SEMANTIC CORRECTNESS OF A COMPILER FOR AN ALGOL-LIKE LANGUAGE

James A. Painter

Stanford University

Prepared for:

Advanced Research Projects Agency

10 March 1967

A. I. Memo No. 44

SEMANTIC CORRECTNESS OF A COMPILER FOR AN ALGOL-LIKE LANGUAGE

BY

JAMES A. PAINTER

MARCH 10, 1967

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
Memo No. 44

March 10, 1967

SEMANTIC CORRECTNESS OF A COMPILER FOR AN ALGOL-LIKE LANGUAGE

by James A. Painter

Abstract: This is a semantic proof of the correctness of a compiler. The abstract syntax and semantic definition are given for the language Mickey, an extension of Micro-algol. The abstract syntax and semantics are given for a hypothetical one-register single-address computer with 14 operations. A compiler, using recursive descent, is defined. Formal definitions are also given for state vector, a and c functions, and correctness of a compiler. Using these definitions, the compiler is proven correct.

This is a corrected version of a previously issued report. N/R

The research reported here was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-183).

Table of Contents

SECTION I

INTRODUCTION

In recent years there has been a widespread increase in the power and availability of computers. Concomitantly there has been increasing interest in the use of formal languages as programming tools. Much work has been done in studying various formalisms, particularly in the areas of language definition and syntax. The development of compiling algorithms proceeded concurrently, albeit on a less formal basis. The question of correctness of an algorithm for a particular language, or of a specific implementation for a language, has not received the same amount of attention. In fact the application of test cases and examination of the results is the only method presently used to validate a compiler. Obviously, this technique does not prove correctness, but assists in establishing a confidence level in the correctness. In this thesis, I will prove the correctness of a compiler. The correctness proof will be based upon the work of J. McCarthy [3], [4], [5].

A compiler cannot be said to be correct without considering its environment. The source language and the target machine must be considered, and correctness really concerns the triple (source, target, compiler). To that end, I will define a formal language and a machine with its associated compiler. The machine will be rather conventional.

This section is an informal discussion of the basic concepts, the language, and the machine. The next section is a more formal presentation of the definition of the language, the machine, and the compiler. The third section is a proof of the fundamental result.

Since the main point of the thesis is the proof of correctness, the language and machine are designed to simplify the proof rather than to be practical entities. Rudimentary as they are, however, they are universal in the sense of Turing machines. (See Appendix I for a proof of the universality of the language. The proof of the correctness of the compiler implies that the machine is universal, since it can execute correct machine language programs compiled from the universal program produced in Appendix I.) In the future, I hope that all compilers will be proven correct prior to implementation. This desire has introduced conflicting goals. The language and machine must model existing programming languages and computers realistically enough to further this aim. On the other hand, simplicity of language and machine facilitates the proof. This is the reason a new machine has been invented, rather than using an existing mathematical machine, e.g., infinite register machines [8], Turing machines [1], [2], pushdown automata [7], finite automata[6]. It is my feeling these machines are not adequate models for programming extant computers in that they either are not programmed at all or have very sparse instruction sets.

Again, since the correctness proof is central to this work, there is no advantage to be gained by specifying the syntax of the language concretely. The proof is semantic, and a syntax is required only to identify and produce elements of the language (program), to which semantics can be applied. That is, questions of character sets, infix vs. prefix vs. postfix notation, etc. are all irrelevant. Likewise, questions of ambiguity and parsing algorithms are also irrelevant. Therefore the concept of an abstract syntax was adopted [3], [4], [5]. Such a

syntax may be either analytic, i.e., one which enables the user to decompose programs into constituent primitives, or synthetic, i.e., one which enables the user to compose programs from primitives. Backus Normal Form is an example of a synthetic syntax. Because both the compiler and the semantic definition of a language require an analytic syntax only, that is the form of the syntax used in the language definition.

The basis of an abstract analytic syntax is that it consists of a collection of predicates and associated functions. The predicates are used to classify syntactic entities into classes, while the functions associated with each predicate are used to further decompose each entity into less complicated syntactic entities. This procedure recurs until an entity has been analyzed into primitive syntactic elements.

The semantic definition of a machine also requires an analytic syntax. A compiler, however, requires a synthetic syntax for the machine language which it produces. Hence in the machine definition, both an analytic and synthetic syntax are given. To recapitulate, a compiler uses an analytic syntax for the source language and a synthetic syntax for the target language. The semantic definitions of the source and target languages require an analytic syntax. Therefore, only an analytic syntax will be provided for the source language and both analytic and synthetic syntaxes for the machine. Because a syntax may be presented as either analytic or synthetic, there is a logical relation required between the analytic and synthetic forms in order to maintain regularity. This relation will be displayed when both forms of the syntax are given.

Along with abstract syntax, another basic concept is that of a state vector. The state vector ξ of a program, at a given time, is

the set of current assignments of values to the implicitly and explicitly referenced quantities of a program. A state vector is an (unordered) set of ordered pairs, each consisting of a symbol and the current value assigned to the symbol. The basic principle underlying the following work is that the meaning of a program is defined by its effect upon the state vector. That is, given a program π, an initial state vector ξ_0, and an input θ, the semantics of π can be made manifest by displaying the terminal state vector ξ after executing the program π with input θ, if any. For the actual execution of the program, either an actual computer must be built or an interpreter, i.e., program simulator, must be written. The procedure of actual execution raises irrelevant questions of specific implementations. It is possible to define a function of the program, the initial state vector, and the input, whose result, if defined, is the terminal state vector. This is in fact what is done. This function then is taken as the semantic definition of the language. The function is defined in terms of the syntactic elements of the abstract syntax. Clearly the same procedure can be used to define the semantics of a machine in terms of its syntactic elements, i.e., its operations. This also has been done.

In order to discuss state vectors, certain functions and relations are required. The function c[variable name, state vector] gives the value assigned to the named variable in the designated state vector. The function a[variable name, value, state vector] gives the state vector produced by assigning the designated value to the named variable, leaving all other components unchanged. These functions satisfy the following relations:

$$c[x,a[y,\alpha,\eta]] = \underline{\text{if}}\ x = y\ \underline{\text{then}}\ \alpha\ \underline{\text{else}}\ c[x,\eta]$$

$$a[x,\alpha,a[y,\beta,\eta]] = \underline{\text{if}}\ x = y\ \underline{\text{then}}\ a[x,\alpha,\eta]\ \underline{\text{else}}\ a[y,\beta,a[x,\alpha,\eta]]$$

$$a[x,c[x,\eta],\eta] = \eta$$

a and c are mnemonic for assign and contents respectively.

A notion of the partial equality for state vectors is also needed.

$$\zeta_1 =_A \zeta_2$$

is intended to mean that corresponding components of ζ_1 and ζ_2 are equal for all components x in A. Partial equality satisfies the following relations, where A and B are appropriate sets.

$\zeta_1 = \zeta_2$ is equivalent to $\zeta_1 = \{x|(x,y)$ is an element of ζ_1 or $\zeta_2\}\ \zeta_2$.

If $B \subseteq A$ and $\zeta_1 =_A \zeta_2$, then $\zeta_1 =_B \zeta_2$.

If $\zeta_1 =_A \zeta_2$ then $a[x,\alpha,\zeta_1] =_{A \cup \{x\}} a[x,\alpha,\zeta_2]$.

If $x \notin A$ then $a[x,\alpha,\zeta] =_A \zeta$.

If $\zeta_1 =_A \zeta_2$ and $\zeta_2 =_B \zeta_3$ then $\zeta_1 =_{A \cap B} \zeta_3$.

It must be noted that this concept of partial equality (to be formalized in the next section) is, in a sense, the inverse of the partial equality previously introduced in the literature [5]. There, the components were equal for all components except for those in A; while here, only those explicitly noted in A are necessarily equal. The problem with the other definition is defining "all" satisfactorily.

With these concepts, the notion of the correctness of a compiler can be described. Assume lang$[\pi,\xi,\theta]$ is the semantic definition of a

source language, and $\underline{mach}[\pi,\eta,\theta]$ is the semantic definition of a computer. Also $\underline{comp}[\pi]$ is the definition of a compiler from the source language to the machine language. Let A be a set of components of the state vectors which are essential in some sense, e.g., output, terminal variables, etc. This varies from machine to machine and language to language. Let ξ be an arbitrary source language state vector, and η an associated machine state vector. There is a functional relationship between ξ and η which corresponds to the requirement of proper starting conditions. The compiler is said to be correct then, if for arbitrary ξ (and associated η) and all admissible programs π, and arbitrary input θ,

$$\underline{lang}[\pi,\xi,\theta] =_A \underline{mach}[\underline{comp}[\pi],\eta,\theta] .$$

The intended interpretation of this equation is that the source language program and machine language program produced by the compiler have the same effect.

This notion of correctness satisfies the intuitive idea of correctness. There are other possible definitions, but this is the one chosen. The most obvious alternate definition, i.e., equality of output for the source language and machine language programs, was rejected for two reasons. First, the source language may not have any output statements, e.g., Algol 60. Second, and more important, the composition of correct subprograms to form a correct program is a very desirable goal. Equality of output does not guarantee equality of state vectors for entrance to a subsequent subprogram, hence composition would be a nontrivial operation.

On the other hand, it is counterintuitive to have a situation where the output is not equal. This will be controlled by defining the state vector and the essential set A to include the output. Then state vector partial equality over A will imply equal output.

It should be noted that there is a very close relation between lang and mach. In fact, both represent interpreter programs, albeit of different levels of language. Likewise comp can be considered as a transformation from one machine code to another. Then rewriting the basic equation for correctness in terms of mach1, mach2, and trans,

$$\underline{\text{mach1}}[\pi,\xi,\theta] =_A \underline{\text{mach2}}[\underline{\text{trans}}[\pi],\eta,\theta] .$$

But this equation can be considered in several other lights. For instance, if mach1 = mach2, then trans can be what is normally called an optimizer program. If mach1 and mach2 represent real computers, then trans represents a program performing machine to machine translation. It may happen that these alternate interpretations are more fruitful than that of the correctness of a compiler.

It is now appropriate to discuss the source language used in this thesis. The language Mickey (from Micro Extended) is a modification of Microalgol [4]. The basic changes have been the introduction of boolean expressions, read statements, and write statements. Boolean expressions have been added to ensure that conditional expressions have the intended form. The addition of the input-output statements has been made primarily to extend the power of the language. It also has the effect of preventing certain types of compilers from being considered

correct, namely those which at compile time interpret the program, develop results, and produce a program which merely loads the proper answers into the appropriate variables. These compilers have already been precluded by the requirement that the compiled program operate on arbitrary initial state vectors, but these additional statements reinforce the exclusion.

The read statement assumes a "read and destroy" type operation, i.e., the same item of data can be read only once. This assumption is made in order to conform with the operation of most current computer input devices. This is accomplished by assuming that the input is supplied by an "oracle" function θ, which if given the argument n returns the n^{th} input item. "Oracle" is used in the recursive function sense to emphasize that the input function is not necessarily computable. In order to obtain a different consecutive integer as argument each time, a counter (denoted "ctr") of the number of read statements executed is kept, and this counter is used as an argument. In order to accept all possible inputs, a functional (denoted "read") with arguments θ and ctr is used. It satisfies the equation

$$\text{read}(\theta,\text{ctr}) = \theta(\text{ctr}) .$$

If the input were not "read and destroy", e.g., sense switches, toggles, banks of relays, etc., then the same formalism could be used, except that "ctr" would not necessarily be incremented every time data is read.

The write statement assumes a component "out" of the state vector. When a write statement is executed, the desired data is appended to the string of the previous output in the component "out". The assumption

that write effects the state vector is made to prevent a correct compiler from remaining correct if it simply ignored all write statements. This would be true if write did not affect the state vector. The latter trouble could be obviated by changing the definition of correctness, e.g., requiring that the output produced by the source language program and the machine language program be identical. This raises the prior difficulty that there would be no simple way to correctly compose correct subprograms. Since equality of both the output and the state vectors is desirable, it seems easiest to consider the output as a component of the state vector by convention.

Mickey programs contain four types of statements, conditional go to statements, assignment statements, read statements, and write statements. The conditional go to statement is of the form:

if boolean expression then go to destination.

Destination is an integer n, denoting the n^{th} statement in the program. There are neither labels nor labeled statements in Mickey.

The assignment statement has a left side denoting the variable whose value is to be replaced. It also has a right side consisting of an arithmetic expression composed of a variable or a constant or a conditional expression or binary sums, differences, products, and quotients. The conditional expressions are of the form:

if boolean expression then arithmetic expression else arithmetic expression.

The arguments of sums, differences, products, and quotients are either constants or variables or further arithmetic expressions. The boolean expressions are two arithmetic expressions connected by the relation = or the relation < .

Read statements and write statements are of the form "Read variable" and "Write variable" .

