3

Proving Compiler Correctness in a Mechanized Logic

R. Milner and R. Weyhrauch
Computer Science Department
Stanford University

Abstract
We discuss the task of machine-checking the proof of a simple compiling algorithm. The proof-checking program is LCF, an implementation of a logic for computable functions due to Dana Scott, in which the abstract syntax and extensional semantics of programming languages can be naturally expressed. The source language in our example is a simple ALGOL-like language with assignments, conditionals, whiles and compound statements. The target language is an assembly language for a machine with a pushdown store. Algebraic methods are used to give structure to the proof, which is presented only in outline. However, we present in full the expression-compiling part of the algorithm. More than half of the complete proof has been machine checked, and we anticipate no difficulty with the remainder. We discuss our experience in conducting the proof, which indicates that a large part of it may be automated to reduce the human contribution.

INTRODUCTION

In this paper we describe our experiences in trying to use a mechanized version of a logic for computable functions, LCF (Milner 1972a,b; Weyhrauch and Milner 1972), to express and formally prove the correctness of a compiler. This logic is based on the theory of the typed lambda calculus, augmented by a powerful induction rule, suggested in private communications with Dana Scott. More particularly: (1) We show how to define in LCF an extensional semantics for our target language T which contains an unrestricted jump instruction. This definition provides, in a direct manner, a single recursive definition MT for the semantics of a program. This contrasts with the approach of McCarthy (1963) where each program is assigned a set of mutually recursive function definitions as its semantics. (2) We give a description, using algebraic methods, of the proof of the correctness of a compiling algorithm for a simple ALGOL-like source language S. (3) We present in its entirety a machine-checked proof of the correctness of an algorithm for compiling expressions. We call this the McCarthy-Painter lemma, as it is essentially the algorithm proved correct by them (McCarthy and Painter 1967).

The question of rigorous verification of compilers has already been the subject of considerable research. As we mentioned, McCarthy and Painter have given a proof for an expression-compiling algorithm. Kaplan (1967) and Painter (1967) have verified compiling algorithms for source languages of about the same complexity as ours; in both cases the source language contains jump instructions, whereas our source language with conditional and while statements is in the spirit of the 'goto-less' programming advocated by Djikstra (1968) and others. Burstall and Landin (1969) first explored the power of algebraic methods in verifying a compiler for expressions, and in pursuing this exploration with a more powerful language we have been helped by discussions with Lockwood Morris, whose forthcoming doctoral thesis (1972) is concerned with this topic. London (1971, 1972) has given a rigorous proof of two compilers for a LISP subset. All these authors have looked forward to the possibility of machine-checked compiler proofs, and Diffie (1972) has successfully carried out such a proof for the expression compiler of McCarthy and Painter, using a proof-checking program for the First Order Predicate Calculus written by him. We believe that LCF has advantages over First Order Predicate Calculus for the expression and proof of compiler correctness, and our current paper is in part an attempt to justify this belief. Briefly, the advantages of LCF consist in its orientation towards partial computable functions, and functions of higher type. For example, we consider that the meaning of a program is a partial computable function from states to states, where a state is conveniently represented (at least for the simple source language which we consider) as a function from names to values.

THE SOURCE LANGUAGE S

Expressions in S are built up from names by the application of binary operators. The collection of such operators is an entirely arbitrary but fixed set. Thus expressions are defined in LCF by the equations

$$\begin{aligned}&iswfse \equiv \alpha f.\ wfsefun(f),\\&wfsefun \equiv \lambda f\ e.\ type(e) =\_N \rightarrow TT,\\&\qquad type(e) =\_E \rightarrow isop(opof(e)) \wedge f(arg1of(e)) \wedge f(arg2of(e)),\\&\qquad UU,\end{aligned}$$

that is, well-formed source expressions are just those individuals on which *iswfse* (which is meant to abbreviate 'is well-formed source expression') takes the value TT. Here '$\alpha f.\ F(f)$' denotes the least fixed point of the functional F, and 'TT', 'UU' denote the truth-values true and undefined respectively. The '$\rightarrow$' denotes the McCarthy conditional operator; $(p \rightarrow q, r)$ means if p

and q, else r, and in LCF is undefined if p is undefined. A more detailed description of the terms of LCF can be found in Milner (1972a). The 'type' of an object is determined axiomatically, for example $type(e)=\_N$ is true just in case e is a name.

There are assignment, conditional, and while statements as well as compound statements formed by pairing any two statements. A well-formed source program is just any statement, that is,

$$\begin{aligned}
&iswfs \equiv \alpha f.\ wfsfun(f),\\
&wfsfun \equiv \lambda fp.\, type(p)=\_A \rightarrow type(lhsof(p))=\_N \wedge iswfse(rhsof(p)),\\
&\quad type(p)=\_C \rightarrow iswfse(ifof(p)) \wedge f(thenof(p)) \wedge f(elseof(p)),\\
&\quad type(p)=\_W \rightarrow iswfse(testof(p)) \wedge f(bodyof(p)),\\
&\quad type(p)=\_CM \rightarrow f(firstof(p)) \ \wedge\ f(secondof(p)),\\
&\quad UU.
\end{aligned}$$

Of course in LCF programs are expressed by means of an abstract syntax for S, using appropriate constructors and selectors, some of which appear in the equation above. A complete list of the axioms for the abstract syntax is found below in Appendix 1.

We consider that the meaning of a program in S is a statefunction, that is, a function that maps states on to states, where a state is a function from the set of names to the set of values. Thus the meaning function MSE for expressions is a function which 'evaluates' an expression in a state.

$$\begin{aligned}
&MSE \equiv \alpha M.\ \lambda e\, sv.\\
&\quad type(e)=\_N \rightarrow sv(e),\\
&\quad type(e)=\_E \rightarrow\\
&\qquad (MOP(opof(e)))(M(arg1of(e), sv), M(arg2of(e), sv)),\\
&\quad UU.
\end{aligned}$$

That is, to give the meaning of an expression e in a state sv, we compute as follows: if e is a name, look up e in the state, that is, evaluate $sv(e)$; if e is a compound expression, $opof(e)$ is the selector which computes from e its operator symbol, and MOP is a function which maps an operator symbol onto a binary function, which is then applied to the meanings of the subexpressions of e.

The following combinators are used in defining the semantics MS of S:

$$\begin{aligned}
&ID \equiv \lambda x.\ x,\\
&WHILE \equiv \alpha g.\ \lambda q f.\ COND(q, f \otimes g(q,f), ID),\\
&COND \equiv \lambda q f g\ s.\ (q(s) \rightarrow f(s), g(s)),\\
&\otimes \equiv \lambda f g\ x.\ g(f(x)).\\
&SCMPND \equiv \lambda f g.\, f \otimes g,\\
&SCOND \equiv \lambda e f g.\ COND(MSE(e) \otimes true, f, g).\\
&SWHILE \equiv \lambda e f.\ WHILE(MSE(e) \otimes true, f).
\end{aligned}$$

MS is now defined as

$$\begin{aligned}
&MS \equiv \alpha M.\ \lambda p.\\
&\quad type(p)=\_A \rightarrow [\lambda sv\ m.\ m=lhsof(p) \rightarrow MSE(rhsof(p), sv), sv(m)],\\
&\quad type(p)=\_C \rightarrow SCOND(ifof(p))(M(thenof(p)), M(elseof(p))),
\end{aligned}$$

$type(p) = {}_{-}W \rightarrow SWHILE(testof(p))(M(bodyof(p))),$
$type(p) = {}_{-}CM \rightarrow SCMPND(M(firstof(p)), M(secondof(p))),$
$UU.$

Several things should be noted about this definition. First of all there are no boolean expressions *per se*. Their place is taken by the function '*true*' which yields 'TT' just on those values which we wish to let represent true. This corresponds to the LISP convention, for example, where NIL is false and any other expression is considered true. Secondly, if the program p is an assignment (that is, $type(p) = {}_{-}A$) it has been treated asymmetrically from the others. The reason for this will appear below.

