us focus our attention on the most interesting point, in which the pointer reversal is introduced into *nrev2*.

Using the *pair* axiom for the *decode* function, properties of abstract lists, and the input conditions, we may transform our goal into

$$\begin{aligned} & pair(s_0{:}a)\ and \\ & decode(s_0{:};z) = rev(decode(s_0{:};right(a))) \,\square \\ & \qquad \boxed{left(decode(s_0{:};a)) \bullet decode(s_0{:};b)} \end{aligned}$$

We omit the details of how this was done.

At this point, we invoke the *abstract* property of *setright*,

$$\begin{aligned} if\ & pair(x)\ and \\ & not\ access(:left(x, w), x, w)\ and \\ & not\ access(y, x, w) \\ then\ & decode(:;setright(x, y, w)) = \\ & \boxed{left(decode(x, w)) \bullet decode(y, w)}. \end{aligned}$$

The boxed subsentence of the above property is equationally unifiable with the boxed subsentence of our goal; a unifying substitution is $\theta : \{x \leftarrow s_0{:}a, y \leftarrow s_0{:}b, w \leftarrow s_0\}$.

To see that θ is indeed an equational unifier, observe that

$$
\begin{aligned}
&(left(decode(s_0{:}{;}a)) \bullet decode(s_0{:}{;}b))\theta \\
&= left(decode(s_0{:}a,\ s_0{;}a)) \bullet decode(s_0{:}b,\ s_0{;}b) \\
&\quad \text{(by our abbreviation)} \\
&= left(decode(s_0{:}a,\ s_0)) \bullet decode(s_0{:}b,\ s_0) \\
&\quad \text{(by the } \textit{production atom} \text{ axiom)} \\
&= (left(decode(x,\ w)) \bullet decode(y,\ w))\theta.
\end{aligned}
$$

This reasoning is carried out by the *equational unification* algorithm (Fay [79], see also Martelli and Rossi [86]).

We can thus use the property to deduce that it suffices to establish the goal

$$
\begin{aligned}
&pair(s_0{:}a) \ \textit{and} \\
&not\ access(s_0{:}left(a),\ s_0{:}a,\ s_0) \ \textit{and} \\
&not\ access(s_0{:}b,\ s_0{:}a,\ s_0) \ \textit{and} \\
&decode(s_0{:}{;}z) \ = \ rev(decode(s_0{:}{;}right(a))) \ \square \\
&\qquad\qquad\qquad\quad decode(s_0{:}{;}setright(a, b)).
\end{aligned}
$$

Formally this reasoning is carried out by the *equality replacement* rule.

The second conjunct of the goal, that $s_0{:}a$ is inaccessible from $s_0{:}left(a)$, is a consequence of the *purity* input condition on $s_0{:}a$. The third conjunct, that $s_0{:}a$ is inaccessible from $s_0{:}b$, is a consequence of the *isolation* input condition. These deductions can be made easily within the system. Hence we are left with the goal

$$
\begin{aligned}
&pair(s_0{:}a) \ \textit{and} \\
&decode(s_0{:}{;}z) \ = \ rev(decode(s_0{:}{;}right(a))) \square \\
&\qquad\qquad\qquad\quad decode(s_0{:}{;}setright(a, b)).
\end{aligned}
$$

Now we may use induction to introduce the recursive call into the program *nrev2*. We omit how this is done. The complete program derivation is described in the report version of this paper.

The program we have obtained is an in-place reverse, which does not use any additional space. Of course, nothing in the derivation process ensures that the program we obtain is so economical. Other, more wasteful programs meet the same specification.

If we want to guarantee that no additional storage is required, we must include that property in the specification. More precisely, we can define a function *space*(e, s) that yields the number of additional locations (*cons* cells and *gensyms*) required to evaluate fluent e in state s. We may then include the new condition

$$space(z, s_0) = 0$$

in the theorem to be proved.

We could then derive the same in-place reverse program *nrev* but we could not derive the more wasteful ones. Our derivation for *nrev* would be longer, but would include a proof that the derived program uses no additional space.

Discussion

The ultimate purpose of this work is the design of automatic systems capable of the synthesis of imperative programs. By performing detailed hand-derivations of sample imperative programs, we achieve a step in this direction.

First of all, we ensure that the system is expressive enough to specify and derive the program we have in mind. But it is not enough that the derivation be merely possible. If the derivation requires many gratuitous, unmotivated steps, it may be impossible for a person or system to discover it unless the final program is known in advance. Such a system may be useful to verify a given program but hardly to synthesize a new one.

Of course, the fact that we can construct a well-motivated proof by hand does not guarantee that an automatic theorem prover will discover it. We expect, however, that the proofs we require are not far beyond the capabilities of existing systems. Looking at many hand derivations assists us in the design of a theorem prover capable of finding such derivations, work that is still underway. Close examination of many proofs suggests what rules of inference and strategies are appropriate to discover them.

Acknowledgments

The authors would like to thank Tom Henzinger and Ian Mason for valuable discussions and careful reading of the manuscript, and Evelyn Eldridge-Diaz for TEXing many versions.

References

Burstall [69] R. M. Burstall, Formal description of program structure and semantics in first-order logic, *Machine Intelligence 5* (B. Meltzer and D. Michie, editors), Edinburgh University Press, Edinburgh, Scotland, 1969, pp. 79–98.

Fay [79] M. Fay, First-order unification in an equational theory, *Proceedings of the Fourth Workshop on Automated Deduction*, Austin, Texas, Feb. 1979, pp. 161–167.

Green [69] C. C. Green, Application of theorem proving to problem solving, *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington, D.C., May 1969, pp. 219–239.

Martelli and Rossi [86] A. Martelli and G. Rossi, An algorithm for unification in equational theories, *Proceedings of the Third Symposium on Logic Programming*, Salt Lake City, Utah, Sept. 1986.

McCarthy [63] J. McCarthy, Situations, actions, and causal laws, technical report, Stanford University, Stanford, Calif., 1963. Reprinted in *Semantic Information Processing* (Marvin Minsky, editor), MIT Press, Cambridge, Mass., 1968, pp. 410–417.

Manna and Waldinger [80] Z. Manna and R. Waldinger, A deductive approach to program synthesis, *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 1, Jan. 1980, pp. 90–121.

Manna and Waldinger [81] Z. Manna and R. Waldinger, Problematic features of programming languages: a situational-logic approach, *Acta Informatica*, Vol. 16, 1981, pp. 371–426.

Manna and Waldinger [87a] Z. Manna and R. Waldinger, The origin of the binary-search paradigm, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, Calif., Aug. 1985, pp. 222–224. Also in *Science of Computer Programming* (to appear).

Manna and Waldinger [87b] Z. Manna and R. Waldinger, How to clear a block: a theory of plans, *Journal of Automated Reasoning* (to appear).

Mason [86] I. A. Mason, Programs via transformation, *Symposium on Logic in Computer Science*, Cambridge, Mass., June 1986, pp. 105–117.

Waldinger and Lee [69] R. J. Waldinger and R. C. T. Lee, PROW: A step toward automatic program writing, *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington, D.C., May 1969, pp. 241–252.