A Mickey program consists of a sequence of statements. There are neither declarations nor blocks such as Algol 60 has. An n statement program halts when an attempt is made to execute the $n + 1^{st}$ statement. Hence, by convention no destination number in a go to can be larger than $n + 1$. The values which can be assigned to variables range over some domain. In the next section, the functions and predicates +, -, x, ÷, <, and = are assumed primitive in the semantic definitions of Mickey and the machine. They must be the same in both definitions however. If they are given their usual interpretation, the domain of values must be at least as complex as an ordered field, e.g., the rationals. There is no requirement for a complete ordered field, e.g., the reals, but there is nothing to preclude one either. If the machine is actually implemented as a finite physical entity, neither the reals nor the rationals can be realized. In this case the usual interpretation is to equate + with digital addition, etc. Then the domain is the set of digital numbers. In Appendix I, the assumption is made that the operations are the usual ones, hence the domain is either the rationals or the reals. The assumption that the domain of values is the reals is made in the main section to simplify definitions and proofs. Otherwise a whole host of irrelevant problems, e.g., memory wrap around, round off, etc., would

have to be incorporated into the language and machine definitions. These problems are usually not resolved in source language definitions in order to attain machine independence. The problems must be faced, however, when actually implementing a language upon a specific machine, and are the cause of programs written in a machine independent language actually being machine dependent.

The machine which is the target is intended to mirror a large class of contemporary computers. It uses a single address instruction format. Because Mickey has no arrays, neither index registers nor indirect addressing are included. All addressing is direct, except for a load immediate instruction. The intention here is to permit incorporation of constants into the program rather than the state vector. Otherwise the correct machine language state vector η corresponding to ξ is a function of the compiler and the source program π. This seems to be an unnecessary complication, although not insurmountable. However, a load immediate instruction, for which there is ample precedent in existing computers, permits a useful simplification.

The machine is assumed to have a single register, the accumulator, denoted "ac", and an instruction counter, denoted "IC". It recognizes 14 instructions. These are:

li	(load immediate)
load	(load accumulator)
add	(add)
sub	(subtract)
mpy	(multiply)
div	(divide)

sto	(store accumulator)
tra	(transfer control)
tmi	(transfer control if ac minus)
tze	(transfer control if ac zero)
rd	(read)
wrt	(write)
halt	(terminate execution)
synctra	(synchronize and transfer control)

The last instruction (synchronize and transfer control) is included for technical purposes only. It is intended to operate identically with the transfer control instruction, except that its occurrence can be noted. The proof of partial equality of the source and target state vectors will be by induction, i.e., they start partially equal, and are partially equal after every intermediate evaluation of Mickey. The conclusion will be either they are undefined together, or both are defined and partially equal. The natural stages for partial evaluation of Mickey are after the execution of each statement. Likewise, the natural stages for the machine are after the operation of each instruction. However, the machine instructions are much "finer" in their effect upon the state vector so that a sequence of them is required to duplicate the effect of a Mickey statement. Hence the two state vectors do not require the same number of partial evaluations, nor are they partially equal except at certain times, i.e., after executing the sequence of instructions corresponding to a statement. The synctra instruction is intended to mark these times, i.e., to synchronize when the state vectors should be compared. In essence it marks the end of a compiled Mickey statement.

A precedent for this type instruction is the "halt and transfer" instruction on the IBM 7090, which acts on the state vector, i.e., the 32,768 memory locations, AC, MQ, SI, sense lights, sense switches, keys, I/O, etc., identically to a "transfer" instruction. Clearly it does effect the timing circuits differently from the "transfer" instruction. However, a program, all of whose transfers were replaced by halt and transfers, would produce the same results as the original, albeit requiring a very patient console operator. Another precedent would be the "halt and proceed" and "no operation" pair of instructions upon the same computer.

The compiler is a rather primitive one using the method of recursive descent. It produces a program, starting at location 1 with absolute program addresses, e.g., the transfers are to absolute numeric values. Variables do not have absolute addresses, but are symbolic, e.g., load x . Temporary working storage is designated by addresses of the form t + n, where n is a positive integer. This can be construed as an array (or block) of temporaries, or as a set of symbolic addresses of a rather special form. This permits some flexibility in interpretation. Otherwise, the machine requires one special array not used, or usable, by Mickey. The compiler produces the appropriate code for each statement, separated by one or two synctra instructions to the next appropriate sequence. Two synctra's are required by the conditional go to .

Having an informal description of the language, machine, and compiler, the formal descriptions of the next section should be easier to follow.

SECTION II

FORMAL DEFINITIONS

In this section I will introduce formal definitions of state vectors with associated functions and relations; a semantic definition and analytic syntax for Mickey; and analytic syntax, synthetic syntax, relations between them, and semantic definition for the target machine. In addition the formal definition of correctness for a compiler will be given, as well as some preliminary results based upon these definitions. First, however, I will discuss the method used to specify abstract syntax and semantic definitions. This will be the major area where the discussion is entirely informal.

The analytic and synthetic syntaxes for a language will be given by tables. The analytic syntax table consists of a list of predicates and associated functions. The predicates are intended to classify entities. In actual compilers, they often are incorporated in the "scanner". The associated functions are intended to aid in the decomposition of a compound entity into simpler ones. Therefore the predicates for the primitive entities have no associated functions. The predicates which apply to each class of entity are assumed to be exhaustive and mutually exclusive. Predicates are consistently named by the scheme: ISname is the predicate for recognizing the entity named (or abbreviated) by name. It is read "is (unabbreviated name)?". The associated functions are given more or less mnemonic names.

The synthetic syntax table consists of a list of functions. These functions have the property of producing the appropriate syntactic entity.

In actual compilers, they are often incorporated in "generators". They are consistently named thus: MKname is the function producing the entity named (or abbreviated) by name. They are read "make (unabbreviated name)." In those cases where both analytic and synthetic syntaxes are given, a set of relations among them will be listed. These are the relations required to ensure regularity.

Semantic definitions will be presented in the form of sets of function definitions, some or all of which may be recursive. The functions will be defined formally by conditional expressions of the form if p then a else b where p is a predicate and a and b are functions or further conditional expressions. Certain functions and relations are primitive and assumed known. These are not otherwise defined. All other functions and relations are defined. (See Appendix II for an index of the definitions.)

Definition 1.0 The following table lists the primitive functions and relations not elsewhere defined. These are assumed known.

Function	Notation	Example
ARITHMETIC		
addition	+	3 + 5
subtraction	-	3 - 5
multiplication	x	3 x 5
division	÷	3 ÷ 5
SET THEORETIC		
union	U	A U B
difference	~	A ~ B

Function	Notation	Example
SET THEORETIC (continued)		
membership	$\in$	$x \in A$
non-membership	$\notin$	$x \notin A$
inclusion	$\subseteq$	$A \subseteq B$
intersection	$\cap$	$A \cap B$
ordered pair	(,)	(x,y)
LOGICAL		
conjunction	$\wedge$	$A \wedge B$
disjunction	$\vee$	$A \vee B$
negation	$\neg$	$\neg A$
implication	$\supset$	$A \supset B$
existential quantification	$\exists$	$(\exists x)(A(x))$
universal quantification	$\forall$	$(\forall x)(A(x))$
MISCELLANEOUS		
string concatenation	*	a * b
value of an expression denoting a constant	val	If the expression "16" denotes the integer also denoted by "4 x 4", then that integer equals val[16] .
conditional expression	if then else	if p then q else r
less than	<	3 < 5
equal	=	3 = 5
empty string	null	null[s]

This discussion of the syntax has been informal and intuitive in an attempt to keep attention focused upon the main goal. This thesis is primarily concerned with semantics, and the syntax is ancilliary. The syntax is required to provide elements to which semantics can be attached, but otherwise is not required. This is not an attempt to ignore or denigrate syntactic problems, but only a recognition of the emphasis required.

Some preliminary remarks are in order before defining state vectors. Let L be a programming language, with a semantic definition function $L[\pi,\xi,\theta]$ defined for program π written in L, state vector ξ, and input θ. There are certain quantities whose values change during evaluation of $L[\pi,\xi,\theta]$, i.e., during execution of the program π. Those mentioned explicitly by π are commonly called variables. Those required by the evaluation procedure itself are internal variables. They are implicit variables of π. A variable may be both implicit and explicit, e.g., an addressable accumulator on the IBM 7070, which is an implicit variable in the operation of adding to it, but could also be an explicit variable as an operand. For a given π, there can be only a finite number of explicit variables. Likewise, since the implicit variables depend upon the function $L[\pi,\xi,\theta]$, there is only a finite number of them.

In the following definitions, five functions are required which are intuitively well understood but are extremely variable from language to language. To denote the functions and their values, I shall use a nonstandard form of λ-notation. Specifically, I shall use $\lambda x \ldots z.name[x, \ldots, z]$ to denote the function and $\underline{name}[X, \ldots, Z]$ to denote the value of the function with arguments $X, \ldots, Z$. Thus

the only nonstandard notation is the use of a name and list of arguments in place of a form. The name is intended to be mnemonic for the function. The domain, range, and English description will be supplied for each function. The functions identified with the general λ functions will be supplied after Mickey and machine, the two primary languages of this thesis, are defined. In general, given any language, the functions corresponding to the λ functions can be supplied; but they are so variable and individual that no precise general notation seems possible. Thus I have adopted this circumlocution.

For the first λ function, $\lambda xy.range[x,y]$ has as arguments the name of a variable and a programming language. For each of the implicit and explicit variables in a programming language L, there is a certain set of values which can be associated with the variable. For the variable x, this set is the range of x, denoted range[x, L] . In some languages, e.g., Algol 60, range[x, Algol 60] is stated explicitly for each x . In some, e.g., FORTRAN, range[x, FORTRAN] is defined implicitly.

The appropriate definitions for state vectors can now be given. Let L be a programming language. Let $L[\pi,\xi,\theta]$ be a semantic definition for L, given program π, initial state vector ξ, and input θ . Let $\lambda xy.explicitvars[x,y]$ be the function that has as its domain $L[\pi,\xi,\theta]$ and programs written in L, and has as value the set of variables explicitly mentioned in the argument program. Let $\lambda xy.implicitvars[x,y]$ be a function whose arguments are a program π and the semantic definition function $L[\pi,\xi,\theta]$. The value of the function is the set of variables implicitly required in the evaluation of

$L[\pi,\xi,\theta]$ for π and for all ξ and θ. Let $\lambda xy.\text{iovariables}[x,y]$ be a function whose arguments are a program π and the semantic definition function $L[\pi,\xi,\theta]$. The value of the function is the set of variables implicitly required in the evaluation of input/output statements, but not required in evaluations of other statements. Clearly for a fixed π and L,

$$\underline{\text{iovariables}}[\pi, L] \subseteq \underline{\text{implicitvars}}[\pi, L]$$

Definition 1.1 ξ is a state vector means

$$\xi \subseteq \{(x, y) \mid (x, y) \text{ is an ordered pair}\} \wedge ((\forall u)(\forall v)(\forall w)(\forall x)$$
$$(((u, v) \in \xi \wedge (w, x) \in \xi \wedge u = w) \supset v = x)) .$$

Definition 1.2 ξ is a principal state vector (abbreviated p.s.v.) of π in L means

$$\xi \text{ is a state vector} \wedge (\forall x)(\forall y)((x, y) \in \xi \supset$$
$$(x \in \underline{\text{explicitvars}}[\pi, L] \cup \underline{\text{implicitvars}}[\pi,L] \wedge y \in \underline{\text{range}}[x, L])) .$$

Definition 1.3 ξ is an extended state vector (abbreviated e.s.v.) of π in L means

$$\xi \text{ is a state vector} \wedge (\exists \eta)(\eta \text{ is a } \underline{\text{p.s.v. of } \pi \text{ in } L} \wedge \eta \subseteq \xi) .$$

In the following where no misunderstanding is possible, the words of π in L or in L may be omitted. It is clear from the definitions that ξ is a p.s.v. of π in L implies ξ is a e.s.v. of π in L.

Definition 1.4 ξ is a proper extended state vector (abbreviated p.e.s.v.) of π in L means

(ξ is a e.s.v. of π in L) $\wedge \neg$(ξ is a p.s.v. of π in L).

Again, the words of π in L or in L may be omitted where no confusion is introduced. The reason for introducing extended state vectors is the intuitive idea that small programs should operate upon large computers as well as small computers. It also is required for composition of programs, since it is possible that the principal state vector of a composed pair of programs is larger than either individual p.s.v.

Definition 1.5 For a state vector ξ, the set variables (ξ) is defined by variables (ξ) = $\{x | (\exists y)((x, y) \in \xi)\}$.

Formal definitions of the functions a and c and the relation partial equality (denoted $=_A$) can now be given.

Definition 1.6 Let ξ be a state vector. Let $x \in$ variables (ξ). Then the c function is defined by

$c[x, \xi] = y$ iff $(x, y) \in \xi$.

Definition 1.7 $a[x, \alpha, \xi]$ = if $x \in$ variables (ξ) then

$(\xi \sim \{(x, c[x, \xi])\}) \cup \{(x, \alpha)\}$ else $\xi \cup \{(x, \alpha)\}$.

Definition 1.8 Let ξ_1 and ξ_2 be state vectors. Let A be a set. Then $\xi_1 =_A \xi_2$ (partial equality) is defined by the two conditions:

i. $A \subseteq \text{variables}(\xi_1) \cap \text{variables}(\xi_2)$

ii. $(\forall x)(x \in A \supset (\exists y)((x, y) \in \xi_1 \cap \xi_2))$.