THE TARGET LANGUAGE T

Our target language T is an elementary assembly language which contains unrestricted jumps and manipulates a stack. We consider the meaning of a program p in T to be a storefunction, that is, a function from stores to stores. A store sp is a pair consisting of a state sv and a list pd called the pushdown. We use '|' as the constructor for pairing a state and a pushdown, and '*svof*', '*pdof*' as the respective selectors. The following axioms hold:

$\forall sv\, pd.\quad svof(sv|pd) \equiv sv,$
$\forall sv\, pd.\quad pdof(sv|pd) \equiv pd,$
$svof(UU) \equiv UU,$
$pdof(UU) \equiv UU,$
$UU|UU \equiv UU.$

In T an instruction is a pair, whose head is the type of the instruction and whose tail is either a name m, an operator symbol o, or a natural number i. We assume that the set of operator symbols of T contains that of S. The instructions are:

Instruction		*Meaning*
(head)	(tail)	
JF	i	If head of pd is false, jump to label i, otherwise proceed to next instruction. In either case delete head of pd.
J	i	Jump to label i.
$FETCH$	m	Look up value of m in sv, and place it on top of pd.
$STORE$	m	Assign head of pd to m in sv, and delete head of pd.
$LABEL$	i	Serves only to label next instruction.
DO	o	Apply $MOP(o)$ – that is the binary function denoted by o – to the top two elements of pd, and replace them by the result.

We use '&' for the pairing operation and '*hd*', '*tl*' as the selectors for pairs. These, together with *null*, NIL and @ (*append*) are the conventional list-processing operations. Thus the pair whose members are JF and i is formally written $(JF\&i)$. We give the axioms for lists in Appendix 1.

By a program we mean a list of instructions. Unfortunately the existence of labels in T allows the meaning of such a program to be undetermined; for example, there might be two instructions $(LABEL\&6)$ in the list. To which one is $(JF\&6)$ to go? Although there are many alternatives we have chosen the following. A program p is well-formed if (i) the set L of numbers appearing in label statements forms an initial segment of the natural numbers; (ii) for each $n \in L$, $(LABEL\&n)$ occurs only once in p, and (iii) the set of numbers occurring in J and JF instructions is a subset of L, that is, there is no instruction which tries to jump to a non-existent label. These properties are guaranteed by the following definition of *iswft*:

$iswft \equiv \lambda p.\ iswft1(count(p), p),$
$iswft1 \equiv \alpha w.\ \lambda n\, p.\ n=\emptyset \to TT,\ occurs1(n-1, p) \to w(n-1, p),\ FF,$
$count \equiv \alpha c.\ \lambda p.\ null(p) \to \emptyset,$
$\quad (hd(hd(p))=J) \vee (hd(hd(p))=JF) \vee (hd(hd(p))=LABEL) \to$
$\quad\quad \max(tl(hd(p))+1,\ c(tl(p))),\ c(tl(p)),$
$occurs \equiv \alpha oc.\ \lambda n\, p.\ (count(p) \leqslant n) \to FF,$
$\quad hd(hd(p))=LABEL \to tl(hd(p))=n \to TT,\ oc(n, tl(p)),\ oc(n, tl(p)),$
$occurs1 \equiv \alpha oc1.\ \lambda n\, p.\ count(p) \leqslant n \to FF,$
$\quad hd(hd(p))=LABEL \to tl(hd(p))=n \to$
$\quad\quad \neg(occurs(n, tl(p))),\ oc1(n, tl(p)),\ oc1(n, tl(p)).$

$count(p)$ computes the least natural number not appearing in a program p. $occurs(n, p)$ yield true if n occurs in p, false otherwise, and $occurs1(n, p)$ checks that a label occurs exactly once. $iswft1(n, p)$ checks that for every natural number m, $\emptyset \leqslant m < n$, the instruction $(LABEL\&m)$ occurs exactly once. Thus $iswft(p)$ is as described above.

We can now define MT.

$MT \equiv \lambda p.\ MT1(p, p),$
$MT1 \equiv [\alpha f.\ [\lambda p\, q.\ null(q) \to [\lambda sp.\ svof(sp) \mid pdof(sp)],$
$\quad hd(hd(q))=JF \to [\lambda sp.\ (true(hd(pdof(sp))) \to f(p, tl(q)),$
$\quad\quad f(p, find(p, tl(hd(q)))))(svof(sp) \mid tl(pdof(sp)))],$
$\quad hd(hd(q))=J \to f(p, find(p, tl(hd(q)))),$
$\quad hd(hd(q))=FETCH \to [\lambda sp.\ svof(sp) \mid (svof(sp)(tl(hd(q)))\ \&$
$\quad\quad pdof(sp))] \otimes f(p, tl(q)),$
$\quad hd(hd(q))=STORE \to [\lambda sp.[\lambda m.\ m=tl(hd(q)) \to hd(pdof(sp)),$
$\quad\quad svof(sp)(m)] \mid tl(pdof(sp))] \otimes f(p, tl(q)),$
$\quad hd(hd(q))=DO \to [\lambda sp.\ svof(sp) \mid$
$\quad\quad (MOP(tl(hd(q)))(hd(tl(pdof(sp))), hd(pdof(sp)))$
$\quad\quad \&\ tl(tl(pdof(sp))))] \otimes f(p, tl(q)),$
$\quad hd(hd(q))=LABEL \to f(p, tl(q)),$
$\quad\quad UU]],$

$$\begin{aligned}&\mathit{find}\equiv[\alpha f.\ [\lambda p\ n.\ \mathit{null}(p)\rightarrow UU,\ hd(hd(p))=LABEL\rightarrow\\ &\qquad tl(hd(p))=n\rightarrow tl(p), f(tl(p),n), f(tl(p),n)]].\end{aligned}$$

The auxiliary function *find* has as arguments a program p and a label n and if $(LABEL\&n)$ occurs in p it yields that terminal sublist of p immediately following $(LABEL\&n)$, otherwise it yields undefined. One should note that the definition of $MT1$ could be parameterized by a variable in place of find thus allowing any computable method of 'finding' the appropriate instruction to jump to. This corresponds to choosing different notions of the semantics of a program. For example, if one allowed jumps to nonexistent labels '*find*' might simply compute the program NIL when such a jump was attempted. This amounts to choosing the convention that a jump to a nonexistent label terminates the program. Many such conventions can be mimicked by an appropriate find function. For the other instructions the definition of $MT1$ follows their informal description quite closely.

THE COMPILER

Strictly speaking we do not prove the correctness of a compiler in this paper. What we prove is the correctness of a compiling algorithm, which we call '*comp*'. That is, a compiler is a syntactic object written in some programming language; we have not started with such an object and shown that its meaning (semantics) is '*comp*', but rather we have assumed that '*comp*' is indeed the meaning of some suitably chosen compiler.

