CORRECTNESS OF A COMPILER FOR ARITHMETIC EXPRESSIONS*

JOHN McCARTHY and JAMES PAINTER

1967

1 Introduction

This paper contains a proof of the correctness of a simple compiling algorithm for compiling arithmetic expressions into machine language.

The definition of correctness, the formalism used to express the description of source language, object language and compiler, and the methods of proof are all intended to serve as prototypes for the more complicated task of proving the correctness of usable compilers. The ultimate goal, as outlined in references [1], [2], [3] and [4] is to make it possible to use a computer to check proofs that compilers are correct.

The concepts of abstract syntax, state vector, the use of an interpreter for defining the semantics of a programming language, and the definition of correctness of a compiler are all the same as in [3]. The present paper, however, is the first in which the correctness of a compiler is proved.

The expressions dealt with in this paper are formed from constants and variables. The only operation allowed is a binary + although no change in method would be required to include any other binary operations. An example of an expression that can be compiled is

$$(x + 3) + (x + (y + 2))$$

*This is a reprint with minor changes of ”Correctness of a Compiler for Arithmetic Expressions” by John McCarthy and James Painter which was published in MATHEMATICAL ASPECTS OF COMPUTER SCIENCE 1, which was Volume 19 of *Proceedings of Symposia in Applied Mathematics* and published by the American Mathematical Society in 1967 although, because we use abstract syntax, no commitment to a particular notation is made.

The computer language into which these expressions are compiled is a single address computer with an accumulator, called ac, and four instructions: li (load immediate), load, sto (store) and add. Note that there are no jump instructions. Needless to say, this is a severe restriction on the generality of our results which we shall overcome in future work.

The compiler produces code that computes the value of the expression being compiled and leaves this value in the accumulator. The above expression is compiled into code which in assembly language might look as follows:

```
load    x
sto     t
li      3
add     t
sto     t
load    x
sto     t + 1
load    y
sto     t + 2
li      2
add     t + 2
add     t + 1
add     t
```

Again because we are using abstract syntax there is no commitment to a precise form for the object code.

2 The source language

The abstract analytic syntax of the source expressions is given by the table:

predicate	associated functions
isconst(e)	
isvar(e)	
issum(e)	s1(e) s2(e)

which asserts that the expressions comprise constants, variables and binary sums, that the predicates *isconst*, *isvar*, and *issum* enable one to classify each expression and that each sum e has summands $s1(e)$ and $s2(e)$.

The semantics is given by the formula

$$\text{(2.1)} \qquad \text{value}(e,\xi) = \textbf{ if } \text{isconst}(e) \textbf{ then } \text{val}(e) \textbf{ else if } \text{isvar}(e) \textbf{ then } c(e,\xi) \\ \textbf{else if } \text{issum}(e) \textbf{ then } \text{value}(s1(e),\xi) + \text{value}(s2(e),\xi)$$

where val(e) gives the numerical value of an expression e representing a constant, $c(e,\xi)$ gives the value of the variable e in the state vector ξ and + is some binary operation. (It is natural to regard + as an operation that resembles addition of real numbers, but our results do not depend on this).

For our present purposes we do not have to give a synthetic syntax for the source language expressions since both the interpreter and the compiler use only the analytic syntax. However, we shall need the following induction principle for expressions:

Suppose Φ is a predicate applicable to expressions, and suppose that for all expressions e we have

$$\begin{array}{ll} \text{isconst}(e) \supset \Phi(e) & \text{and} \\ \text{isvar}(e) \supset \Phi(e) & \text{and} \\ \text{issum}(e) \supset \Phi(s1(e)) \wedge \Phi(s2(e)) \supset \Phi(e). & \end{array}$$

Then we may conclude that $\Phi(e)$ is true for all expressions e.

3 The object language.

We must give both the analytic and synthetic syntaxes for the object language because the interpreter defining its semantics uses the analytic syntax and the compiler uses the synthetic syntax. We may write the analytic and synthetic syntaxes for instructions in the following table.

operation	predicate	analytic operation	synthetic operation
li α	isli(s)	arg(s)	mkli(α)
load x	isload(s)	adr(s)	mkload(x)
sto x	issto(s)	adr(s)	mksto(x)
add x	isadd(s)	adr(s)	mkadd(x)

A program is a list of instructions and null(p) asserts that p is the null list. If the program p is not null then first(p) gives the first instruction and rest(p)

gives the list of remaining instructions. We shall use the operation $p1 * p2$ to denote the program obtained by appending $p2$ onto the end of $p1$. Since we have only one level of list we can identify a single instruction with a program that has just one instruction.