Because the semantic definitions may include recursive functions, there is a possibility that applying the definitions to specific arguments may produce an undefined result rather than a state vector. Therefore, strong partial equality (denoted $\tilde{=}_A$), is defined.

Definition 1.9 Let ξ_1 and ξ_2 be state vector valued expressions (possibly undefined). Let A be a set. Then $\xi_1 \tilde{=}_A \xi_2$ (strong partial equality) means if ξ_1 and ξ_2 are both defined, then $\xi_1 =_A \xi_2$, while if either ξ_1 or ξ_2 is undefined, then the other is also.

Some properties of the functions a and c and the relation $=_A$ can now be stated.

Lemma 1.1 $c[x, a[y, \alpha, \xi]] = (\underline{\text{if}}\ x = y\ \underline{\text{then}}\ \alpha\ \underline{\text{else}}\ c[x, \xi])$.

Lemma 1.2 $a[x, \alpha, a[y, \beta, \xi]] = (\underline{\text{if}}\ x = y\ \underline{\text{then}}\ a[x, \alpha, \xi]\ \underline{\text{else}}\ a[y, \beta, a[x, \alpha, \xi]])$.

Lemma 1.3 $x \in \text{variables}(\xi) \supset (a[x, c[x, \xi], \xi] = \xi)$.

Lemma 1.4 Let $A = \text{variables}(\xi_1) \cup \text{variables}(\xi_2)$. Then $\xi_1 = \xi_2$ iff $\xi_1 =_A \xi_2$.

Lemma 1.5 If $B \subseteq A$ and $\xi_1 =_A \xi_2$ then $\xi_1 =_B \xi_2$.

Lemma 1.6 If $\xi_1 =_A \xi_2$ then $a[x, \alpha, \xi_1] =_{A \cup \{x\}} a[x, \alpha, \xi_2]$.

Lemma 1.7 Let $A \subseteq \text{variables}(\xi)$. If $x \notin A$, then $a[x, \alpha, \xi] =_A \xi$.

Lemma 1.8 If $\xi_1 =_A \xi_2$ and $\xi_2 =_B \xi_3$ then $\xi_1 =_{A \cap B} \xi_3$.

Proof of Lemmas 1.1 - 1.8:

For 1.1 it sufficies to examine cases.

Case I. $x = y$

$LHS = c[x, a[y, \alpha, \xi]]$

$= c[x, \xi \sim \{(y, c[y, \xi])\} \cup \{(y, \alpha)\}]$ by Def. 1.7

$= c[x, \xi \sim \{(x, c[x, \xi])\} \cup \{(x, \alpha)\}]$ by Case I: $x = y$

$= \alpha$ by Def. 1.6 since $(x, \alpha) \in \xi \sim \{(x, c[x, \xi])\} \cup \{(x, \alpha)\}$

$= RHS$

Case II. $x \neq y$

$LHS = c[x, \xi \sim \{(y, c[y, \xi])\} \cup \{(y, \alpha)\}]$ by Def. 1.7

$= c[x, \xi]$. If $(x, \beta) \in \xi - \{(y, c[y, \xi])\} \cup \{(y, \alpha)\}$, then $(x, \beta) \in \xi \sim \{(y, c[y, \xi])\}$. This implies $(x, \beta) \in \xi$. These last depend upon the Case II: $x \neq y$.

$LHS = RHS$

1.2 is also proven by cases. Let $\lambda = a[y, \beta, \xi]$.

Case I. $x = y$

$LHS = a[x, \alpha, a[y, \beta, \xi]]$

$= (\xi \sim \{(y, c[y, \xi])\} \cup \{(y, \beta)\}) \sim \{(x, c[x, \lambda])\} \cup \{(x, \alpha)\}$

$= (\xi \sim \{(x, c[x, \xi])\} \cup \{(x, \beta)\}) \sim \{(x, c[x, \lambda])\} \cup \{(x, \alpha)\}$

since $x = y$

$= \xi \sim \{(x, c[x, \xi])\} \cup \{(x, \beta)\} \sim \{(x, \beta)\} \cup \{(x, \alpha)\}$

$= \xi \sim \{(x, c[x, \xi])\} \cup \{(x, \alpha)\} = a[x, \alpha, \xi]$

$= RHS$

Case II. $x \neq y$

$$\text{LHS} = (\xi \sim \{(y, c[y, \xi])\} \cup \{(y, \beta)\}) \sim \{(x, c[x, \lambda])\} \cup \{(x, \alpha)\}$$

$$= (\xi \sim \{(x, c[x, \lambda])\} \cup \{(x, \alpha)\}) \sim \{(y, c[y, \xi])\} \cup \{(y, \beta)\}$$

This reordering is possible since $x \neq y$.

By lemma 1.1 $c[x, \lambda] = c[x, \xi]$ and $c[y, \xi] = c[y, a[x, \alpha, \xi]]$.

$$\text{LHS} = (a[x, \alpha, \xi]) \sim \{(y, c[y, a[x, \alpha, \xi]])\} \cup \{(y, \beta)\}$$

$$= a[y, \beta, a[x, \alpha, \xi]]$$

$$= \text{RHS}$$

For 1.3, let $x \in$ variables (ξ) .

$$\text{LHS} = a[x, c[x, \xi], \xi]$$

$$= \xi \sim \{(x, c[x, \xi])\} \cup \{(x, c[x, \xi])\}$$

$$= \text{RHS}$$

The condition $x \in$ variables (ξ) is required to guarantee that $c[x, \xi]$ is defined.

For the proof of 1.4, let $A =$ variables $(\xi_1) \cup$ variables (ξ_2) . For the left to right part, assume $\xi_1 = \xi_2$. From $\xi_1 = \xi_2$, $A =$ variables $(\xi_1) \cap$ variables (ξ_2) . From the definition of variables (Def. 1.5), assuming $x \in$ variables (ξ_1), $(\exists y)((x, y) \in \xi_1)$. From the definition of $=$, $(x, y) \in \xi_2$. Since the same argument can be used with ξ_2 interchanged with ξ_1, for all $x \in$ variables $(\xi_1) \cup$ variables $(\xi_2) = A$

$$\xi_1 = \{x\} \xi_2$$

Thus $\xi_1 =_A \xi_2$. For the reverse implication, assume $\xi_1 =_A \xi_2$. Let

$(x, y) \in \xi_1$. By Def. 1.8 and the assumption, $(x, y) \in \xi_2$. Hence $\xi_1 \subseteq \xi_2$. Let $(x, y) \in \xi_2$. By Def. 1.8 and the assumption, $(x, y) \in \xi_1$. Hence $\xi_2 \subseteq \xi_1$. Therefore $\xi_1 = \xi_2$.

For the proof of 1.5, assume $B \subseteq A$ and $\xi_1 =_A \xi_2$. Let $x \in B$. Then $x \in A$. From Def. 1.8, and the assumption $\xi_1 =_A \xi_2$

$$(\exists y)((x, y) \in \xi_1 \cap \xi_2) .$$

Hence $\xi_1 =_B \xi_2$.

To prove 1.6, let $\xi_1 =_A \xi_2$. If $x \in A$, then $A \cup \{x\} = A$, and the result follows immediately from Def. 1.7 and Def. 1.8. Assume $x \notin A$. Then $y \in A \cup \{x\}$ implies $y \in A$ or $y = x$. If $y = x$, then by Def. 1.7, $(y, \alpha) \in a[x, \alpha, \xi_1] \cap a[x, \alpha, \xi_2]$. If $y \in A$, and hence, $x \neq y$, then by Def. 1.8, $(\exists z)((y, z) \in \xi_1 \cap \xi_2)$. Using Def. 1.7, then $(y, z) \in a[x, \alpha, \xi_1] \cap a[x, \alpha, \xi_2]$. These two cases prove 1.6.

To prove 1.7, let $y \in A \subseteq$ variables (ξ) . Then $x \neq y$. By Def. 1.5, $(\exists z)((y, z) \in \xi)$. But then by Def. 1.7, $(y, z) \in a[x, \alpha, \xi]$. Hence $(\exists z)((y, z) \in \xi \cap a[x, \alpha, \xi])$. Then by Def. 1.8,

$$a[x, \alpha, \xi] =_A \xi .$$

To prove 1.8, assume $\xi_1 =_A \xi_2$ and $\xi_2 =_B \xi_3$. If $A \cap B$ is empty, Def. 1.8 is satisfied vacuously. Let $x \in A \cap B$. Applying Def. 1.8

twice, $(x, y_1) \in \xi_1$, $(x, y_2) \in \xi_2$, and $(x, y_3) \in \xi_3$; and $y_1 = y_2$ and $y_2 = y_3$. From transitivity, $y_1 = y_3$, thus satisfying Def. 1.8. Hence $\xi_1 =_{A \cap B} \xi_3$.

The formal definition of correctness for a compiler can now be stated. Let L, $L[\pi, \xi, \theta]$, and $\lambda xy.\text{explicitvars}[x, y]$ be as above. Let M be a machine language with semantic definition $M[\pi, \eta, \theta]$ for program π, initial state vector η, and input θ. Let $C[\pi]$ be a compiler function whose domain is the set of programs written in L and whose range is a subset of the set of programs in M. Let $\lambda x.\text{sequencer}[x]$ be a function whose domain is a semantic definition of a language and whose range is the variable used to control sequencing between statements. Thus sequencer[IBM 7090] = "instruction counter".

Definition 1.10 The compiler $C[\pi]$ is correct if

$$(\forall\pi)(\forall\theta)(\forall\xi)(\forall\eta)((\xi \text{ is a p.s.v. of } \pi \text{ in L}$$
$$\wedge\ \eta \text{ is a p.s.v. of } C[\pi] \text{ in M}$$
$$\wedge\ c[\underline{\text{sequencer}}[L], \xi] = c[\underline{\text{sequencer}}[M], \eta] = 1$$
$$\wedge\ A = \underline{\text{explicitvars}}[\pi, L] \cup \underline{\text{iovariables}}[\pi, L]$$
$$\wedge\ \xi =_A \eta) \supset L[\pi, \xi\ \theta] \cong_A M[C[\pi], \eta, \theta]) .$$

The language Mickey involves a syntax and a semantic definition. Since Mickey is the source language, only an analytic syntax will be given. It is given by the following table. The symbol π denotes a program, the symbol s denotes a statement, the symbol ae denotes an arithmetic expression, i.e., quantities which can appear on the right

side of assignment statements or as arguments of the relation in a Boolean expression while be denotes a Boolean expression, i.e., in the range of the function prop .

Definition 1.11 The analytic syntax of Mickey is presented in the following table:

predicate	associated function		
isprogramM[π]	firstate[π]	restprog[π]	
isgoto[s]	prop[s]	dest[s]	
isassign[s]	left[s]	right[s]	
isread[s]	argum[s]		
iswrite[s]	argum[s]		
isvar[ae]			
isconst[ae]			
issum[ae]	arg1[ae]	arg2[ae]	
isdiff[ae]	arg1[ae]	arg2[ae]	
isprod[ae]	arg1[ae]	arg2[ae]	
isquot[ae]	arg1[ae]	arg2[ae]	
iscond[ae]	prop[ae]	ante[ae]	consq[ae]
isequal[be]	arg1[be]	arg2[be]	
isless[be]	arg1[be]	arg2[be]	

The functions are named for first statement, rest of program, proposition, destination, argument, argument 1, argument 2, antecedent, and consequent.

The following definitions provide predicates for recognizing arithmetic expressions, Boolean expressions, statements, and expressions contained in programs.

isae[e] iff isvar[e] ∨ isconst[e] ∨
((issum[e] ∨ isdiff[e] ∨ isprod[e] ∨ isquot[e]) ∧
isae[arg1[e]] ∧ isae[arg2[e]]) ∨
(iscond[e] ∧ isbe[prop[e]] ∧ isae[ante[e]] ∧
isae[consq[e]]).

isbe[e] iff isae[arg1[e]] ∧ isae[arg2[e]] ∧
(isless[e] ∨ isequal[e]) .

s is a statement iff $(\exists\pi)$(isprogramM(π) ∧ s = firstate[π]) .

e is an arithmetic expression contained in an arithmetic expression E iff

isae[e] ∧ isae[E] ∧ (e = E ∨ $(\exists F)$(isae[F] ∧
e is an arithmetic expression contained in an arithmetic expression F ∧
(((issum[E] ∨ isdiff[E] ∨ isprod[E] ∨ isquot[E]) ∧
(F = arg1[E] ∨ F = arg2[E])) ∨ (iscond[E] ∧
(F = ante[E] ∨ F = consq[E] ∨ F = arg1[prop[E]] ∨ F = arg2[prop[E]])))))).

e is a proper arithmetic expression contained in a program π iff

isae[e] ∧ $(\exists\tau)(\exists\tau')(\exists s)(\exists E)$(s is a statement ∧ $\pi = \tau * s * \tau'$ ∧
isprogramM[π] ∧ isae [E] ∧ τ, τ' are (possibly null)

sequences of statements ∧

((isassign[s] ∧ E = right[s]) ∨ (isgoto[s] ∧ (E = arg1[prop[s]] ∨
E = arg2[prop[s]]))) ∧

e is an arithmetic expression contained in an arithmetic expression E).

e is a proper Boolean expression contained in a program π iff

isbe[e] $\wedge$ $(\exists\tau)(\exists\tau')(\exists s)(\exists E)$ (s is a statement $\wedge$ $\pi = \tau * s * \tau'$ $\wedge$ isprogramM[π] $\wedge$ τ, τ' are (possibly null) sequences of statements $\wedge$ ((isgoto[s] $\wedge$ e = prop[s]) $\vee$ (isae[E] $\wedge$ iscond[E] $\wedge$ e = prop[E] $\wedge$ E is a proper arithmetic expression contained in a program π)))

Axiom 1.11.1 The analytic syntax of Mickey satisfies the following five conditions:

(a). isprogramM[π] $\supset$ π = firstate[π] * restprog[π]

(b). $\neg$ null[π] $\wedge$ null[restprog[π]] $\supset$ π = firstate[$\pi * \tau$]

(c). null[$\pi * \tau$] iff null[π] $\wedge$ null[τ]

(d). The predicates in the following sets are exhaustive and mutually exclusive: {isgoto, isassign, isread, iswrite}, {isvar, isconst, issum, isdiff, isprod, isquot, iscond}, and {isequal, isless}, when applied to statements, arithmetic expressions, and Boolean expressions respectively.