Expressions are compiled by

$$\begin{aligned}&\mathit{compe}\equiv\alpha f.\ \mathit{compefun}(f),\\ &\mathit{compefun}\equiv\lambda f\ e.\\ &\qquad \mathit{type}(e)=\_N\rightarrow(FETCH\&e)\&NIL,\\ &\qquad \mathit{type}(e)=\_E\rightarrow f(\mathit{arg1of}(e))\ @\ f(\mathit{arg2of}(e))\ @\\ &\qquad\quad ((DO\&\mathit{opof}(e))\&NIL),\\ &\qquad UU.\end{aligned}$$

In order to define *comp* we use the following auxiliary functions:

$$\begin{aligned}&\mathit{shift}=\alpha sh.\ \lambda n\ p.\ \mathit{count}(p)=\emptyset\rightarrow p,\\ &\qquad (hd(hd(p))=J)\vee(hd(hd(p))=JF)\vee(hd(hd(p))=LABEL)\rightarrow\\ &\qquad (hd(hd(p))\&(tl(hd(p))+n))\&sh(n,tl(p)),\\ &\qquad hd(p)\&sh(n,tl(p)),\\ &\mathit{mktcmpnd}\equiv\lambda p\ q.\ p\ @\ \mathit{shift}(\mathit{count}(p),q)),\\ &\mathit{mktcond}\equiv\lambda e.\lambda p\ q.\ \mathit{compe}(e)\ @\\ &\qquad ((JF\&\mathit{count}(p))\&NIL)\ @\\ &\qquad p\ @\\ &\qquad ((J\&(\mathit{count}(p)+1))\&NIL)\ @\\ &\qquad ((LABEL\&\mathit{count}(p))\&NIL)\ @\\ &\qquad \mathit{shift}(\mathit{count}(p)+2,q)\ @\\ &\qquad ((LABEL\&(\mathit{count}(p)+1))\&NIL,\\ &\mathit{mktwhile}\equiv\lambda e.\lambda p.\ ((LABEL\&\mathit{count}(p))\&NIL)\ @\\ &\qquad \mathit{compe}(e)\ @\end{aligned}$$

$$\begin{array}{l}((JF\&(count(p)+1))\&NIL)@\\ p@\\ ((J\&count(p))\&NIL)@\\ ((LABEL\&(count(p)+1))\&NIL).\end{array}$$

shift(n, p) adds n to the integer in each label and jump instruction occurring in p. Using *shift* in the definitions of the other combinators then guarantees that when applied to well-formed objects, *mktcmpnd*, *mktcond* and *mktwhile* generate well-formed target programs. *Comp* is defined as

$$\begin{array}{l}comp \equiv \alpha f.\ compfun(f),\\ compfun \equiv \lambda f\, p.\\ \quad type(p) =\_A \rightarrow compe(rhsof(p))\,@((STORE\&lhsof(p))\&NIL),\\ \quad type(p) =\_C \rightarrow mktcond(ifof(p))(f(thenof(p)), f(elseof(p))),\\ \quad type(p) =\_W \rightarrow mktwhile(testof(p))(f(bodyof(p))),\\ \quad type(p) =\_CM \rightarrow mktcmpnd(f(firstof(p)), f(secondof(p))),\\ \quad UU.\end{array}$$

For well-formed source programs p the correctness of this compiler can be expressed as

$$(MS(p))(sv) \equiv svof((MT(comp(p)))(sv|NIL)).$$

This equation simply states that the result of executing a source program p on a state sv is the same as the state component of the store resulting from the execution of the compiled program $comp(p)$ on the store $sv|NIL$.

OUTLINE OF THE PROOF

Once we had defined the source and target languages and the compiling algorithm, and formulated the statement of compiler correctness, we proceeded to tackle the proof with the help of our proof-checking program LCF. The natural approach is to use structural induction on source programs. However, it was soon clear that the proof would be long and uninformative, and we became concerned not merely with carrying it out but also with giving it enough structure to make it intelligible. Observe that if we define

$$SIMUL\text{: storefunctions} \rightarrow \text{statefunctions}$$

by

$$SIMUL \equiv \lambda g.\ \lambda sv.\ svof(g(sv|NIL))$$

then the compiler correctness is equivalently stated by

$$MS \equiv comp \otimes MT \otimes SIMUL \qquad \text{(G1)}$$

where it is understood that both sides are restricted to the domain of well-formed source programs. Now this equation is equivalent to the commutativity of the diagram

$$\begin{tikzcd} S \arrow[r, "comp"] \arrow[d, "MS"'] & T \arrow[d, "MT"] \\ \hat{S} & \hat{T} \arrow[l, "SIMUL"'] \end{tikzcd}$$

where $\hat{S}$ and $\hat{T}$ are the sets of statefunctions and storefunctions respectively. This diagram suggests an algebraic approach; in fact, by defining appropriate operations on the sets S, $\hat{S}$, T, $\hat{T}$ we will show that with respect to these operations the mappings in the diagram are actually homomorphisms. Then our result will follow as an instance of a fundamental theorem of universal algebra, as we shall explain below. This algebraic approach gives our proof the desired structure; it is an open question whether a similar approach will extend to more complex languages.

We now introduce those few concepts of universal algebra that we need. These may be found in Cohn (1965); we take the liberty of giving somewhat less formal definitions than his, since our needs are simple. For a clear exposition of some of the concepts of universal algebra written for computer scientists, see also Lloyd (1972).

An operator domain Ω is a set of operator symbols each with an associated integer $n \geq \emptyset$, called its *arity*, which is the number of arguments taken by the operation that the symbol will denote in an algebra.

An Ω-algebra $\mathbf{A}$ is a set A, called the carrier of $\mathbf{A}$, together with an n-ary operation for each member of Ω with arity n. An Ω-algebra $\mathbf{B}$ is a subalgebra of $\mathbf{A}$ if its carrier B is a subset of A, its operations are the restrictions to B of $\mathbf{A}$'s operations, and B is closed under these operations.

Given any set of terms X, the Ω-word algebra $\mathbf{W}_\Omega(X)$ has as carrier the smallest set of terms containing X and such that if $a \in \Omega$ and a has arity n, and if $w1, w2, \ldots, wn$ are in $\mathbf{W}_\Omega(X)$, then the term $a(w1, w2, \ldots, wn)$ is in $\mathbf{W}_\Omega(X)$. This term-building operation is the operation corresponding to a in $\mathbf{W}_\Omega(X)$. The members of X are called the generators of $\mathbf{W}_\Omega(X)$.

We are concerned only with algebras for a certain fixed Ω. We need not trouble to name the members of Ω; Ω is only a device for defining a 1–1 correspondence between the operations of different algebras, and this correspondence will be clear from the way we define our algebras.

The fundamental theorem that we need – see Cohn (1965), p. 120, Theorem 2.6 – states that if $\mathbf{W}$ is a word algebra, then any mapping from the generators of $\mathbf{W}$ into the carrier of an algebra $\mathbf{A}$ extends in only one way to a homomorphism from $\mathbf{W}$ to $\mathbf{A}$. In our case the word algebra $\mathbf{W}$ is the algebra $\mathbf{S}$ of well-formed source programs, whose generators are the assignment statements and whose denumerably many operations are as follows:
(i) The binary operation *mkscmpnd*
(ii) For each well-formed source expression e, the binary operation *mkscond*(e)
(iii) For each such e, the unary operation *mkswhile*(e).

The second algebra $\mathbf{A}$ is the algebra $\hat{\mathbf{S}}$ of statefunctions, with operations as follows:
(i) The binary operation $SCMPND$
(ii) For each well formed source expression e, the binary operation $SCOND(e)$

(iii) For each such e, the unary operation $SWHILE(e)$.

Our main goal is (G1). We proceed to set up a tree of subgoals to attain this goal, and we will first state each of the subgoals in algebraic terms and then later list the formal statements of the subgoals as sentences of the logic *LCF*.

Our first level of subgoaling is justified by the fundamental theorem; to achieve (G1) it is sufficient to prove that

$MS:\mathbf{S}\rightarrow\hat{\mathbf{S}}$ is a homomorphism (G1.1)

$comp\otimes MT\otimes SIMUL:\mathbf{S}\rightarrow\hat{\mathbf{S}}$ is a homomorphism (G1.2)

and that

$MS\equiv comp\otimes MT\otimes SIMUL$,
when both sides are restricted to the generators of S. (G1.3)

Before going further, we must mention that in proving the formal statement of (G1) from the formal statements of (G1.1), (G1.2) and (G1.3) we do not rely on a formal statement in LCF of this fundamental theorem (though we believe that a restricted version of the theorem is indeed expressible and provable in LCF); rather we prove in LCF the relevant instance of that theorem. Thus we are using algebra as a guide to structuring our proof, not as a formal basis for the proof.