The synthetic and analytic syntaxes of instructions are related by the following.

$$\begin{aligned}
&\mathit{isli}(\mathit{mkli}(\alpha))\\
&\alpha = \mathit{arg}(\mathit{mkli}(\alpha))\\
&\mathit{isli}(s) \supset s = \mathit{mkli}(\mathit{arg}(s))\\
&\mathit{null}(\mathit{rest}(\mathit{mkli}(\alpha)))\\
&\mathit{isli}(s) \supset \mathit{first}(s) = s
\end{aligned}\tag{3.1}$$

$$\begin{aligned}
&\mathit{isload}(\mathit{mkload}(x))\\
&x = \mathit{adr}(\mathit{mkload}(x))\\
&\mathit{isload}(x) \supset x = \mathit{mkload}(\mathit{adr}(x))\\
&\mathit{null}(\mathit{rest}(\mathit{mkload}(x)))\\
&\mathit{isload}(s) \supset \mathit{first}(s) = s
\end{aligned}\tag{3.2}$$

$$\begin{aligned}
&\mathit{issto}(\mathit{mksto}(x))\\
&x = \mathit{adr}(\mathit{mksto}(x))\\
&\mathit{issto}(x) \supset x = \mathit{mksto}(\mathit{adr}(x))\\
&\mathit{null}(\mathit{rest}(\mathit{mksto}(x)))\\
&\mathit{issto}(s) \supset \mathit{first}(s) = s
\end{aligned}\tag{3.3}$$

$$\begin{aligned}
&\mathit{isadd}(\mathit{mkadd}(x))\\
&x = \mathit{adr}(\mathit{mkadd}(x))\\
&\mathit{isadd}(x) \supset x = \mathit{mkadd}(\mathit{adr}(x))\\
&\mathit{null}(\mathit{rest}(\mathit{mkadd}(x)))\\
&\mathit{isadd}(x) \supset \mathit{first}(s) = s
\end{aligned}\tag{3.4}$$

$$\neg\, \mathit{null}(p) \supset p = \mathit{first}(p) * \mathit{rest}(p),\tag{3.5}$$

$$\neg \mathit{null}(p1) \wedge \mathit{null}(\mathit{rest}(p1)) \supset p1 = \mathit{first}(p1 * p2)\tag{3.6}$$

$$\mathit{null}(p1 * p2) \equiv \mathit{null}(p1) \wedge \mathit{null}(p2).\tag{3.7}$$

The $*$ operation is associative. (The somewhat awkward form of these relations comes from having a general concatenation operation rather than just an operation that prefixes a single instruction onto a program.)

A state vector for a machine gives, for each register in the machine, its contents. We include the accumulator denoted by ac as a register. There are two functions of state vectors as introduced in [3], namely

1. $c(x, \eta)$ denotes the value of the contents of register x in machine state η.

2. $a(x, \alpha, \eta)$ denotes the state vector that is obtained from the state vector η by changing the contents of register x to α leaving the other registers unaffected.

These functions satisfy the following relations:

$$c(x, a(y, \alpha, \eta)) = \textbf{ if } \ x = y \ \textbf{ then } \ \alpha \textbf{ else } c(x, \eta), \tag{3.8}$$

$$a(x, \alpha, a(y, \beta, \eta)) = \textbf{ if } \ x = y \ \textbf{ then } \ a(x, \alpha, \eta) \textbf{ else } a(y, \beta, a(x, \alpha, \eta)), \tag{3.9}$$

$$a(x, c(x, \eta), \eta) = \eta. \tag{3.10}$$

Now we can define the semantics of the object language by

$$\begin{aligned} step(s, \eta) = &\textbf{ if } isli(s) \textbf{then } a(ac, arg(s), \eta) \\ &\textbf{else if} isload(s) \textbf{ then } a(ac, c(adr(s), \eta), \eta) \\ &\textbf{else if} issto(s) \textbf{ then} a(adr(s), c(ac, \eta), \eta) \\ &\textbf{elseif } isadd(s) \textbf{ then } a(ac, c(adr(s), \eta) + c(ac, \eta), \eta) \end{aligned} \tag{3.11}$$