(e). isassign[s] $\supset$ isvar[left[s]]

isread[s] $\supset$ isvar[argum[s]]

iswrite[s] $\supset$ isvar[argum[s]]

The semantics of Mickey are given by a set of recursively defined functions which determine the state (if defined) in which a Mickey program π terminates if entered in state ξ. These functions are:

Name	Purpose
Mickey	Basic definition, steps from one statement to the next.
execute	Compute result of executing single statement.
avalue	Compute arithmetic value of expression e .
bvalue	Compute Boolean value of expression e .
state	Compute n^{th} statement of program.
end	Predicate to determine if program evaluation should end. It is true iff there are fewer than n statements in π .

Definition 1.12 The semantic definition of Mickey is the following set of functions and predicates:

Mickey[π, ξ, θ] = if end[c[sn, ξ], π] then ξ else
Mickey[π, execute[π, ξ, θ], θ]

execute[π, ξ, θ] = (λs. if isgoto[s] then a[sn,
if bvalue[prop[s], ξ] then dest[s] else c[sn, ξ] +1, ξ] else
a[sn, c[sn, ξ] +1, (if isassign[s] then a[left[s],
avalue[right[s], ξ], ξ] else if isread[s] then
a[ctr, c[ctr, ξ] +1, a[argum[s], read[θ, c[ctr, ξ]], ξ]] else
if iswrite[s] then a[out, c[out, ξ] * c[argum[s], ξ], ξ])])
(state[c[sn, ξ], π])

avalue[e, ξ] = if isvar[e] then c[e, ξ] else

if isconst[e] then val[e] else

if issum[e] then (avalue[argl[e], ξ] + avalue[arg2[e], ξ]) else

if isdiff[e] then (avalue[argl[e], ξ] - avalue[arg2[e], ξ]) else

if isprod[e] then (avalue[argl[e], ξ] x avalue[arg2[e], ξ]) else

if isquot[e] then (avalue[argl[e], ξ] ÷ avalue[arg2[e], ξ]) else

if iscond[e] then (if bvalue[prop[e], ξ] then

avalue[ante[e], ξ] else avalue[consq[e], ξ])

bvalue[e, ξ] = if isequal[e] then (avalue[argl[e], ξ] =

avalue[arg2[e], ξ]) else

if isless[e] then (avalue[argl[e], ξ] < avalue[arg2[e], ξ])

state[n, π] = if n = 1 then firstate[π] else state[n-1, restprog[π]]

end[n, π] = if n = 1 then null[π] else end[n-1, restprog[π]]

One method of proof for propositions about Mickey arithmetic expressions, e.g., quantities in the range of the function right[s], is induction.

Definition 1.13 Let Φ be a proposition about arithmetic expressions, and

isvar[e] ⊃ Φ[e]

isconst[e] ⊃ Φ[e]

issum[e] ∧ Φ[argl[e]] ∧ Φ[arg2[e]] ⊃ Φ[e]

isdiff[e] ∧ Φ[argl[e]] ∧ Φ[arg2[e]] ⊃ Φ[e]

isprod[e] ∧ Φ[argl[e]] ∧ Φ[arg2[e]] ⊃ Φ[e]

$$\text{isquot}[e] \wedge \Phi[\text{arg1}[e]] \wedge \Phi[\text{arg2}[e]] \supset \Phi[e]$$

$$\text{iscond}[e] \wedge (\text{isequal}[\text{prop}[e]] \vee \text{isless}[\text{prop}[e]]) \wedge$$
$$\Phi[\text{arg1}[\text{prop}[e]]] \wedge \Phi[\text{arg2}[\text{prop}[e]]] \wedge \Phi[\text{ante}[e]] \wedge$$
$$\Phi[\text{consq}[e]] \supset \Phi[e]$$

Then the induction principle for arithmetic expressions of Mickey asserts $\Phi[e]$ for all e.

Axiom 1.13.1 The analytic syntax of Mickey satisfies the induction principle for arithmetic expressions.

We are now ready to present the applicable definitions for the machine. The intention is to define a machine similar to a large class of contemporary computers. The machine is a one register, single address machine. It recognizes 14 instructions. The addresses relevant to the program are positive integers, while those pertaining to "data" are symbolic. Data and the program can be considered separate since the compiler produces read-only code. There is no requirement to separate them however. The symbolic data addresses can be converted to numeric addresses by essentially adding an assembly program to the compiler.

In the following syntax table, α denotes a numeric expression, i.e., a constant, x denotes a variable, i.e., a data address, n denotes an integer used as a program address, and s denotes a statement, i.e., an instruction.

Definition 1.14 The analytic and synthetic syntax of machine is presented by the following table.

operation	predicate	analytic function	synthetic function
li α	isli (s)	arg (s)	mkli (α)
load x	isload (s)	addr (s)	mkload (x)
add x	isadd (s)	addr (s)	mkadd (x)
sub x	issub (s)	addr (s)	mksub (x)
mpy x	ismpy (s)	addr (s)	mkmpy (x)
div x	isdiv (s)	addr (s)	mkdiv (x)
sto x	issto (s)	addr (s)	mksto (x)
tra n	istra (s)	addr (s)	mktra (n)
tmi n	istmi (s)	addr (s)	mktmi (n)
tze n	istze (s)	addr (s)	mktze (n)
rd x	isrd (s)	arg (s)	mkrd (x)
wrt x	iswrt (s)	arg (s)	mkwrt (x)
halt	ishalt (s)	--------	mkhalt
syntra n	issyntra (s)	addr (s)	mksyntra (n)

In addition the predicate isprogramm[π] and associated functions first[π] and rest[π] are part of the analytic syntax of machine.

Axiom 1.14.1 The analytic syntax of machine satisfies the following three conditions:

(a). π = first[π] * rest[π]

(b). $\neg$ null[π] $\wedge$ null[rest[π]] $\supset$ π = first[π * τ]

(c). The predicates in the following set are exhaustive and mutually exclusive: {isli, isload, isadd, issub, ismpy, isdiv, issto, istra, istmi, istze, isrd, iswrt, ishalt, issyntra}, when applied to instructions.

The previous table contained two different syntaxes. It is obvious that certain logical relations between the components are required if they are defining the same object. This condition is called regularity [3].

Definition 1.15 The regular condition for analytic and synthetic syntax of machine (shortly regularity of machine) is presented by the following tables.

1.15.1
$$isli(mkli(\alpha))$$
$$\alpha = arg(mkli(\alpha))$$
$$isli(s) \supset s = mkli(arg(s))$$
$$null(rest(mkli(\alpha)))$$
$$isli(s) \supset first(s) = s$$

1.15.2
$$isload(mkload(x))$$
$$x = addr(mkload(x))$$
$$isload(s) \supset s = mkload(addr(s))$$
$$null(rest(mkload(x)))$$
$$isload(s) \supset first(s) = s$$

1.15.3
$$isadd(mkadd(x))$$
$$x = addr(mkadd(x))$$
$$isadd(s) \supset s = mkadd(addr(s))$$
$$null(rest(mkadd(x)))$$
$$isadd(s) \supset first(s) = s$$

1.15.4 issub(mksub(x))

x = addr(mksub(x))

issub(s) ⊃ s = mksub(addr(s))

null(rest(mksub(x)))

issub(s) ⊃ first(s) = s

1.15.5 ismpy(mkmpy(s))

x = addr(mkmpy(x))

ismpy(x) ⊃ s = mkmpy(addr(s))

null(rest(mkmpy(x)))

ismpy(s) ⊃ first(s) = s

1.15.6 isdiv(mkdiv(x))

x = addr(mkdiv(x))

isdiv(s) ⊃ s = mkdiv(addr(s))

null(rest(mkdiv(x)))

isdiv(s) ⊃ first(s) = s

1.15.7 issto(mksto(x))

x = addr(mksto(x))

issto(s) ⊃ s = mksto(addr(s))

null(rest(mksto(x)))

issto(s) ⊃ first(s) = s

1.15.8 istra(mktra(n))

n = addr(mktra(n))

istra(s) ⊃ s = mktra(addr(s))

null(rest(mktra(n)))

istra(s) ⊃ first(s) = s

1.15.9 istmi(mktmi(n))

n = addr(mktmi(n))

istmi(s) ⊃ s = mktmi(addr(s))

null(rest(mktmk(n)))

istmi(s) ⊃ first(s) = s

1.15.10 istze(mktze(n))

n = addr(mktze(n))

istze(s) ⊃ s = mktze(addr(s))

null(rest(mktze(n)))

istze(s) ⊃ first(s) = s

1.15.11 isrd(mkrd(x))

x = arg(mkrd(x))

isrd(s) ⊃ s = mkrd(arg(s))

null(rest(mkrd(x)))

isrd(s) ⊃ first(s) = s

1.15.12 iswrt(mkwrt(x))

x = arg(mkwrt(x))

iswrt(s) ⊃ s = mkwrt(arg(s))

null(rest(mkwrt(x)))

iswrt(s) ⊃ first(s) = s

1.15.13 ishalt(mkhalt)

ishalt(s) ⊃ s = mkhalt

null(rest(mkhalt))

ishalt(s) ⊃ first(s) = s

1.15.14 issyntra(mksyntra(n))

n = addr(mksyntra(n))

issyntra(s) ⊃ s = mksyntra(addr(s))

null(rest(mksyntra(n)))

issyntra(s) ⊃ first(s) = s

The following definition gives the functions used to define the semantics of machine. Conceptually these should be defined ab initio. However to facilitate the proof, they are not defined quite as expected; for instance, the occurence of both a syntra and a tra instruction. However, examination of the definitions shows that the machine works quite as expected. The functions are:

Name	Purpose
machine	Basic definition, steps from one block of code to the next.
xeq	Steps through a block of code. Returns to machine at halts and syntras. Executes all transfers of control.
step	Executes all non-transfer instructions.
presentinst	Obtain the n^{th} instruction.

Definition 1.16 The semantic definition of machine is the following set of functions and predicates:

machine[π, ξ, θ] = if ishalt[presentinst[π, c[IC, ξ]]] then ξ

else machine[π, xeq[π, ξ, θ], θ]

```
xeq[π, ξ, θ] = (λi. if ishalt[i] then ξ else if issyntra[i]
        then a[IC, addr[i], ξ] else if istra[i]
        then xeq[π, a[IC, addr[i], ξ], θ] else if istze[i]
        then xeq[π, a[IC, if c[ac, ξ] = 0 then addr[i]
        else c[IC, ξ] + 1, ξ], θ] else if istmi[i]
        then xeq[π, a[IC, if c[ac, ξ] < 0 then addr[i]
        else c[IC, ξ] +1, ξ], θ] else xeq[π, a[IC, c[IC, ξ] +1,
        step[i, ξ, θ]], θ])(presentinst[π, c[IC, ξ]])
step[i, ξ, θ] = if isload[i] then a[ac, c[addr[i], ξ], ξ] else
        if islі[i] then a[ac, arg[i], ξ] else
        if issto[i] then a[addr[i], c[ac, ξ], ξ] else
        if isadd[i] then a[ac, c[ac, ξ] + c[addr[i], ξ], ξ] else
        if issub[i] then a[ac, c[ac, ξ] - c[addr[i], ξ], ξ] else
        if ismpy[i] then a[ac, c[ac, ξ] x c[addr[i], ξ], ξ], else
        if isdiv[i] then a[ac, c[ac, ξ] ÷ c[addr[i], ξ], ξ] else
        if isrd[i] then a[arg[i], read[θ, c[ctr, ξ]], ξ] else
        if iswrt[i] then a[out, c[out, ξ] * c[arg[i], ξ], ξ]
presentinst[π, n] = if n = 1 then first[π]
        else presentinst[rest[π], n-1]
```

Now the compiler can be defined. It operates by the method of recursive descent. An initial pass is made to construct a table. If the argument for entering the table is the Mickey statement number, the value will be the machine program location of the first instruction of the code corresponding to the Mickey statement. The functions are:

Name	Purpose
Compile	Starts compilation at location 1 and produces the one and only halt for termination.