Now (G1.1) is a ready consequence of the definitions of MS, as the reader might suspect if he considers the operators of the algebras S and $\hat{\mathbf{S}}$. To achieve (G1.3), remember that the generators of S are the assignment statements, so we need a lemma which states that expressions – in particular, the right hand sides of assignments – compile correctly. This is expressed by:

$MT(compe(e))\equiv\lambda sp.\,(svof(sp)|(MSE(e,sp)\&pdof(sp)))$,
whenever e is a well-formed source expression. (G1.3.1)

This says that the target program for an expression places the value of the expression on top of the stack and leaves the store otherwise unchanged.

In order to prove (G1.2), it is helpful to introduce some further algebras. First, the algebra **T** of well-formed target programs whose operations are as follows:

(i) The binary operation *mktcmpnd*

(ii) For each well formed source expression e, the binary operation $mktcond(e)$

(iii) For each such e, the unary operation $mktwhile(e)$.

We have defined *mktcmpnd*, *mktcond* and *mktwhile* in the previous section. Second, we need the algebra $\hat{\mathbf{T}}$ of storefunctions whose operations are as follows:

(i) The binary operation $TCMPND$

(ii) For each well-formed source expression e, the binary operation $TCOND(e)$

(iii) For each such e, the unary operation $TWHILE(e)$,
where we define

$TCMPND \equiv \otimes$,

$TCOND \equiv \lambda e.\lambda f g.\ MT(compe(e)) \otimes COND(GET, POP \otimes f, POP \otimes g)$,

$TWHILE \equiv \lambda e.\lambda f.\ \alpha g.MT(compe(e)) \otimes COND(GET, POP \otimes f \otimes g, POP)$,

which in turn require the definitions

$GET \equiv pdof \otimes hd \otimes true$,

$POP \equiv \lambda sp.\ (svof(sp) \mid tl(pdof(sp)))$.

Consider these definitions for a moment. *POP* is a storefunction which just deletes the top stack element. *GET(sp)* yields the truth-value represented by the top stack element in the store *sp*. *MT(compe(e))* is a storefunction which simply places the value of the expression *e* on top of the stack. *COND(GET, POP⊗f, POP⊗g)* is a storefunction which examines the top stack element and then, after deleting this element, performs either the storefunction *f* or the storefunction *g*, according to the truth-value represented by it.

To achieve (G1.2) it is sufficient to prove

$comp: \mathbf{S} \to \mathbf{T}$ is a homomorphism (G1.2.1)

$MT: \mathbf{T} \to \hat{\mathbf{T}}$ is a homomorphism (G1.2.2)

$SIMUL: \hat{\mathbf{T}}' \to \hat{\mathbf{S}}$ is a homomorphism (G1.2.3)

where $\hat{\mathbf{T}}'$ is the subalgebra of $\hat{\mathbf{T}}$ induced by the homomorphism $comp \otimes MT: \mathbf{S} \to \hat{\mathbf{T}}$. (G1.2.1) is an immediate consequence of the definition of *comp*, provided that *comp* does indeed generate well-formed target programs from well-formed source programs. This is a consequence of the following two subgoals

comp takes the generators of **S** onto well-formed target programs (G1.2.1.1)

The operations of **T** preserve well-formedness of target programs (G1.2.1.2)

(G1.2.2) uses following general lemma about target programs, which we call the context-free lemma for *MT*, since it states that under certain conditions the execution of a sub-program is independent of its environment:

$MT1(p @ q' @ r, q' @ r) \equiv MT(q) \otimes MT1(p @ q' @ r, r)$,

provided that q is well-formed, $q' = shift(n, q)$ for some n, and $p @ q' @ r$ is also well-formed. (G1.2.2.1)

(G1.2.3) depends critically on the property of storefunctions g in $\hat{\mathbf{T}}'$ that

$svof(g(sv \mid pd)) \equiv svof(g(sv \mid NIL))$,

that is, the left hand side is independent of *pd*. This of course is not true for an arbitrary storefunction. Stated algebraically,

For all g in $\hat{\mathbf{T}}'$, $g \otimes svof \equiv svof \otimes SIMUL(g)$. (G1.2.3.1)

This concludes our attempt to structure the proof of the correctness of the compiler using algebraic methods. We have given eleven subgoals, most of which have a simple algebraic interpretation and therefore contribute significantly to the understanding of the proof as a whole.

We now list the goals as they are represented formally by sentences of LCF.

We have abbreviated $comp \otimes MT \otimes SIMUL$ by H throughout.

(G1) (Compiler correctness)

$iswfs(p) \equiv TT \vdash MS(p) \equiv SIMUL(MT(comp(p)))$

(G1.1) (MS is a homomorphism)

$iswfse(e) \equiv TT, iswfs(p) \equiv TT, iswfs(q) \equiv TT \vdash$
$MS(mkscmpnd(p, q)) \equiv SCMPND(MS(p), MS(q)),$
$MS(mkscond(e)(p, q)) \equiv SCOND(e)(MS(p), MS(q)),$
$MS(mkswhile(e)(p)) \equiv SWHILE(e)(MS(p))$

(G1.2) (H is a homomorphism)

$iswfse(e) \equiv TT, iswfs(p) \equiv TT, iswfs(q) \equiv TT \vdash$
$H(mkscmpnd(p, q)) \equiv SCMPND(H(p), H(q)),$
$H(mkscond(e)(p, q)) \equiv SCOND(e)(H(p), H(q),$
$H(mkswhile(e)(p)) \equiv SWHILE(e)(H(p))$

(G1.3) (MS and H agree on the generators of **S**)

$isname(n) \equiv TT, iswfse(e) \equiv TT \vdash$
$MS(mkassn(n, e)) \equiv H(mkassn(n, e))$

(G1.2.1) ($comp$ is a homomorphism)

$iswfse(e) \equiv TT, iswfs(p) \equiv TT, iswfs(q) \equiv TT \vdash$
$comp(mkscompnd(p, q)) \equiv mktcmpnd(comp(p), comp(q)),$
$comp(mkscond(e)(p, q)) \equiv mktcond(e)(comp(p), comp(q)),$
$comp(mkswhile(e)(p)) \equiv mktwhile(e)(comp(p))$

(G1.2.2) (MT is a homomorphism)

$iswfse(e) \equiv TT, iswft(p) \equiv TT, iswft(q) \equiv TT \vdash$
$MT(mktcmpnd(p, q)) \equiv TCMPND(MT(p), MT(q)),$
$MT(mktcond(e)(p, q)) \equiv TCOND(e)(MT(p), MT(q)),$
$MT(mktwhile(e)(p)) \equiv TWHILE(e)(MT(p))$

(G1.2.3) ($SIMUL$ is a homomorphism)

$iswfse(e) \equiv TT, iswfs(p) \equiv TT, iswfs(q) \equiv TT \vdash$
$SIMUL(TCMPND(MT(comp(p)), MT(comp(q)))) \equiv$
$SCMPND(SIMUL(MT(comp(p))), SIMUL(MT(comp(q)))).$
$SIMUL(TCOND(e)(MT(comp(p)), MT(comp(q)))) \equiv$
$SCOND(e)(SIMUL(MT(comp(p))), SIMUL(MT(comp(q)))),$
$SIMUL(TWHILE(e)(MT(comp(p)))) \equiv$
$SWHILE(e)(SIMUL(MT(comp(p))))$

(G1.3.1) (well-formed expressions compile correctly)

$iswfse(e) \equiv TT \vdash$
$MT(compe(e)) \equiv \lambda sp.(svof(sp) \mid (MSE(e) \& pdof(sp)))$

(G1.2.1.1) (assignment statements compile into well-formed target programs)

$$isname(n)\equiv TT,\ iswfse(e)\equiv TT \vdash$$
$$iswft(comp(mkassn(n, e)))\equiv TT$$