which gives the state vector that results from executing an instruction and
(3.12)
$$\text{outcome}(p, \eta) \quad = \text{ if null}(p) \ \text{ then } \eta \text{ else outcome}(\text{rest}(p), \text{step}(\text{first}(p), \eta))$$

which gives the state vector that results from executing the program p with state vector η.

The following lemma is easily proved.

$$\text{outcome}(p1 * p2, \eta) = \text{outcome}(p2, \text{outcome}(p1, \eta)) \tag{3.13}$$

4 The compiler

We shall assume that there is a map giving for each variable in the expression a location in the main memory of the machine loc(ν, map) gives this location and we shall assume

$$c(\text{loc}(\nu, \text{map}), \eta) = c(\nu, \xi) \tag{4.1}$$

as a relation between the state vector η before the compiled program starts to act and the state vector ξ of the source program.

Now we can write the compiler. It is

```
        compile($e, t$) = if  isconst($e$) then  mkli(val(e))
 (4.2)      else if isvar(e) then  mkload(loc(e,map))
             else  if issum($e$) then  compile($s1(e), t$) $*$ mksto($t$) $*$ compile($s2, t+1$) $*$ mkadd($t$)
```

Here t is the number of a register such that all variables are stored in registers numbered less than t, so that registers t and above are available for temporary storage.

Before we can state our definition of correctness of the compiler, we need a notion of partial equality for state vectors

$$\zeta_1 =_A \zeta_2,$$

where ζ_1 and ζ_2 are state vectors and A is a set of variables means that corresponding components of ζ_1 and ζ_2 are equal except possibly for values, of variables in A. Symbolically, $x \notin A \supset c(x, \zeta_1) = c(x, \zeta_2)$. Partial equality satisfies the following relations:

$$\zeta_1 = \zeta_2 \text{ is equivalent to } \zeta_1 =_{\{\}} \zeta_2, \text{ where } \{\} \text{ denotes the empty set },\tag{4.3}$$

$$\text{if } A \subset B \text{ and } \zeta_1 =_A \zeta_2 \text{ then } \zeta_1 =_B \zeta_2.\tag{4.4}$$

$$\text{if } \zeta_1 =_A \zeta_2 \text{ then } a(x, \alpha, \zeta_1) =_{A-\{x\}} a(x, \alpha, \zeta_2).\tag{4.5}$$

$$\text{if } x \in A \text{ then } a(x, \alpha, \zeta) =_A \zeta,\tag{4.6}$$

$$\text{if } \zeta_1 =_A \zeta_2 \text{ and } \zeta_2 =_B \zeta_3 \text{ then } \zeta_1 =_{A \cup B} \zeta_3.\tag{4.7}$$

In our case we need a specialization of this notation and will use

$$\zeta_1 =_t \zeta_2 \text{ to denote } \zeta_1 =_{\{x|x\geq t\}} \zeta_2$$

and

$$\zeta_1 =_{ac} \zeta_2 \text{ to denote } \zeta_1 =_{\{ac\}} \zeta_2$$

and

$$\zeta_1 =_{t,ac} \zeta_2 \text{ to denote } \zeta_1 =_{\{x|x=ac\vee x\geq t\}} \zeta_2.$$

The correctness of the compiler is stated in

THEOREM 1. *If η and ξ are machine and source language state vectors respectively such that*

$$(4.8)\quad c(loc(v,\eta) = c(v,\xi), \quad \text{then} \quad outcome(compile(e,t),\eta) =_t a(\text{ac,value}(e,\xi),\eta).$$

It states that the result of running the compiled program is to put the value of the expression compiled into the accumulator. No registers except the accumulator and those with addresses $\geq t$ are affected.

5 Proof of Theorem 1.

The proof is accomplished by an induction on the expression e being compiled. We prove it first for constants, then for variables, and then for sums on the induction hypothesis that it is true for the summands. Thus there are three cases.