Name	Purpose
coml	Steps through the program statement by statement. This is the basic recursion.
com2	Produces the code for each statement.
acom	Produces the code for arithmetic expressions.
bcom	Produces the code for Boolean expressions.
lengths	Computes the number of instructions for a statement.
lengthe	Computes the number of instructions for an arithmetic expression.
lengthb	Computes the number of instructions for a Boolean expression.
tlu	Table look up to find machine address corresponding to Mickey statement number.
table	Produces table used by tlu .
tabl	Basic recursion used in producing table .
t	Produces n^{th} temporary storage element symbol.

It should be noted that 0(1) is used to represent true (false) in the machine by the program produced by the compiler. Also, temporary storage is denoted by variables of the form "t + k" where k is a non-negative integer. The form is produced by the function t and is intended to suggest an array, but the only condition really required is that t[k] = t[j] iff k = j .

Definition 1.17 The compiler is the following set of functions and predicates:

Compile[π] = coml[π, 1, table[π]] * mkhalt

```
com1[π, n, ℓ] = if null[π] then π else
    com2[firstate[π], n, ℓ] * mksyntra[n + 1 +
    lengths[firstate[π]]] * com1[restprog[π],
    n + 1 + lengths[firstate[π]], ℓ]

com2[s, n, ℓ] = if isgoto[s] then bcom[prop[s], n, 0]
    * mktze[n + lengths[s] -1] * mktra[n + lengths[s]]
    * mksyntra[tlu[dest[s], ℓ]] else if
    isassign[s] then acom[right[s], n, 0] *
    mksto[left[s]] else if isread[s] then
    mkrd[arg[s]] * mkli[1] * mkadd[ctr] * mksto
    [ctr] else if iswrite[s] then mkwrt[arg[s]]

acom[e, j, k] = if isconst[e] then mkli[e] else
    if isvar[e] then mkload[e] else if iscond[e]
    then bcom[prop[e], j, k] * mktze[j + lengthb[prop[e]] + 2 +
    lengthe[consq[e]]] * acom[consq[e],
    j + lengthb[prop[e]] + 1, k] * mktra[
    j + lengthe[e]] * acom[ante[e], j + lengthb
    [prop[e]] + 2 + lengthe[consq[e]], k] else acom[arg2[e],
    j, k] * mksto[t[k]] * acom[arg1[e], j + lengthe[arg2[e]]
    + 1, k + 1] * (if issum[e] then mkadd[t[k]] else if
    isdiff[e] then mksub[t[k]] else if isprod[e] then
    mkmpy[t[k]] else if isquot[e] then mkdiv[t[k]])
```

```
bcom[e, j, k] = acom[arg2[e], j, k] * mksto[t[k]]
    * acom[argl[e], j + 1 + lengthe[arg2[e]], k + 1] *
    mksub[t[k]] * (if isequal[e] then mktze[j +
    lengthe[arg2[e]] + lengthe[argl[e]] + 4] *
    mkli[1] else if isless[e] then mktmi[j +
    lengthe[arg2[e]] + lengthe[argl[e]] + 5] * mkli
    [1] * mktra[j + lengthe[arg2[e]] + lengthe[argl[e]]
    + 6] * mkli[0])

lengths[s] = if isgoto[s] then 3 + lengthb[prop[s]]
    else if isassign[s] then 1 + lengthe[right[s]]
    else if isread[s] then 4 else if iswrite[s] then 1

lengthe[e] = if isconst[e] then 1 else if isvar[e] then
    1 else if iscond[e] then 2 + lengthe[ante[e]] +
    lengthe[consq[e]] + lengthb[prop[e]] else if
    issum[e] then 2 + lengthe[argl[e]] + lengthe[arg2[e]]
    else if isdiff[e] then 2 + lengthe[argl[e]] + lengthe[arg2[e]]
    else if isprod[e] then 2 + lengthe[argl[e]] + lengthe[arg2[e]]
    else if isquot[e] then 2 + lengthe[argl[e]] + lengthe[arg2[e]]

lengthb[e] = if isequal[e] then 4 + lengthe[arg2[e]]
    + lengthe[argl[e]] else if isless[e] then 6 +
    lengthe[arg2[e]] + lengthe[argl[e]]

tlu[n, t] = if n = 1 then first[t] else if
    null[rest[t]] then tlu[n-1, t] else
    tlu[n-1, rest[t]]
```

table[π] = tabl[1, π]

tabl[n, π] = if null[π] then n else n *
tabl[n + lengths[firstate[π]] + 1, restprog[π]]

t[n] = "t+" * "n" where "t" and "+" are symbols such that for all integers n, $\neg$ isvar[t[n]] .

Now that the machine language and compiler have been defined, it is possible to define specifically all of the general functions used in the definition of correctness for a compiler (Def. 1.10). The general λ functions for the languages Mickey and machine will be given. The function vars[π] calls a function varsa with the program and the empty set as arguments. varsa is recursive and adds the variables in the first statement to the set as it steps through the program. It uses the functions varsb and varsc (which correspond to vars and varsa respectively) to obtain the variables in expressions.

Definition 1.18 The general λ functions are defined for Mickey and machine languages.

range[x, Mickey]	= strings of reals	if x = out
	= positive integers	if x $\in$ {sn, ctr}
	= reals	otherwise
range[x, machine]	= strings of reals	if x = out
	= positive integers	if x $\in$ {IC, ctr}
	= reals	otherwise

```
explicitvars[π, Mickey] = vars[π]

explicitvars[π, machine] = machvars[π]

implicitvars[π, Mickey] = {sn} ∪ iovars[π]

implicitvars[π, machine] = {IC, ac} ∪ machiovars[π]

iovariables[π, Mickey] = {ctr, out}

iovariables[π, machine] = {ctr, out}

sequencer[Mickey] = sn

sequencer[machine] = IC

vars[π] = varsa[π, {}]

varsa[π, S] = if null[π] then S else
      if isgoto[firstate[π]] then varsa[restprog[π],
      S ∪ varsb[arg1[prop[firstate[π]]]]
      ∪ varsb[arg2[prop[firstate[π]]]]] else
      if isassign[firstate[π]] then varsa[restprog[π],
      S ∪ {left[firstate[π]]} ∪ varsb[right[firstate[π]]]]
      else varsa[restprog[π], S ∪ {argum[firstate[π]]}]
varsb[ae] = varsc[ae, {}]
varsc[ae, S] = if isvar[ae] then S ∪ {ae} else
      if isconst[ae] then S else
      if iscond[ae] then S ∪ varsb[arg1[prop[ae]]]
      ∪ varsb[arg2[prop[ae]]] ∪ varsb[ante[ae]] ∪ varsb[consq[ae]]
      else S ∪ varsb[arg1[ae]] ∪ varsb[arg2[ae]]

machvars[π] = machvarsa[π, {}]
machvarsa[π, S] = if null[π] then S else if isload[first[π]] ∨
      isadd[first[π]] ∨ issub[first[π]] ∨ ismpy[first[π]] ∨
      isdiv[first[π]] ∨ issto[first[π]] then machvarsa[rest[π],
```

S ∪ {addr [first[π]]}] else if isrd[first[π]] ∨

iswrt[first[π]] then machvarsa[rest[π], S ∪ {arg[first[π]]}]

else machvarsa[rest[π], S]

In order to manipulate lists and sublists, i.e., programs and subprograms, several auxiliary functions are required. The symbol NIL will be used to denote the empty list. The following three functions are defined. last[π] returns the last item of a non-empty list, NIL otherwise. #[π] counts the number of statements in a Mickey program π, while #instr[τ] does the same for a machine program τ .

Definition 1.19

last[π] = if null[rest[π]] then π else last[rest[π]]

#[π] = numb[π, 0]

numb[π, n] = if null[π] then n else numb[restprog[π], n+1]

#inst[π] = num[π, 0]

num[π, n] = if null[π] then n else num[rest[π], n+1]

Axiom 1.19.1

isprogramM[π] ⊃ #[π] is defined.

isprogramm[π] ⊃ #inst[π] is defined.

This concludes the definitions. Several lemmas based upon the previous definitions will now be proven. These lemmas will be used in the next section in the proof of the fundamental result, i.e., the correctness of the compiler Compile .

The next lemma shows that vars[π] contains all those variables whose initial values may be required to be known. In fact, it contains all variables explicitly mentioned in π, but the lemma only shows it contains those variables which might be evaluated before being modified. In general, this property could be taken as a hypothesis; but, given a specific definition for $\lambda xy.$ explicitvars[x, y], it seems appropriate to prove it.

Lemma 1.9 Let e be a proper arithmetic expression contained in a program π. Let s be a statement. Let τ, τ' be (possibly null) sequences of statements such that $\pi = \tau * s * \tau'$. Let isprogramM[π]. Then isvar[e] $\supset$ e $\in$ vars[π] and iswrite[s] $\supset$ argum[s] $\in$ vars[π].

Proof: First to show e $\in$ vars[π], assuming isvar[e]. By Def. 1.11, there exists a statement s and an arithmetic expression E such that

(i) isassign[s] $\wedge$ E = right[s]

or (ii) isgoto[s] $\wedge$ (E = arg1[prop[s]] $\vee$ E = arg2[prop[s]])

and e is contained in E.

The following four assertions will be shown, and then the result e $\in$ vars[π] follows.

(a) e contained in E implies e $\in$ varsb[E].

(b) varsa[π, S] $\subseteq$ varsa[π, S $\cup$ T].

(c) (i) or (ii) imply varsb[E] $\subseteq$ varsa[s, {}].

(d) varsa[s, {}] $\subseteq$ vars[$\tau * s * \tau'$].

The proof of (a) will be by induction upon E using Def. 1.13. isvar[E] implies e = E, hence varsb[E] = {e} by Def. 1.18.

(a) is vacuously true for isconst[E] .

Assume (a) true for the appropriate induction hypotheses. For issum[E], varsb[E] = varsb[arg1[E]] ∪ varsb[arg2[E]] . But by Def. 1.11 e contained in E implies e is contained in arg1[F] or e is contained in arg2[E] . Then by the induction hypotheses,

$$e \in \text{varsb[arg1[E]]} \cup \text{varsb[arg2[E]]} .$$

The same argument holds for isdiff[E], isprod[E], and isquot[E] .

For iscond[E],

varsb[E] = varsb[arg1[prop[E]]] ∪
varsb[arg2[prop[E]]] ∪ varsb[ante[E]] ∪ varsb[consq[E]] .

Hence by the induction hypotheses,

$$e \in \text{varsb[E]} .$$

Thus by Def. 1.13, (a) is proven.

(b) is proven by induction upon π . Let π = NIL . Then

$$\text{varsa}[\pi, S] = S, \quad \text{varsa}[\pi, S \cup T] = S \cup T,$$

and

$$S \subseteq S \cup T .$$

Assume (b) is true for all π such that $\#[\pi] \leq n$. Let $\pi' = s' * \pi$.

varsa[π', S] = if isgoto[s'] then varsa[π, S ∪ varsb[arg1
[prop[s']]] ∪ varsb[arg2[prop[s']]]] else
if isassign[s'] then varsa[π, S ∪ {left[s']} ∪ varsb[right[s']]]
else varsa[π, S ∪ {argum[s']}]

varsa[π', S $\cup$ T] has an analogous expression, replacing each S by S $\cup$ T . Consideration of the cases, i.e. isgoto[s'], isassign[s'], isread[s'], and iswrite[s'], and applying the induction hypothesis gives varsa[π', S] $\subseteq$ varsa[π', S $\cup$ T] , completing the induction.

To prove (c), first assume (i). By Def. 1.18,

$$\begin{aligned} \text{varsa}[s, \{\}] &= \text{varsa}[\text{NIL}, \{\text{left}[s]\} \cup \text{varsb}[\text{right}[s]]] \\ &= \{\text{left}[s]\} \cup \text{varsb}[E] . \end{aligned}$$

Then varsb[E] $\subseteq$ varsa[s, {}] . Now assume (ii). Then

$$\text{varsa}[s, \{\}] = \{\} \cup \text{varsb}[\text{arg1}[\text{prop}[s]]] \cup \text{varsb}[\text{arg2}[\text{prop}[s]]] .$$

Then

$$E = \text{arg1}[\text{prop}[s]] \vee E = \text{arg2}[\text{prop}[s]]$$

implies

$$\text{varsb}[E] \subseteq \text{varsa}[s, \{\}] .$$

Thus (c) is true.

The proof of (d) will be an induction upon τ and τ' . Let $\tau = \tau' = \text{NIL}$. Then

$$\text{vars}[\tau * s * \tau'] = \text{varsa}[s, \{\}]$$

by Def. 1.18. Hence

$$\text{varsa}[s, \{\}] \subseteq \text{vars}[\tau * s * \tau'] .$$

Now assume (d) is true for $\tau = \text{NIL}$ and all τ' such that $\#[\tau'] \leq n$.