(G1.2.1.2) (the operations of **T** preserve well-formedness)

$$iswfse(e)\equiv TT,\ iswft(p)\equiv TT,\ iswft(q)\equiv TT \vdash$$
$$iswft(mktcmpnd(p, q))\equiv TT,$$
$$iswft(mktcond(e)(p, q))\equiv TT,$$
$$iswft(mktwhile(e)(p))\equiv TT$$

(G1.2.2.1) (Context-free lemma for MT)

$$iswft(q)\equiv TT,\ isnat(n)\equiv TT,\ q'\equiv shift(n, q),$$
$$iswft(p @ q' @ r)\equiv TT \vdash$$
$$MT1(p @ q' @ r, q' @ r)\equiv MT(q)\otimes MT1(p @ q' @ r, r)$$

(G1.2.3.1) (For $g\in\hat{\mathbf{T}}'$, $svof(g(sv|pd))$ is independent of pd)

$$iswfs(p)\equiv TT \vdash$$
$$MT(comp(p))\otimes svof\equiv svof\otimes SIMUL(MT(comp(p)))$$

In Appendix 1 we give, in a form acceptable to the proof-checker, those axioms and definitions required for the proof which do not appear above. The only omission is the axioms for natural numbers. In Appendix 3 we give in full the machine printout of the proof of (G1.3.1), the McCarthy-Painter lemma, together with some notes as an aid to understanding it. This theorem has a somewhat independent status, as it states the correctness of that part of our compiler, *compe*, which compiles expressions. Our machine-checked proof therefore parallels the informal proof of essentially the same theorem given by McCarthy and Painter (1967). In Appendix 2 we give the sequence of commands typed by the user in generating the proof of the McCarthy-Painter lemma. We do not explain these commands; we give them merely to indicate that although the proof generated is quite long, the user does not have very much to type.

DISCUSSION OF THE PROOF

In this section we discuss the machine proof, and what we have learnt from carrying it out.

As is apparent from the details we have presented, the proof is lengthy but not profound. We have in fact not checked the whole proof on the machine – (G1.2.2), (G1.2.2.1) and parts of (G1.2.3) and (G1.2.1.2) remain to be done – so at present we cannot claim to have completely proved the correctness of a compiler on the machine. However, the aims were rather (i) to demonstrate that the proof is feasible, (ii) to explore the use of algebraic methods to give structure to the proof, and (iii) to obtain a case study which, in conjunction with those in our previous work (Milner 1972b, Weyhrauch and Milner 1972), give us a feeling for how to enhance our implementation to diminish the human contribution to a proof. We have no significant doubt that the remainder of the proof can be done on the machine.

We have already discussed the value of algebraic methods, at least for this example of a simple compiler. It remains to be seen whether more complex compilers and semantics will fall naturally into the algebraic framework, or whether they may be coerced into the framework – and if so whether the advantages will justify the effort of coercion. But what is certain is that for machine-checked compiler proofs some way of structuring the proof is desirable.

Concerning feasibility; one measure of this is the number of proof steps required. The part of the proof that we have executed took about 600 steps, and we estimate that this is more than half of the total, although (G1.2.2) is not a trivial task. This measure does not take into account the considerable human effort in planning the proof, but – at least if the algebraic method can be applied in more complex cases – some part of this effort will be common to many compiler proofs.

This case study and the others referenced above have convinced us that the formal proofs were indeed feasible, but would not have been so without two features of our proof-checker, namely its subgoaling facility and its simplification mechanism. Usually the most creative contribution that the human makes is the decision as to what instance of the induction rule to apply (we do not discuss the induction rule, but many forms of structural induction are instances of it; for example the goal (G1), once the other goals have been proved, merely requires an induction on the structure of well-formed source programs). Once this decision is made, the remainder of the proof, if it requires no further inductions, follows a pattern which is sufficiently pronounced to give us hope for automation.

Acknowledgements

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defence Under Contract SD–183 and in part by the National Aeronautics and Space Administration under Contract NSR 05–020–500.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency, the National Aeronautics and Space Administration, or the U.S. Government.

We would like to thank Malcolm Newey for his work on the implementation of LCF, Lockwood Morris for valuable discussions concerning algebraic methods, and Henri Ajenstat for carrying out most of the proof of (G1.2.1.2) on the machine.

REFERENCES

Burstall, R.M. & Landin, P.J. (1969) Programs and their proofs: an algebraic approach. *Machine Intelligence 4*, pp. 17–43 (eds Meltzer, B. & Michie, D.). Edinburgh: Edinburgh University Press.

Cohn, P.M. (1965) *Universal Algebra*. New York: Harper and Row.

Diffie, W. (1972) Mechanical verification of a compiler correctness proof. Forthcoming *A.I. Memo*, Stanford University.

Dijkstra, E.W. (1968) Goto statement considered harmful. Letter to Editor, *Comm. Ass. Comp. Mach.*, **11**, 147–8.

Kaplan, D.M. (1967) Correctness of a compiler for ALGOL-like programs. *A. I. Memo 48*. Stanford University.

Lloyd, C. (1972) Some concepts of universal algebra and their application to computer science. CSWP–1, Computing Centre, University of Essex.

London, R.L. (1971) Correctness of two compilers for a LISP subset. *A. I. Memo 151.* Stanford University.

London, R.L. (1972) Correctness of a compiler for a LISP subset. *Proc. Conf. on Proving Assertions about Programs.* New Mexico State University.

McCarthy, J. (1963) Towards a mathematical science of computation. Information processing; *Proc. IFIP Congress 62*, pp. 21–8 (ed. Popplewell, C.M.). Amsterdam: North Holland.

McCarthy, J. & Painter, J.A. (1967) Correctness of a compiler for arithmetic expressions. *Proceedings of a Symposium in Applied Mathematics*, **19**, *Mathematical Aspects of Computer Science*, pp. 33–41 (ed. Schwartz, J.T.). Providence, Rhode Island: American Mathematical Society.

Milner, R. (1972a) Logic for computable functions; description of a machine implementation. *A.I. Memo 169*. Stanford University.

Milner, R. (1972b) Implementation and applications of Scott's logic for computable functions. *Proc. Conf. on Proving Assertions about Programs.* New Mexico State University.

Morris, L. (1972) Forthcoming Ph.D. Thesis, Stanford University.

Painter, J.A. (1967) Semantic correctness of a compiler for an ALGOL-like language. *A.I. Memo 44*. Stanford University; also Ph.D. Thesis. Stanford University.

Weyhrauch, R.W. and Milner, R. (1972) Program semantics and correctness in a mechanized logic. *Proc. USA-Japan Computer Conference*, Tokyo.