I. isconst(e). We have

$$\begin{array}{lll}
 & & \underline{\text{Justification}} \\
outcome(compile(e,t),\eta) & = outcome(mkli(val(e)),\eta) & 4.2 \\
 & = step(mkli(val(e)),\eta) & 3.12, 3.1 \\
 & = a(ac, arg(mkli(val(e))),\eta) & 3.1, 3.11 \\
 & = a(ac, val(e),\eta) & 3.1 \\
 & = a(ac, value(e,\xi),\eta) & 2.1 \\
 & =_t a(ac, value(e,\xi),\eta). & 4.3, 4.4
\end{array}$$

II. isvar(e). We have

$$\begin{array}{lll} \mathit{outcome}(\mathit{compile}(e,t),\eta) & & \\ & = \mathit{outcome}(\mathit{mkload}(\mathit{loc}(e,\mathit{map})),\eta) & 4.2 \\ & = a(ac, c(\mathit{adr}(\mathit{mkload}(\mathit{loc}(e))),\eta),\eta) & 3.12, 3.2, 3.11 \\ & = a(ac, c(\mathit{loc}(e,\mathit{map}),\eta,\eta) & 3.2 \\ & = a(ac, c(e,\xi),\eta) & 4.1 \\ & = a(ac, \mathit{value}(e,\xi),\eta) & 2.1 \\ & =_t a(ac, \mathit{value}(e,\xi),\eta). & 4.3, 4.4 \end{array}$$

III. issum(e). In this case, we first write

$$\begin{array}{ll} \mathit{outcome}(\mathit{compile}(e,t),\eta) & \\ \quad = \mathit{outcome}(\mathit{compile}(s1(e),t) * \mathit{mksto}(t) & \\ \qquad *\text{compile}(s2(e),t+1) * \text{mkadd}(t),\eta) & \text{by } 4.2 \end{array}$$

$$= \mathit{outcome}(\mathit{mkadd}(t), \mathit{outcome}(\mathit{compile}(s2(e),t+1),$$

$$\text{outcome}(\text{mksto}(t),\text{outcome}(\text{compile}(s1(e),t),\eta)))) \qquad \text{by } 3.13$$

using the relation between concatenating programs and composing the functions they represent. Now we introduce some notation. Let

$$\begin{aligned} \nu &= \text{value}(e,\xi), \\ \nu_1 &= \text{value}(s1(e),\xi), \\ \nu_2 &= \text{value}(s2(e),\xi), \end{aligned}$$

so that $\nu = \nu_1 + \nu_2$. Further let

$$\begin{aligned} \zeta_1 &= \text{outcome}(\text{compile}(s1(e),t),\eta), \\ \zeta_2 &= \text{outcome}(\text{mksto}(t),\zeta_1), \\ \zeta_3 &= \text{outcome}(\text{compile}(s2(e),t+1),\zeta_2), \\ \zeta_4 &= \text{outcome}(\text{mkadd}(t),\zeta_3) \end{aligned}$$

so that $\zeta_4 = \text{outcome}(\text{compile}(e,t),\eta$, and we want to prove that

$$\zeta_4 =_t a(\text{ac},\nu,\eta).$$

We have

$$\begin{aligned} \zeta_1 &= outcome(compile(s1(e),t),\eta) \\ &=_t a(ac,\nu_1,\eta) && \text{Induction Hypothesis} \end{aligned}$$

and

$$c(ac,\zeta_1)=\nu_1. \qquad 3.8$$

Now

$$\begin{aligned} \zeta_2 &= outcome(mksto(t),\zeta_1) \\ &= a(t,c(\text{ac},\zeta_1),\zeta_1) && 3.12, 3.3, 3.11 \\ &= a(t,\nu_1),\zeta_1) && \textit{Substitution} \\ &=_{t+1} a(t,\nu_1,a(\text{ac},\nu_1,\eta)) && 4.5 \\ &=_{t+1,ac} a(t,\nu_1,\eta) && 4.5, 3.9 \end{aligned}$$

and

$$c(t,\zeta_2)=\nu_1 \qquad 3.8$$

Next

$$\begin{aligned} \zeta_3 =& \text{ outcome(compile(}s2(e),t+1),\zeta_2) \\ =& \;_{t+1}a(\text{ac},\nu_2,\zeta_2). \end{aligned}$$