Let $\tau'' = s' * \tau'$. Then

$$\text{vars}[\tau * s * \tau''] = \text{varsa}[s * s' * \tau', \{\}]$$
$$= \text{varsa}[s' * \tau', \text{varsa}[s, \{\}]]$$
$$= \text{varsa}[\tau', \text{varsa}[s, \{\}] \cup \text{varsa}[s', \{\}]] .$$

By the induction hypothesis,

$$\text{varsa}[s, \{\}] \subseteq \text{varsa}[s * \tau', \{\}] .$$

But

$$\text{varsa}[s * \tau', \{\}] = \text{varsa}[\tau', \text{varsa}[s, \{\}]] .$$

Then

$$\text{varsa}[s, \{\}] \subseteq \text{varsa}[\tau', \text{varsa}[s, \{\}]] .$$

By (b)

$$\text{varsa}[\tau', \text{varsa}[s, \{\}]] \subseteq \text{varsa}[\tau', \text{varsa}[s, \{\}] \cup \text{varsa}[s', \{\}]] =$$
$$\text{vars}[\tau * s * \tau''] .$$

Hence

$$\text{varsa}[s, \{\}[\subseteq \text{vars}[\tau * s * \tau''] .$$

Now assume (d) is true for a fixed τ' and all τ such that $\#[\tau] \leq n$. Let $\tau'' = s' * \tau$. Then

$$\text{vars}[\tau'' * s * \tau'] = \text{varsa}[\tau * s * \tau', \text{varsa}[s', \{\}]] .$$

By the induction hypothesis

$$\text{varsa}[s, \{\}] \subseteq \text{varsa}[\tau * s * \tau', \{\}]$$

while by (b)

$$\text{varsa}[\tau * s * \tau', \{\}] \subseteq \text{varsa}[\tau * s * \tau', \text{varsa}[s', \{\}]] .$$

Hence

$$\text{varsa}[s, \{\}] \subseteq \text{vars}[\tau'' * s * \tau'] .$$

Then by induction (d) is true, Thus, by (a), (c), and (d),

$$e \in \text{varsb}[E] \subseteq \text{varsa}[s, \{\}] \subseteq \text{vars}[\tau * s * \tau'] .$$

Hence

$$e \in \text{vars}[\pi] .$$

To show the second part of the lemma, let $\pi = \tau * s * \tau'$. The proof will be by induction upon τ and τ' . Let $\tau = \tau' = \text{NIL}$. Then by Def. 1.18,

$$\text{vars}[\pi] = \text{varsa}[s, \{\}] = \{\text{argum}[s]\} .$$

Assume the result is true for all τ' with $\#[\tau'] \leq n$. Let $\tau'' = s' * \tau'$. Then

$$\text{vars}[\pi] = \text{varsa}[s * s' * \tau', \{\}] = \text{varsa}[s' * \tau', \text{varsa}[s, \{\}]]$$
$$= \text{varsa}[\tau', \text{varsa}[s, \{\}] \cup \text{varsa}[s', \{\}]] .$$

By the induction hypothesis, and (b) above

$$\text{argum}[s] \in \text{varsa}[\tau', \text{varsa}[s, \{\}]] \subseteq \text{vars}[\pi] .$$

Now assume the result is true for fixed τ' and all τ with $\#[\tau] \leq n$.

Let $\tau'' = s' * \tau$. Then

$$\text{vars}[\pi] = \text{varsa}[s' * \tau * s * \tau', \{\}]$$
$$= \text{varsa}[\tau * s * \tau', \text{varsa}[s', \{\}]] .$$

But by the induction hypothesis and (b),

$$\text{argum}[s] \in \text{varsa}[\tau * s * \tau', \{\}] \subseteq \text{vars}[\pi] .$$

Hence the second part of the lemma is true.

Lemma 1.10 $\xi_1 =_A \xi_2 \supset \text{avalue}[e, \xi_1] = \text{avalue}[e, \xi_2]$ where e is a proper arithmetic expression contained in a program π and $A = \text{vars}[\pi] \cup \{\text{out}, \text{ctr}\}$.

This will be a proof by induction upon the arithmetic expression e, using Def. 1.13.

Case 1. isvar[e] .

From Definitions 1.6, 1.8, and 1.12, and Lemma 1.9,

$$\text{avalue}[e, \xi_1] = c[e, \xi_1] = c[e, \xi_2] = \text{avalue}[e, \xi_2]$$

Case 2. isconst[e] .

By Def. 1.12,

$$\text{avalue}[e, \xi_1] = \text{val}[e] = \text{avalue}[e, \xi_2]$$

Now assume the appropriate induction hypotheses for the following five cases.

Case I. issum[e] .

$$\text{avalue}[e, \xi_1] = \text{avalue}[\text{argl}[e], \xi_1] + \text{avalue}[\text{arg2}[e], \xi_1]$$ by Def. 1.12

$$= \text{avalue}[\text{argl}[e], \xi_2] + \text{avalue}[\text{arg2}[e], \xi_2]$$ by inductive hypothesis

$$= \text{avalue}[e, \xi_2]$$ by Def. 1.12

Case II, III, IV. issdiff[e], isprod[e], isquot[e] .

Identical with Case I, replacing + by -, × and ÷ respectively.

Case V. iscond[e] .

Using Def. 1.12,

$$\text{avalue}[e, \xi_1] = \underline{\text{if}}\ \text{bvalue}[\text{prop}[e], \xi_1]\ \underline{\text{then}}\ \text{avalue}[\text{ante}[e], \xi_1]\ \underline{\text{else}}\ \text{avalue}[\text{consq}[e], \xi_1]$$

$$= \underline{\text{if}}\ \text{bvalue}[\text{prop}[e], \xi_2]\ \underline{\text{then}}\ \text{avalue}[\text{ante}[e], \xi_2]\ \underline{\text{else}}\ \text{avalue}[\text{consq}[e], \xi_2]$$

$$= \text{avalue}[e, \xi_2]$$

This depended upon $\text{bvalue}[\text{prop}[e], \xi_1]$ iff $\text{bvalue}[\text{prop}[e], \xi_2]$. But for this equivalence,

$$\text{LHS} = \underline{\text{if}}\ \text{isequal}[\text{prop}[e]]\ \underline{\text{then}}\ (\text{avalue}[\text{argl}[\text{prop}[e]], \xi_1] = \text{avalue}[\text{arg2}[\text{prop}[e]], \xi_1])\ \underline{\text{else if}}\ \text{isless}[\text{prop}[e]]\ \underline{\text{then}}\ (\text{avalue}[\text{argl}[\text{prop}[e]], \xi_1] < \text{avalue}[\text{arg2}[\text{prop}[e]], \xi_1])$$

$$= \underline{\text{if}}\ \text{isequal}[\text{prop}[e]]\ \underline{\text{then}}\ (\text{avalue}[\text{argl}[\text{prop}[e]], \xi_2] = \text{avalue}[\text{arg2}[\text{prop}[e]], \xi_2])\ \underline{\text{else if}}\ \text{isless}[\text{prop}[e]]\ \underline{\text{then}}\ (\text{avalue}[\text{argl}[\text{prop}[e]], \xi_2] < \text{avalue}[\text{arg2}[\text{prop}[e]], \xi_2])$$

$$= \text{RHS}$$

Hence by the induction principle, the lemma is true for all arithmetic expressions e .

Corollary 1.10.1 $\xi_1 \equiv_A \xi_2 \supset$ bvalue[e, ξ_1] iff bvalue[e, ξ_2] where e is a proper Boolean expression contained in a program π, and $A = vars[\pi] \cup \{out, ctr\}$.

Proof:

Case I. isequal[e] .

Then bvalue[e, ξ_1] = (avalue[argl[e], ξ_1] = avalue[arg2[e], ξ_1])
= (avalue[argl[e], ξ_2] = avalue[arg2[e], ξ_2])
= bvalue[e, ξ_2]

The first and last equalities come from Def. 1.12, while the middle one uses Lemma 1.10.

Case II. isless[e] .

Then bvalue[e, ξ_1] = (avalue[argl[e], ξ_1] < avalue[arg2[e], ξ_1])
= (avalue[argl[e], ξ_2] < avalue[arg2[e], ξ_2])
= bvalue[e, ξ_2]

The justification is identical with that of Case I.

Lemma 1.11 For all arithmetic expressions ae, all Boolean expressions be, and all statements s,

lengthe[ae] $\geq$ 1

lengthb[be] $\geq$ 1

lengths[s] $\geq$ 1 .

Proof. The proof of the first inequality will be by induction upon arithmetic expressions (Def. 1.13). Let e be an arithmetic expression. By Def. 1.17, isvar[e] implies lengthe[e] = 1 . Also isconst [e] implies lengthe[e] = 1 . Now assume the appropriate induction hypotheses.

Case I. issum[e] .

By Def. 1.17,

lengthe[e] = 2 + lengthe[arg1[e]] + lengthe[arg2[e]] $\geq$ 1

Case II, III, IV. isdiff[e], isprod[e], and isquot[e] .

Identical to Case I.

Case V. iscond[e] .

Then lengthe[e] = 2 + lengthe[ante[e]] + lengthe[consq[e]]
+ (if isequal[prop[e]] then 4 + lengthe[arg2[prop
[e]]] + lengthe[arg1[prop[e]]] else if isless[prop
[e]] then 6 + lengthe[arg2[prop[e]]] + lengthe[arg1[
prop[e]]]) $\geq$ 1

Hence, by induction lengthe[e] $\geq$ 1 .

For the second inequality, let e be a Boolean expression.

Then isequal[e] implies lengthb[e] = 4 + lengthe[arg1[e]] +
lengthe[arg2[e]] $\geq$ 1,

and isless[e] implies lengthb[e] = 6 + lengthe[arg1[e]] + lengthe[arg2[e]] $\geq$ 1 .

These inequalities used the previously proven first part of the lemma.

For the third inequality, letting s be a statement,

if isgoto[s] then lengths[s] = 3 + lengthb[prop[s]] $\geq$ 1,

if isassign[s] then lengths[s] = 1 + lengthe[right[s]] $\geq$ 1,

if isread[s] then lengths[s] = 4 $\geq$ 1,

if iswrite[s] then lengths[s] = 1 $\geq$ 1 .

Again, these inequalities used the previously proven sections of the lemma.

Lemma 1.12

Let e be a proper arithmetic expression contained in a program π . Then

Compile[π] = τ_1 * acom[e, i, k] * τ_2 with i $\geq$ 1, k $\geq$ 0 .

Proof: First the result, that if e is contained in an arithmetic expression E and i $\geq$ 1, k $\geq$ 0, then acom[E, i, k] = τ * acom[e, i',k'] *τ' with i' $\geq$ 1, k' $\geq$ 0, τ and τ' possibly null, will be proven by induction upon E .

Cases 1 and 2. isvar[E] and isconst[E] .

Then e = E and acom[E, i, k] = acom[e, i, k] by Def. 1.11, and 1.17, and Axiom 1.11.1. Assume the appropriate induction hypotheses.

Case I. issum[E] .

Then by Def. 1.11, there exists E' such that e is contained in E' and E' = argl[E] or E' = arg2[E] . By Def. 1.17

$$\text{acom}[E, i, k] = \text{acom}[\text{arg2}[E], i, k] * \text{mksto}[t[k]] * \text{acom}[\text{argl}[E], i + \text{lengthe}[\text{arg2}[E]] + 1, k+1] * T$$

where T is defined in Def. 1.17. But the result is true by the induction hypotheses for argl[E] and arg2[E] . Hence it is true for E .

Cases II, III, IV. isdiff[E], isprod[E], isquot[E] .

The proof is identical with Case I, except for the term T, which is immaterial in the proof of Case I.

Case V. iscond[E] .

By Def. 1.17 and 1.11,

$$\begin{aligned}\text{acom}[E, i, k] = \underline{\text{if}}\ \text{isequal}[E]\ \underline{\text{then}}\ & \text{acom}[\text{arg2}[\text{prop}[E]], i, k] * \text{mksto}[t[k]] * \text{acom}[\text{argl}[\text{prop}[E]], I_1, K_1] * \text{mksub}[t[k]] \\ & * \text{mktze}[T_1] * \text{mkli}[1] * \text{mktze}[T_2] * \text{acom}[\text{consq}[E], i + I_2, k] \\ & * \text{mktra}[T_3] * \text{acom}[\text{ante}[E], i + I_3, k]\ \underline{\text{else if}}\ \text{isless}[E] \\ & \underline{\text{then}}\ \text{acom}[\text{arg2}[\text{prop}[E]], i, k] * \text{mksto}[t[k]] * \text{acom}[\text{argl}[\text{prop}[E]], I_1, K_1] * \text{mksub}[t[k]] * \text{mktmi}[T_4] * \text{mkli}[1] * \text{mktra}[T_5] \\ & * \text{mkli}[0] * \text{mktze}[T_2] * \text{acom}[\text{consq}[E], i + I_2, k] * \text{mktra}[T_3] \\ & * \text{acom}[\text{ante}[E], i + I_3, k]\end{aligned}$$

where I_1, K_1, T_1, T_2, I_2, T_3, I_3, T_4 and T_5 are obtained from Def. 1.17. But by the induction hypotheses, the result is true for arg2[prop[E]], argl[prop[E]], ante[E]], and consq[E] . Hence the result is true for this case, and hence for all E by Def. 1.13.