APPENDIX 1: Some of the axioms

```
AXIOM LOGAX:
 ∧ ≡ [λP Q ,¬(¬(P)∨¬(Q))], ¬ ≡ [λP ,(P→FF,TT)], [λP Q ,P∨Q] ≡ [λP Q ,Q∨P],
 [λP ,P∨TT] ≡ [λP ,TT], [λP ,P∨FF] ≡ [λP ,P];

AXIOM EQUAX:
 ∂ ≡ [λx ,x=x], ∀x y ,(x=y):: x ≡ y,
 ∀x y ,(∂ x)∧(∂ y) ≡ (x=y)∨¬(x=y);

AXIOM LISAX:
 ∀x y, hd(x&y) ≡ x,     ∀x y, tl(x&y) ≡ y,     ∀x y, null(x&y) ≡ FF,
 null NIL ≡ TT,    ∀x, null x :: x ≡ NIL,     null UU ≡ UU,
 hd UU ≡ UU,     tl UU ≡ UU,    ∀x, ¬(null x) :: (hd x)&(tl x) ≡ x;

AXIOM LISFN:
 @ ≡ αf. λl m, null l → m, (hd l)&f(tl l,m);

AXIOM SYNAXS:
 ∀o e1 e2, type (mkse o e1 e2) ≡ _E,          _N=_N ≡ TT,
 ∀o e1 e2, opof (mkse o e1 e2) ≡ o ,          _E=_N ≡ FF,
 ∀o e1 e2, arg1of(mkse o e1 e2) ≡ e1,         _N=_E ≡ FF,
 ∀o e1 e2, arg2of(mkse o e1 e2) ≡ e2,         _E=_E ≡ TT,

 ∀n e, type(mksassn n e) ≡ _A,                _A=_A ≡ TT,
 ∀n e, lhsof(mksassn n e) ≡ n ,               _C=_A ≡ FF,
 ∀n e, rhsof(mksassn n e) ≡ e ,               _W=_A ≡ FF,
                                              _CM=_A ≡ FF,
 ∀e p1 p2, type (mkscond e p1 p2) ≡ _C,       _A =_C  ≡ FF,
 ∀e p1 p2, ifof (mkscond e p1 p2) ≡ e,        _C =_C  ≡ TT,
 ∀e p1 p2, thenof(mkscond e p1 p2) ≡ p1,      _W =_C  ≡ FF,
 ∀e p1 p2, elseof(mkscond e p1 p2) ≡ p2,      _CM=_C  ≡ FF,
                                              _A =_W  ≡ FF,
 ∀e p, type(mkswhile e p) ≡ _W,               _C =_W  ≡ FF,
 ∀e p, testof (mkswhile e p) ≡ e ,            _W =_W  ≡ TT,
 ∀e p, bodyof (mkswhile e p) ≡ p,             _CM=_W  ≡ FF,
                                              _A =_CM ≡ FF,
 ∀p1 p2, type(mkscmpnd p1 p2) ≡ _CM,          _C =_CM ≡ FF,
 ∀p1 p2, firstof(mkscmpnd p1 p2) ≡ p1,        _W =_CM ≡ FF,
 ∀p1 p2,secondof(mkscmpnd p1 p2) ≡ p2,        _CM=_CM ≡ TT,

AXIOM SYNAXT:
 JF    =JF     ≡ TT,      JF    =J     ≡ FF,      JF    =FETCH ≡ FF,
 J     =JF     ≡ FF,      J     =J     ≡ TT,      J     =FETCH ≡ FF,
 FETCH=JF      ≡ FF,      FETCH=J      ≡ FF,      FETCH=FETCH ≡ TT,
 STORE=JF      ≡ FF,      STORE=J      ≡ FF,      STORE=FETCH ≡ FF,
 DO    =JF     ≡ FF,      DO    =J     ≡ FF,      DO    =FETCH ≡ FF,
 LABEL=JF      ≡ FF,      LABEL=J      ≡ FF,      LABEL=FETCH ≡ FF,
 JF    =STORE ≡ FF,       JF    =DO    ≡ FF,      JF    =LABEL ≡ FF,
 J     =STORE ≡ FF,       J     =DO    ≡ FF,      J     =LABEL ≡ FF,
 FETCH=STORE ≡ FF,        FETCH=DO     ≡ FF,      FETCH=LABEL ≡ FF,
 STORE=STORE ≡ TT,        STORE=DO     ≡ FF,      STORE=LABEL ≡ FF,
 DO    =STORE ≡ FF,       DO    =DO    ≡ TT,      DO    =LABEL ≡ FF,
 LABEL=STORE ≡ FF,        LABEL=DO     ≡ FF,      LABEL=LABEL ≡ TT,
```

APPENDIX 2: command sequence for McCarthy-Painter lemma

```
GOAL ∀e sp,Iswfse e::MT(compe e,sp)≡svof(sp)|((MSE(e,svof sp))&pdof(sp)),
     ∀e,Iswfse e::Iswft(compe e)≡TT,
     ∀e,Iswfse e::(count(compe e)=Ø)≡TT;
TRY 1 INDUCT 56;
 TRY 1 SIMPL;
 LABEL INDHYP;
 TRY 2 ABSTR;
  TRY 1 CASES wfsefun(f,e);
  LABEL TT;
   TRY 1 CASES type e=_N;
    TRY 1 SIMPL BY ,FMT1,,FMSE;,FCOMPE,,FISWFT1,,FCOUNT;
    TRY 2;SS-,TT;SIMPL,TT;QED;
    TRY 3 CASES type e=_E;
     TRY 1 SUBST ,FCOMPE;
      SS-,TT;SIMPL,TT;USE BOTH3 -;SS+,TT;
      INCL-,1;SS+-;INCL--,2;SS+-;INCL---,3;SS+-;
       TRY 1 CONJ;
        TRY 1 SIMPL;
         TRY 1 USE COUNT1;
          TRY 1;
          APPL ,INDHYP+2,arg1of e;
          LABEL CARG1;
          SIMPL-;QED;
          TRY 2 USE COUNT1;
           TRY 1;
           APPL ,INDHYP+2,arg2of e;
           LABEL CARG2;
           SIMPL-;QED;
           LABEL CDO;
           TRY 2 SIMPL BY ,FCOUNT;
          TRY 2 SIMPL BY ,FISWFT1,--;
          SIMPL Iswft(compe(arg1of e)) BY ,FISWFT1,,CARG1;
          SIMPL Iswft(compe(arg2of e)) BY ,FISWFT1,,CARG2;
          USE THM3 -,,CDO;SS+-;
          USE COUNT1 ,CARG2,,CDO;
          USE THM3 ----,-;SS+-;
      APPL ,INDHYP,arg1of e;SIMPL-;SS+-;
      APPL ,INDHYP,arg2of e;SIMPL-;SS+-;
      TRY 3 SUBST ,FMSE OCC 1;
      SS-108;
      TRY 1 SIMPL;
      SS+108;
      TRY 1 SIMPL BY ,FMT1;
  TRY 2;SS-,TT;SIMPL,TT;QED;
  TRY 3;SS-,TT;SIMPL,TT;QED;
  TBY 2 SIMPL;
  TRY 3 SIMPL;
```

APPENDIX 3: proof of the McCarthy-Painter lemma

```
 ---T------------------
 |TRY #1   Ve sp , |swfse(e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve . |swfse(e) :: |swf-
t(compe(e)) ≡ TT , Ve , |swfse(e) :: (count(compe(e))=Ø) ≡ TT     INDUCT 56 ,
 | -T------------------
 | |TRY #1#1   Ve sp , UU(e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve . UU(e) :: |swft(co-
mpe(e)) ≡ TT , Ve , UU(e) :: (count(compe(e))=Ø) ≡ TT      SIMPL .
 | |151   Ve sp . UU(e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve . UU(e) :: |swft(compe(e-
)) ≡ TT , Ve . UU(e) :: (count(compe(e))=Ø) ≡ TT   --- SIMPL.
 | -T------------------
 | -T------------------
 | |TRY #1#2   Ve sp , wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve . wfsefun(f,-
e) :: |swft(compe(e)) ≡ TT , Ve . wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT ! ASSUME   Ve sp , f(e) :: MT(compe(e-
),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve , f(e) :: |swft(compe(e)) ≡ TT , Ve , f(e) :: (count(compe(e)-
)=Ø) ≡ TT     ABSTR.
 | |152   Ve sp . f(e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp)))  (152) --- ASSUME.
 | |153   Ve , f(e) :: |swft(compe(e)) ≡ TT  (153) --- ASSUME.
 | |154   Ve , f(e) :: (count(compe(e))=Ø) ≡ TT  (154) --- ASSUME.
 | | -------------------
 | | |TRY #1#2#1   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |sw-
ft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT     CASES wfsefun(f,e).
 | | | -------------------
 | | | |TRY #1#2#1#1   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ::-
|swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT ! SASSUME   wfsefun(f,e) ≡ TT     CASES (type(e-
)=_N).
 | | | |155   wfsefun(f,e) ≡ TT  (155) --- SASSUME.
 | | | | -------------------
 | | | | |TRY #1#2#1#1#1   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e-
) :: |swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT ! SASSUME   (type(e)=_N) ≡ TT     SIMPL BY -
140 138 147 146 142.
 | | | | |156   (type(e)=_N) ≡ TT  (156) --- SASSUME.
 | | | | |157   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swft(-
compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT  (155 156) --- SIMPL BY 5 9 10 11 15 63 71 77 83 101 1-
08 111 112 116 127 130 138 140 142 146 147 150 155 156.
 | | | | -------------------
 | | | | -------------------
 | | | | |TRY #1#2#1#1#2   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e-
) :: |swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT ! SASSUME   (type(e)=_N) ≡ UU    .
 | | | | |158   (type(e)=_N) ≡ UU  (158) --- SASSUME.
 | | | | |159   UU ≡ TT  (155 158) --- SIMPL 155 BY 57 158.
 | | | | -------------------
```