Here we again use the induction hypothesis that $s2(e)$ is compiled correctly. In order to apply it, we need $c(\text{loc}(\nu,\text{map}),\zeta_2) = c(\nu,\xi)$ for each variable ν which is proved as follows:

$$\begin{aligned} c(\text{loc}(\nu,\text{map}),\zeta_2) &= c(\text{loc}(\nu),\text{map})a(t,\nu_1,\eta)) \text{ since loc}(\nu,\ \text{map}) < t \\ &= c(\text{loc}(\nu,\text{map}),\eta) \text{ for the same reason} \\ &= c(\nu,\xi) \text{ by the hypothesis of the theorem.} \end{aligned}$$

Now we can continue with

$$\zeta_3 =_{t+1} a(ac,\nu_2,a(t,\nu_1,\eta)) \qquad \text{by } 3.9$$

Finally,

$$\begin{aligned} \zeta_4 =& \text{ outcome(mkadd }(t),\zeta_3) \\ &= a(\text{ac},c(t,\zeta_3)+c(\text{ac},\zeta_3),\zeta_3) && 3.12, 3.4, 3. \\ &= a(\text{ac},\nu,\zeta_3) && \text{Definition of } \nu\text{, substitution} \\ &=_{t+1} a(\text{ac},\nu,a(\text{ac},\nu_2,a(t,\nu_1,\eta))) && 4.5 \\ &=_{t+1} a(\text{ac},\nu,a(t,\nu_1,\eta)) && 3.9 \\ &=_t a(\text{ac},\nu,\eta). && 3.9, 4.6, 4.7 \end{aligned}$$

This concludes the proof.

6 Remarks

The problem of the relations between source language and object language arithmetic is dealt with here by assuming that the + signs in formulas (2.1) and (3.11) which define the semantics of the source and object languages represent the same operation. Theorem 1 does not depend on any properties of this operation, not even commutativity or associativity.

The proof is entirely straightforward once the necessary machinery has been created. Additional operations such as subtraction, multiplication and division could be added without essential change in the proof.

For example, to put multiplication into the system the following changes would be required. 1. Add isprod(e), and $p1$(e), and $p2(e)$ to the abstract syntax of the source language.

2. Add a term

$$\textbf{if } \text{isprod}(e) \textbf{ then } \text{value}(p1(e), \zeta)\times \text{ value}(p2(e), \zeta)$$

to Equation (2.1).

3. Add

$$\text{isprod}(e) \wedge \Phi(p1(e)) \wedge \Phi(p2(e)) \supset \Phi(e)$$

to the hypotheses of the source language induction principle.

4. Add an instruction mul x and the three syntactical functions ismul(s) adr(r), mkmul(x) to the abstract syntax of the object language together with the necessary relations among them.

5. Add to the definition (3.11) of *step* a term

$$\textbf{else if } \text{ismul}(s) \textbf{ then } a(\text{ac}, c(\text{adr}(s), \eta) \times x(\text{ac}, \eta), \eta).$$

6. Add to the compiler a term

$$\textbf{if } isprod(e)\textbf{then}compile(p1(e), t) * mksto(t) * compile(p2(e), t + 1) * mkmul(t).$$

7. Add to the proof a case isprod(e) which parallels the case issum(e) exactly.

The following other extensions are contemplated. 1. Variable length sums.

2. Sequences of assignment statements.

3. Conditional expressions.

4. **go to** statements in the source language.

In order to make these extensions, a complete revision of the formalism will be required.

7 References

1. J. McCarthy, *Computer programs for checking mathematical proofs,* Proc. Sympos. Pure Math. Vol. 5, Amer. Math. Soc., Providence, R. I., 1962, pp. 219-227.

2. ———, "A basis for a mathematical theory of computation" in *Computer programming and formal systems,* edited by P. Braffort and D. Hershberg, North-Holland, Amsterdam, 1963.

3. ———, *Towards a mathematical theory of computation,* Proc. Internat. Congr. on Information Processing, 1962.

4. ———, *A formal description of a subset of Algol,* Proc. Conf. on Formal Language Description Languages, Vienna, 1964.