By Def. 1.11, there exists a statement s and sequences τ and τ' such that $\pi = \tau * s * \tau'$. The proof of the lemma is by induction upon τ and τ' . Define $\ell = \text{table}[\pi]$. Let $\pi = s$, i.e. $\tau = \tau' = \text{NIL}$. Then $\text{compile}[\pi] = \text{coml}[s, 1, \ell] * \text{mkhalt}$.

Case 1. $\text{isassign}[s]$.

Then by Def. 1.11 and 1.17,

$$\text{compile}[\pi] = \text{com2}[s, 1, \ell] * \text{mkhalt} = \text{acom}[E, 1, 0] * \text{mksto}[\text{left}[s]] * T$$

where T is obtainable from Def. 1.17.

Case 2. $\text{isgoto}[s]$.

$$\text{compile}[\pi] = \text{acom}[\text{arg2}[\text{prop}[s]], 1, 0] * \text{mksto}[t[0]] * \text{acom}[\text{arg1}[\text{prop}[s]], 1 + \text{lengthe}[\text{arg2}[\text{prop}[s]]], 1] * T$$

where T is obtainable from Def. 1.17, while by Def. 1.11,

$$E = \text{arg2}[\text{prop}[s]] \vee E = \text{arg1}[\text{prop}[s]] .$$

Hence the lemma is true for $\tau = \tau' = \text{NIL}$.

Assume the lemma is true for $\tau = \text{NIL}$ and all τ' such that $\#[\tau'] \leq n$. Let $\tau'' = s' * \tau'$. Then

$$\text{compile}[\pi] = \text{com2}[s, 1, \ell] * \text{mksyntra}[2 + \text{lengths}[s]] * \text{coml}[\tau'', N, \ell] * \text{mkhalt} .$$

But then the proof of the above two cases suffices since the additional terms only effect τ_2 .

Assume the lemma is true for fixed τ' and all τ such that $\#[\tau] \leq n$. Let $\tau'' = s' * \tau$. Then

$$\text{compile}[\pi] = \text{com2}[s', 1, \ell] * \text{mksyntra}[2 + \text{lengths}[s']] * \text{coml}[\tau * s * \tau', 2 + \text{lengths}[s'], \ell] * \text{mkhalt}.$$

But by the induction hypotheses, the lemma is true for $\text{coml}[\tau * s * \tau', n, \ell] * \text{mkhalt}$. Hence it is true for $\pi = s' * \tau * s * \tau'$. Then by induction it is true for all π.

Corollary 1.12.1

Let e be a proper Boolean expression contained in a program π. Then

$$\text{compile}[\pi] = \tau_1 * \text{bcom}[e, i, k] * \tau_2 \quad \text{with} \quad i \geq 1, \; k \geq 0.$$

Proof: By Def. 1.11, there are two cases.

Case I. $\text{isgoto}[s] \wedge e = \text{prop}[s]$.

But this is proven in Case 2 of the proof in Lemma 1.11.

Case II. $(\exists E)(\text{isae}[E] \wedge \text{iscond}[F] \wedge e = \text{prop}[E] \wedge E$ is a proper arithmetic expression contained in a program $\pi)$.

This is a direct result of Lemma 1.11 and Def. 1.17.

SECTION III

PROOF OF FUNDAMENTAL RESULT

In this section, the central correctness theorem will be stated and proven. In addition some obvious extensions of the basic result are noted. Throughout this section, the specific source language, target language, and compiler are respectively Mickey, machine, and Compile unless otherwise noted. The correctness theorem can now be stated from Def. 1.10 and the identification of the functions mentioned.

Theorem 2.1 The compiler Compile[π] is correct.

The theorem as proven will have an alternate statement based upon Def. 1.10.

Theorem 2.1 (Alternate).

$$(\forall\pi)(\forall\theta)(\forall\xi)(\forall\eta)((\xi \text{ is a p.s.v. of } \pi \text{ in Mickey} \wedge$$
$$\eta \text{ is a p.s.v. of Compile}[\pi] \text{ in machine} \wedge$$
$$c[IC, \eta] = c[sn, \xi] = 1 \wedge$$
$$A = vars[\pi] \cup \{out, ctr\} \wedge$$
$$\xi =_A \eta) \supset$$
$$Mickey[\pi, \xi, \theta] \cong_A machine[Compile[\pi], \eta, \theta]).$$

The proof will require three auxiliary theorems, and several lemmas. The theorems have slightly weaker hypotheses than Theorem 2.1. After the lemmas and theorems have been proven, we will return to the proof of Theorem 2.1.

In the following, we shall have occasion to compose the functions rest and restprog with themselves many times. Accordingly we will use subscripts to denote the number of functions involved in the composition. That is $rest_1[\pi] = rest[\pi]$, $rest_2[\pi] = rest[rest[\pi]]$, ..., $rest_n[\pi]$ = the composition of the function rest n times. By convention, $rest_0[\pi] = \pi$. The same system of denotation will be used for composition of the function restprog .

Lemma 2.2 $\neg null[\ell] \supset first[\ell * m] = first[\ell]$.

Assume $\neg null[\pi] \wedge null[rest[\pi]]$.

Then, by Axiom 1.14.1

$$first[\pi * \tau_1] = \pi = first[\pi] * rest[\pi]$$
$$= first[\pi] * NIL = first[\pi]$$

Likewise

$$first[\pi * (\tau_1 * \tau_2)] = first[\pi] = first[\pi * \tau_1]$$

Since * is associative,

$$\pi * (\tau_1 * \tau_2) = (\pi * \tau_1) * \tau_2$$

Hence

$$first[(\pi * \tau_1) * \tau_2] = first[\pi * \tau_1] .$$

The lemma follows by identifying π as $first[\ell]$, τ_1 as $rest[\ell]$, and τ_2 as m .

Corollary 2.2.1 For all π, tlu[1, table[π]] = 1 .

Proof:

By Def. 1.17,

$$\begin{aligned} &\text{tlu[1, table[}\pi\text{]]} = \text{first[[tabl[1, }\pi\text{]]]} \\ &= \underline{\text{if}}\ \text{null[}\pi\text{]}\ \underline{\text{then}}\ \text{first[1]}\ \underline{\text{else}} \\ &\quad \underline{\text{if}}\ \text{null[restprog[}\pi\text{]]}\ \underline{\text{then}}\ \text{first[1 * 1 + lengths[firstate[}\pi\text{]] +1]} \\ &\quad \underline{\text{else if}}\ \text{null[restprog}_2\text{[}\pi\text{]]}\ \underline{\text{then}}\ \text{first[1 * 1 + lengths[firstate[}\pi\text{]] + 1 * 1 + lengths[firstate[}\pi\text{]] + 1 + lengths[firstate[restprog[}\pi\text{]]] + 1 + 1]}\ \underline{\text{else}}\ \ldots\ \text{first[1 * 1 + } \ldots\ \text{]} \end{aligned}$$

Since $\neg$ null[1], by Lemma 2.1

$$\text{first[1 * } \tau\text{]} = 1 .$$

Then tlu[1, table[π]] = 1 .

Lemma 2.3 For all $i \geq 0$, $\neg$ null[first[rest$_i$[π]]] $\supset$ rest$_{i+1}$[π * τ] = rest$_{i+1}$[π] * τ .

From 1.14.1,

$$\begin{aligned} \pi * \tau &= \text{first[}\pi * \tau\text{]} * \text{rest[}\pi * \tau\text{]} \\ &= \text{first[}\pi\text{]} * \text{rest[}\pi * \tau\text{]} \end{aligned}$$

from Lemma 2.2. But, by Axiom 1.14.1,

$$\pi * \tau = \text{first[}\pi\text{]} * \text{rest[}\pi\text{]} * \tau .$$

Hence

$$\neg\ \text{null[first[}\pi\text{]]} \supset \text{rest[}\pi * \tau\text{]} = \text{rest[}\pi\text{]} * \tau$$

i.e., the lemma is true for $i = 0$.

Assume the lemma is true for all $i \leq n$.

Assume $\neg$ null[first[$rest_{i+1}[\pi]$]] .

$$rest_{i+2}[\pi * \tau] = rest[rest_{i+1}[\pi * \tau]]$$

Since $\neg$ null[first[$rest_{i+1}[\pi]$]] implies $\neg$ null[first[$rest_{i}[\pi]$]], then by the induction hypothesis,

$$rest_{i+1}[\pi * \tau] = rest_{i+1}[\pi] * \tau .$$

Then $rest_{i+2}[\pi * \tau] = rest[rest_{i+1}[\pi] * \tau]$. Applying the same proof as used for $i = 0$,

$$rest[rest_{i+1}[\pi] * \tau] = rest[rest_{i+1}[\pi]] * \tau$$

Hence

$$rest_{i+2}[\pi * \tau] = rest_{i+2}[\pi] * \tau .$$

Thus the lemma is proven by induction.

We can now give a general formula for the function presentinst applied to the concatenation of two programs.

Lemma 2.4 For $n > 0$,

presentinst[$\tau * \pi$, n] = **if** $n \leq$ #inst[τ] **then** presentinst[τ, n] **else** presentinst[π, n - #inst[τ]] .

Proof: The proof will be by cases.

Let p = #inst[τ] .

Case 1. $n \leq \#inst[\tau] = p$.

By Def. 1.19,

$$null[rest_p[\tau]] \text{ and } \neg null[rest_{p-1}[\tau]] .$$

By Def. 1.16,

$$LHS = first[rest_{n-1}[\tau * \pi]]$$

$$RHS = first[rest_{n-1}[\tau]]$$

But from Lemma 2.3, since $n \leq p$ and $\neg null[rest_{p-1}[\tau]]$

$$rest_{n-1}[\tau * \pi] = rest_{n-1}[\tau] * \pi$$

Hence, using Lemma 2.2,

$$LHS = RHS$$

Case 2. $n > p$.

By Def. 1.16 and 1.14.1

$$\begin{aligned} LHS &= presentinst[rest_{p-1}[\tau * \pi], n - p + 1] \\ &= presentinst[rest_{p-1}[\tau] * \pi, n - p + 1] \qquad \text{by Lemma 2.3,} \\ &= presentinst[rest[rest_{p-1}[\tau] * \pi], n - p] \\ &= presentinst[rest_p[\tau] * \pi, n - p] \\ &= presentinst[NIL * \pi, n - p] = presentinst[\pi, n - p] = RHS \end{aligned}$$

This completes the proof.

The next two lemmas prove the basic facts which are needed to discuss the function tlu applied to the list table[π] .

Lemma 2.5

Let $c[sn, \xi] = n \leq \#[\pi]$. Let $state[n, \pi] = s$. Let $tlu[n, table[\pi]] = i$.

Then

$$i + 1 + lengths[s] = tlu[n + 1, table[\pi]] .$$

Proof: From Def. 1.17,

$$tlu[n + 1, table[\pi]] = first[rest_n[table[\pi]]]$$

and

$$tlu[n, table[\pi]] = first[rest_{n-1}[table[\pi]]]$$

These are based upon the facts, to be shown shortly, that

$$\neg null[rest_n[table[\pi]]] \text{ and } \neg null[rest_{n-1}[table[\pi]]] .$$

By Axiom 1.14 1,

$$rest_{n-1}[table[\pi]] = first[rest_{n-1}[table[\pi]]] * rest_n[table[\pi]]$$

By hypothesis,

$$tlu[n, table[\pi]] = i \text{ where } \neg null[i] .$$

Then by Def. 1.17 and 1.14.1, and substituting into the above equation,

$$rest_{n-1}[table[\pi]] = i * tabl[i + lengths[firstate[restprog_{n-1}[\pi]]] + 1, restprog_n[\pi]]$$

Then, using Lemma 2.2,

$$tlu[n + 1, table[\pi]] = i + lengths[firstate[restprog_{n-1}[\pi]]] + 1$$

It remains to show $\neg null[rest_n[table[\pi]]]$ and $\neg null[rest_{n-1}[table[\pi]]]$. But the first implies the second, by 1.11.1 and 1.14.1, so only the first will be shown. Assume $null[rest_n[table[\pi]]]$. Then

$$rest_{n-1}[table[\pi]] = N_1 \text{ or } NIL$$

$$rest_{n-2}[table[\pi]] = N_2 * N_1 \text{ or } N_1 \text{ or } NIL$$

$$\vdots$$

$$rest[table[\pi]] = N_{n-1} * N_{n-2} * \ldots N_2 * N_1 \text{ or}$$

$$N_{n-2} * \ldots * N_2 * N_1 \text{ or } \ldots N_1 \text{ or } NIL$$

$$rest_0[table[\pi]] = table[\pi] = N_n * N_{n-1} * \ldots * N_1 \text{ or}$$

$$N_{n-1} * \ldots * N_1 \text{ or } \ldots \text{ or } N_1 \text{ or } NIL$$

where $N_1, N_2, \ldots, N_{n-1}$ are integers.

Also from Def. 1.17,

$$restprog_{n-1}[\pi] = NIL$$

$$restprog_{n-2}[\pi] = s_1 \text{ or } NIL$$

$$\vdots$$

$$restprog_0[\pi] = \pi = s_{n-1} * \ldots * s_1 \text{ or } s_{n-2} * \ldots * s_1 \text{ or } \ldots \text{ or } NIL$$

Then from Def. 1.19, by direct computation,

$$\#[\pi] = numb[\pi, 0] = n\text{-}1 \text{ or } n\text{-}2 \text{ or } \ldots 0 .$$

Hence $n > \#[\pi]$, a contradiction. Hence $\neg null[rest_n[table[\pi]]]$. This completes the proof.