F

```
 | | | | --------------------
 | | | | |TRY #1#2#1#1#3   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e⁻
) :: |swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT | SASSUME   (type(e)=_N) ≡ FF     CASES (ty⁻
pe(e)=_E),
 | | | | |16Ø   (type(e)=_N) ≡ FF  (16Ø) --- SASSUME.
 | | | | | --------------------
 | | | | | |TRY #1#2#1#1#3#1   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun⁻
(f,e) :: |swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT | SASSUME   (type(e)=_E) ≡ TT     SUBST⁻
 147 .
 | | | | | |161   (type(e)=_E) ≡ TT  (161) --- SASSUME,
 | | | | | |162   ((|sop(opof(e))∧f(arg1of(e)))∧f(arg2of(e))) ≡ TT  (155 16Ø 161) --- SIMPL 155 BY 57 16Ø 161,
 | | | | | |163   |sop(opof(e)) ≡ TT , f(arg1of(e)) ≡ TT , f(arg2of(e)) ≡ TT  (155 16Ø 161) --- USE BOTH3 162,
 | | | | | |164   |sop(opof(e)) ≡ TT  (155 16Ø 161) --- INCL 163.
 | | | | | |165   f(arg1of(e)) ≡ TT  (155 16Ø 161) --- INCL 163,
 | | | | | |166   f(arg2of(e)) ≡ TT  (155 16Ø 161) --- INCL 163,
 | | | | | | --------------------
 | | | | | | |TRY #1#2#1#1#3#1#1   wfsefun(f,e) :: MT(compefun(compe,e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp⁻
))) , wfsefun(f,e) :: |swft(compefun(compe,e)) ≡ TT , wfsefun(f,e) :: (count(compefun(compe,e))=Ø) ≡ TT     CONJ⁻
.
 | | | | | | | --------------------
 | | | | | | | |TRY #1#2#1#1#3#1#1#1   wfsefun(f,e) :: (count(compefun(compe,e))=Ø) ≡ TT     SIMPL .
 | | | | | | | | --------------------
 | | | | | | | | |TRY #1#2#1#1#3#1#1#1#1   (count((compe(arg1of(e))@(compe(arg2of(e))@((DO&opof(e))&NIL))))=Ø) ≡⁻
TT      USE COUNT1,
 | | | | | | | | | --------------------
 | | | | | | | | | |TRY #1#2#1#1#3#1#1#1#1#1   (count(compe(arg1of(e)))=Ø) ≡ TT    ,
 | | | | | | | | | |167   [λe .(f(e)→(count(compe(e))=Ø),UU)](arg1of(e)) ≡ [λe .(f(e)→TT,UU)](arg1of(e))  (154) ⁻
--- APPL 154 arg1of(e),
 | | | | | | | | | |168   (count(compe(arg1of(e)))=Ø) ≡ TT  (154 155 16Ø 161) --- SIMPL 167 BY 165.
 | | | | | | | | | --------------------
 | | | | | | | | | --------------------
 | | | | | | | | | |TRY #1#2#1#1#3#1#1#1#1#2   (count((compe(arg2of(e))@((DO&opof(e))&NIL)))=Ø) ≡ TT     USE COU⁻
NT1,
 | | | | | | | | | | --------------------
 | | | | | | | | | | |TRY #1#2#1#1#3#1#1#1#1#2#1   (count(compe(arg2of(e)))=Ø) ≡ TT    .
 | | | | | | | | | | |169   [λe .(f(e)→(count(compe(e))=Ø),UU)](arg2of(e)) ≡ [λe .(f(e)→TT,UU)](arg2of(e))  (154⁻
) --- APPL 154 arg2of(e),
 | | | | | | | | | | |17Ø   (count(compe(arg2of(e)))=Ø) ≡ TT  (154 155 16Ø 161) --- SIMPL 169 BY 166.
 | | | | | | | | | | --------------------
 | | | | | | | | | | --------------------
 | | | | | | | | | | |TRY #1#2#1#1#3#1#1#1#1#2#2   (count(((DO&opof(e))&NIL))=Ø) ≡ TT     SIMPL BY 142,
 | | | | | | | | | | |171   (count(((DO&opof(e))&NIL))=Ø) ≡ TT   --- SIMPL BY 5 9 1Ø 11 15 73 79 1Ø3 142 15Ø,
 | | | | | | | | | | --------------------
 | | | | | | | | | |172   (count((compe(arg2of(e))@((DO&opof(e))&NIL)))=Ø) ≡ TT  (154 155 16Ø 161) --- USE COUNT⁻
1 17Ø 171,
 | | | | | | | | | --------------------
 | | | | | | | | |173   (count((compe(arg1of(e))@(compe(arg2of(e))@((DO&opof(e))&NIL))))=Ø) ≡ TT  (154 155 16Ø 1⁻
61) --- USE COUNT1 168 172,
 | | | | | | | | --------------------
 | | | | | | | |174   wfsefun(f,e) :: (count(compefun(compe,e))=Ø) ≡ TT  (154 155 16Ø 161) --- SIMPL 173 BY 13Ø ⁻
155 26Ø 161,
 | | | | | | | --------------------
 | | | | | | | --------------------
 | | | | | | | |TRY #1#2#1#1#3#1#1#2   wfsefun(f,e) :: |swft(compefun(compe,e)) ≡ TT     SIMPL BY 146 173,
 | | | | | | | |175   wfsefun(f,e) :: |swft(compefun(compe,e)) ≡ TT  (154 155 16Ø 161) --- SIMPL BY 127 13Ø 146 -
155 16Ø 161 173,
 | | | | | | | --------------------
 | | | | | | |176   |swft(compe(arg1of(e))) ≡ TT  (154 155 16Ø 161) --- SIMPL |swft(compe(arg1of(e))) BY 127 146-
 168,
 | | | | | | |177   |swft(compe(arg2of(e))) ≡ TT  (154 155 16Ø 161) --- SIMPL |swft(compe(arg2of(e))) BY 127 146-
 17Ø,
 | | | | | | |178   MT((compe(arg2of(e))@((DO&opof(e))&NIL))) ≡ (MT(compe(arg2of(e)))•MT(((DO&opof(e))&NIL)))  (-
154 155 16Ø 161) --- USE THM3 177 171,
 | | | | | | |179   (count((compe(arg2of(e))@((DO&opof(e))&NIL)))=Ø) ≡ TT  (154 155 16Ø 161) --- USE COUNT1 17Ø -
171,
 | | | | | | |18Ø   MT((compe(arg1of(e))@(compe(arg2of(e))@((DO&opof(e))&NIL)))) ≡ (MT(compe(arg1of(e)))•MT((com-
pe(arg2of(e))@((DO&opof(e))&NIL))))  (154 155 16Ø 161) --- USE THM3 176 179,
 | | | | | | |181   [λe sp .(f(e)→MT(compe(e),sp),UU)](arg1of(e)) ≡ [λe sp .(f(e)→(svof(sp)|(MSE(e,svof(sp))&pdo-
f(sp))),UU)](arg1of(e))  (152) --- APPL 152 arg1of(e),
 | | | | | | |182   ∀sp . MT1(compe(arg1of(e)),compe(arg1of(e)),sp) ≡ (svof(sp)|(MSE(arg1of(e),svof(sp))&pdof(sp-
)))  (152 155 16Ø 161) --- SIMPL 181 BY 1Ø8 165,
 | | | | | | |183   [λe sp .(f(e)→MT(compe(e),sp),UU)](arg2of(e)) ≡ [λe sp .(f(e)→(svof(sp)|(MSE(e,svof(sp))&pdo-
f(sp))),UU)](arg2of(e))  (152) --- APPL 152 arg2of(e),
 | | | | | | |184   ∀sp . MT1(compe(arg2of(e)),compe(arg2of(e)),sp) ≡ (svof(sp)|(MSE(arg2of(e),svof(sp))&pdof(sp-
)))  (152 155 16Ø 161) --- SIMPL 183 BY 1Ø8 166,
 | | | | | | | --------------------
 | | | | | | | |TRY #1#2#1#1#3#1#1#3   wfsefun(f,e) :: MT(compefun(compe,e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdo-
f(sp)))     SUBST 138 OCC 1,
 | | | | | | | | --------------------
 | | | | | | | | |TRY #1#2#1#1#3#1#1#3#1   wfsefun(f,e) :: MT(compefun(compe,e),sp) ≡ (svof(sp)|([λe s .((type(e-
)=_N)→s(e),((type(e)=_E)→MOP(opof(e),MSE(arg1of(e),s),MSE(arg2of(e),s)),UU))](e,svof(sp))&pdof(sp)))     SIMPL ,
 | | | | | | | | | --------------------
 | | | | | | | | | |TRY #1#2#1#1#3#1#1#3#1#1   MT(((DO&opof(e))&NIL),MT(compe(arg2of(e)),MT(compe(arg1of(e)),sp)-
)) ≡ (svof(sp)|(MOP(opof(e),MSE(arg1of(e),svof(sp)),MSE(arg2of(e),svof(sp)))&pdof(sp)))     SIMPL BY 14Ø,
 | | | | | | | | | |185   MT(((DO&opof(e))&NIL),MT(compe(arg2of(e)),MT(compe(arg1of(e)),sp))) ≡ (svof(sp)|(MOP(o-
pof(e),MSE(arg1of(e),svof(sp)),MSE(arg2of(e),svof(sp)))&pdof(sp)))  (152 155 16Ø 161) --- SIMPL BY 9 1Ø 11 15 63-
 73 79 85 91 97 1Ø8 111 112 116 14Ø 182 184,
 | | | | | | | | | --------------------
 | | | | | | | | |186   wfsefun(f,e) :: MT(compefun(compe,e),sp) ≡ (svof(sp)|([λe s .((type(e)=_N)→s(e),((type(e-
)=_E)→MOP(opof(e),MSE(arg1of(e),s),MSE(arg2of(e),s)),UU))](e,svof(sp))&pdof(sp)))  (152 154 155 16Ø 161) --- SIM-
PL 185 BY 63 13Ø 155 16Ø 161 178 18Ø,
 | | | | | | | | --------------------
 | | | | | | | |187   wfsefun(f,e) :: MT(compefun(compe,e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp)))  (152 154-
 155 16Ø 161) --- SUBST 186 USING 138 OCC 1,
 | | | | | | | --------------------
 | | | | | | |188   wfsefun(f,e) :: MT(compefun(compe,e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f-
,e) :: |swft(compefun(compe,e)) ≡ TT , wfsefun(f,e) :: (count(compefun(compe,e))=Ø) ≡ TT  (152 154 155 16Ø 161) -
--- CONJ 174 175 187,
 | | | | | | --------------------
 | | | | | |189   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swf-
t(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT  (152 154 155 16Ø 161) --- SUBST 188 USING 147 ,
 | | | | | --------------------
 | | | | | --------------------
 | | | | | |TRY #1#2#1#1#3#2   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun-
(f,e) :: |swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT ! SASSUME   (type(e)=_E) ≡ UU    ;
 | | | | | |19Ø   (type(e)=_E) ≡ UU  (19Ø) --- SASSUME,
 | | | | | |191   UU ≡ TT  (155 16Ø 19Ø) --- SIMPL 155 BY 57 16Ø 19Ø,
 | | | | | --------------------
 | | | | | --------------------
 | | | | | |TRY #1#2#1#1#3#3   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun~
(f,e) :: |swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT | SASSUME   (type(e)=_E) ≡ FF    ,
 | | | | | |192   (type(e)=_E) ≡ FF  (192) --- SASSUME,
 | | | | | |193   UU ≡ TT  (155 160 192) --- SIMPL 155 BY 57 160 192,
 | | | | | --------------------
 | | | | |194   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swft(~
compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT  (152 154 155 160) --- CASES (type(e)=_E) 189 191 193,
 | | | | --------------------
 | | | |195   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swft(co~
mpe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT  (152 154 155) --- CASES (type(e)=_N) 157 159 194,
 | | | --------------------
 | | | --------------------
 | | | |TRY #1#2#1#2   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ::~
 |swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT | SASSUME   wfsefun(f,e) ≡ UU     SIMPL ,
 | | | |196   wfsefun(f,e) ≡ UU  (196) --- SASSUME,
 | | | |197   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swft(co~
mpe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT  (196) --- SIMPL BY 196,
 | | | --------------------
 | | | --------------------
 | | | |TRY #1#2#1#3   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) ::~
 |swft(compe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT | SASSUME   wfsefun(f,e) ≡ FF     SIMPL ,
 | | | |198   wfsefun(f,e) ≡ FF  (198) --- SASSUME,
 | | | |199   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swft(co~
mpe(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT  (198) --- SIMPL BY 198,
 | | | --------------------
 | | |200   wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , wfsefun(f,e) :: |swft(comp~
e(e)) ≡ TT , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT  (152 154) --- CASES wfsefun(f,e) 195 197 199,
 | | --------------------
 | |201   Ve sp , wfsefun(f,e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve , wfsefun(f,e) ::~
 |swft(compe(e)) ≡ TT , Ve , wfsefun(f,e) :: (count(compe(e))=Ø) ≡ TT  (152 154) --- ABSTR 200,
 | --------------------
 |202   Ve sp , |swfse(e) :: MT(compe(e),sp) ≡ (svof(sp)|(MSE(e,svof(sp))&pdof(sp))) , Ve , |swfse(e) :: |swft(c~
ompe(e)) ≡ TT , Ve , |swfse(e) :: (count(compe(e))=Ø) ≡ TT   --- INDUCT 151 201,
 --------------------
```

The proof presented here actually proves more than just the McCarthy-Painter lemma. It also shows that the *count* of a compiled expression is Ø and thus *compe*(*e*) of a well-formed source expression *e* is a well-formed target program. These three facts are proved simultaneously by a structural induction on well-formed source programs. The main goal is stated at *TRY* #1, and the base case of the induction at *TRY* #1#1. In the machine print-out of a proof the successive subgoals are nested in their natural way by using boxes. The main step of the induction is stated at *TRY* #1#2. The '::' in sentences of the form $A::B\equiv C$ can be interpreted as meaning if $A\equiv TT$ then $B\equiv C$. Actually it is an abbreviaton for $(A\rightarrow B, UU)\equiv(A\rightarrow C, UU)$. The induction hypotheses are stated at steps 152, 153, 154. Assuming *wfsefun*(*f*, *e*) is true, the proof then proceeds in a straightforward way by cases on *type*(*e*); first if *e* is a name and then if *e* is a compound expression. The numbered steps of the printout can be read as a formal proof of the lemma. Each step is followed by its list of dependencies in parentheses, for example, line 174 depends on steps 154, 155, 160, 161.