Lemma 2.6

$$\#[\pi] + 1 \geq i_1 > i_2 \geq 1 \supset tlu[i_1, table[\pi]] > tlu[i_2, table[\pi]]$$

and

$$i \geq \#[\pi] + 1 \supset tlu[i, table[\pi]] = last[table[\pi]] .$$

Proof: For the first statement it is sufficient to show, for all $1 \leq i \leq \#[\pi]$,

$$tlu[i+1, table[\pi]] > tlu[i, table[\pi]] .$$

From Lemma 1.11 and Lemma 2.5,

$$tlu[i+1, table[\pi]] > tlu[i, table[\pi]],$$

i.e., the first part of the lemma is correct.

For the second section of the lemma, by direct calculation using Def. 1.19, and 1.14.1, $last[\pi] = rest_i[\pi]$ for the smallest i for which $null[rest_{i+1}[\pi]]$, and $i \geq 0$.

Case 1. Let $i = \#[\pi] + 1$.

Then

$$tlu[i, table[\pi]] = first[rest_{i-1}[table[\pi]]] .$$

From above

$$\neg\, null[rest_{i-1}[table[\pi]]] .$$

Assume

$$\neg\, \text{null}[\text{rest}_i[\text{table}[\pi]]] \ .$$

Then, using the method of Lemma 2.5

$$\text{rest}_{i-1}[\text{table}[\pi]] = N_1 * N_0 \quad \text{where} \quad N_0 = \text{rest}_i[\text{table}[\pi]]$$
$$\vdots$$
$$\text{rest}_0[\text{table}[\pi]] = \text{table}[\pi] = N_i * N_1 * \ldots N_1 * N_0$$

and

$$\text{restprog}_{i-1}[\pi] = \tau \qquad \text{where } \tau \text{ is a non-null sequence of statements.}$$
$$\vdots$$
$$\text{restprog}_0[\pi] = \pi = s_{i-1} * s_{1-2} * \ldots * s_1 * \tau$$

Then $\#[\pi] > i-1$, a contradiction. Hence $\text{null}[\text{rest}_i[\text{table}[\pi]]]$. Then $\text{first}[\text{rest}_{i-1}[\text{table}[\pi]]] = \text{rest}_{i-1}[\text{table}[\pi]]$ by Axiom 1.14.1. But then $\text{first}[\text{rest}_{i-1}[\text{table}[\pi]]] = \text{last}[\text{table}[\pi]]$ or $\text{tlu}[i, \text{table}[\pi]] = \text{last}[\text{table}[\pi]]$.

Case 2. Let $i > \#[\pi] + 1$. Let $j = \#[\pi] + 1$. From above,

$$\text{null}[\text{rest}_j[\text{table}[\pi]]] \ .$$

Then from Def. 1.17,

$$\text{tlu}[i, \text{table}[\pi]] = \text{tlu}[i-1, \text{rest}[\text{table}[\pi]]] = \ldots =$$
$$\text{tlu}[i - (j-1), \text{rest}_{j-1}[\text{table}[\pi]]] = \text{tlu}[i-j, \text{rest}_{j-1}[\text{table}[\pi]]] = \ldots =$$
$$\text{tlu}[1, \text{rest}_{j-1}[\text{table}[\pi]]] = \text{first}[\text{rest}_{j-1}[\text{table}[\pi]]] =$$
$$\text{tlu}[j, \text{table}[\pi]] = \text{last}[\text{table}[\pi]] \ .$$

Lemma 2.7 For all positive integers n and p and programs π,

$\neg$ ishalt[presentinst[coml[π, p, table[π]], n]] .

The proof will be by induction upon π .

Let π = NIL . Then by Def. 1.17

com[π, p, table[π]] = NIL.

By Def. 1.16

presentinst[NIL, n] = if n = 1 then first[NIL] else
presentinst[rest[NIL], n-1]
= if n = 1 then NIL else
presentinst[NIL, n-1] .

Hence for all $n \geq 1$, presentinst[NIL, n] = NIL . We have null[NIL] . By Axiom 1.14.1, then $\neg$ ishalt[NIL] . Hence the lemma is true for π = NIL . Assume the lemma is true for all π such that #[π] $\leq$ n .

Let #[τ] = n+1, i.e., τ = s * π .

coml[τ, p, table[τ]] = com2[s, p, table[τ]] * mksyntra
[p + 1 + lengths[s]] * coml[restprog[τ], p + 1 + lengths[s],
table[τ]]

By two applications of Lemma 2.4,

presentinst[coml[τ, p, table[τ]], n] = if n $\leq$ #inst[com2[s,
p, table[τ]]] then presentinst[com2[s, p, table[τ]], n]
else if n-#inst[com2[s, p, table[τ]]] $\leq$ 1 then
mksyntra[p + 1 + lengths[s]] else presentinst[coml[π,
p + 1 + lengths[s], table[τ]], n-#inst[com2[s, p, table[τ]]] -1]

Then

ishalt[presentinst[coml[τ, p, table[τ]], n]] iff

ishalt[presentinst[com2[s, p, table[τ]], n]]

∨ ishalt[mksyntra[p + 1 + lengths[s]]]

∨ ishalt[presentinst[coml[π, p + 1 + lengths[s],

table[τ]], n-#inst[com2[s, p, table[τ]]] -1]]

By the induction hypothesis and 1.15.14,

ishalt[presentinst[coml[τ, p, table[τ]], n]] iff

ishalt[presentinst[com2[s, p, table[τ]], n]]

We shall prove ¬ ishalt[presentinst[com2[s, p, table[τ]], n]] which will complete the induction, and prove the lemma. There are four cases, depending on s . First, however, we need a result to the effect that for all positive integers m, n, and p, and all arithmetic expressions e, then ¬ ishalt[presentinst[acom[e, p, m], n]] . The proof will be by induction upon e (Def. 1.13).

Case 1. isconst[e] .

Then

acom[e, p, m] = mkli[e] .

Hence, by Def. 1.16, and 1.15.1

presentinst[acom[e, p, m], n] = <u>if</u> n = 1 <u>then</u> first[mkli[e]]

<u>else</u> presentinst[rest[mkli[e]], n-1]

= if n = 1 then mkli[e] else presentinst[NIL, n-1]

= if n = 1 then mkli[e] else NIL .

Then

ishalt[presentinst[acom[e, p, m], n]] = ishalt[mkli[e]] ∨ ishalt[NIL]

Hence

¬ ishalt[presentinst[acom[e, p, m], n]] .

Case 2. isvar[e] .

Then by Def. 1.17, acom[e, p, m] = mkload[e] . Then, as in Case 1 using 1.15.2 and Def. 1.16,

presentinst[acom[e, p, m], n] = if n = 1 then mkload[e] else NIL

Then ¬ ishalt[presentinst[acom[e, p, m], n]] . Assume the result is true for argl[e], arg2[e], argl[prop[e]], arg2[prop[e]], ante[e], and consq[e] as appropriate for induction hypotheses.

Case I. issum[e] .

acom[e, p, m] = acom[arg2[e], p, m] * mksto[t[m]] * acom[argl[e],
p + lengthe[arg2[e]] + 1, m+1] * mkadd[t[m]]

By Lemma 2.4,

ishalt[presentinst[acom[e, p, m], n]] iff
ishalt[presentinst[acom[arg2[e], p, m], n]]
∨ ishalt[mksto[t[m]]]

∨ ishalt[presentinst[acom[arg1[e], p + lengthe[arg2[e]] +1,

m+1], n-#inst[acom[arg2[e], p, m]] -1]]

∨ ishalt[mkadd[t[m]]] .

By the induction hypothesis and 1.15.7 and 1.15.3

¬ ishalt[presentinst[acom[e, p, m], n]] .

Case II - IV. isdiff[e], isprod[e], isquot[e] .

The proof is the same as Case I, except substituting mksub[t[m]], mkmpy[t[m]], and mkdiv[t[m]] and 1.15.4, 1.15.5, and 1.15.6 for mkadd[t[m]] and 1.15.3 respectively.

Case V. iscond[e] .

There are two subcases. First, Subcase Va - isequal[e] .

Then, as above, using Def. 1.16 and Lemma 2.4

ishalt[presentinst[acom[e, p, m], n]] iff

ishalt[presentinst[acom[arg2[prop[e]], p, m], n]]

∨ ishalt[mksto[t[m]]]

∨ ishalt[presentinst[acom[arg1[prop[e]], p + 1 + lengthe[

arg2[prop[e]]], m+1], N_1]] ∨ ishalt[mksub[t[m]]]

∨ ishalt[mktze[p + lengthe[arg2[prop[e]]]

+ lengthe[arg1[prop[e]]] +4]]

∨ ishalt[mkli[1]]

where N_1 = n-1-#inst[acom[arg2[prop[e]], p, m]] .

By the induction hypothesis, 1.15.1, 1.15.4, 1.15.7, and 1.15.10

$\neg$ ishalt[presentinst[acom[e, p, m], n]] .

Subcase Vb - isless[e] .

Again, as above,

ishalt[presentinst[acom[e, p, m], n]] iff

ishalt[presentinst[acom[arg2[prop[e]], p, m], n]]

$\vee$ ishalt[mksto[t[m]]]

$\vee$ ishalt[presentinst[acom[arg1[prop[e]], P_1, m+1], N_1]]

$\vee$ ishalt[mksub[t[m]]] $\vee$ ishalt[mktmi[J_1]]

$\vee$ ishalt[mkli[1]] $\vee$ ishalt[mktra[J_2]]

$\vee$ ishalt[mkli[0]]

where N_1 is as above, and P_1, J_1, and J_2 are expressions obtainable from Def. 1.16, but not relevant to the predicate ishalt .

By induction hypothesis, 1.15.1, 1.15.4, 1.15.7, 1.15.8, and 1.15.9

$\neg$ ishalt[presentinst[acom[e, p, m], n]] .

Hence the required result is true for all e by Def. 1.13. Now to return to the main lemma, i.e., the proof $\neg$ ishalt[presentinst[com2[s, p, table[τ]], n]] .

Case 1. isgoto[s] .

By Def. 1.16,

com2[s, p, table[τ]] = bcom[prop[s], p, 0]

* mktze[p + lengthb[prop[s]] +2]

* mktra[p + lengthb[prop[s]] +3]

* mksyntra[tlu[dest[s], table[τ]]] .

Then, as above

ishalt[presentinst[com2[s, p, table[τ]], n]] iff

ishalt[presentinst[bcom[prop[s], p, 0], n]]

∨ ishalt[mktze[p + lengthb[prop[s]] +2]]

∨ ishalt[mktra[p + lengthb[prop[s]] +3]]

∨ ishalt[mksyntra[tlu[dest[s], table[τ]]]] .

Examination of the proof of subcases Va and Vb of the intermediate result shows that we have also proven ¬ ishalt[presentinst[bcom[e, p, m], n]] . Hence, using this together with 1.15.8, 1.15.10, and 1.15.14,

¬ ishalt[presentinst[com2[s, p, table[τ]], n]] .

Case 2. isassign[s] .

As above,

LHS iff ishalt[presentinst[acom[right[s], p, 0], n]]

∨ ishalt[mksto[left[s]]] .

By the intermediate result and 1.15.7,

¬ ishalt[presentinst[com2[s, p, table[τ]], n]] .

Case 3. isread[s] .

Then

LHS iff ishalt[mkrd[arg[s]]] ∨ ishalt[mkli[l]]

∨ ishalt[mkadd[ctr]] ∨ ishalt[mksto[ctr]] .

Then by 1.15.1, 1.15.3, 1.15.7, and 1.15.11

¬ ishalt[presentinst[com2[s, p, table[τ]], n]] .

Case 4. iswrite[s] .

Then

LHS iff ishalt[mkwrt[arg[s]]] .

Then by 1.15.12,

¬ ishalt[presentinst[com2[s, p, table[τ]], n]] .

This completes the proof since the four cases are exhaustive.

Lemma 2.8 For $m \geq 0$, $n \geq \ell \geq 1$,

if (∀i)(presentinst[τ * π, n] ≠ presentinst[τ * π, i])

and

presentinst[τ * π, n] = presentinst[π, ℓ]

then

presentinst[τ * π, n+m] = presentinst[π, ℓ+m] .

This lemma can be paraphrased as saying that a unique element of the list π permits aligning the tails of the lists τ * π and π correctly.

Proof: Since Lemma 2.4 provides a general formula for presentinst[$\tau * \pi$, n] for all n, and presentinst[π, ℓ] is unique by hypothesis, then n-#inst[τ] = ℓ in Lemma 2.4. Hence n > #inst[τ] . Then n + m > #inst[τ] .

Then, applying Lemma 2.4 to the LHS,

$$\text{presentinst}[\tau * \pi, n+m] = \text{presentinst}[\pi, n+m - \#\text{inst}[\tau]]$$
$$= \text{presentinst}[\pi, \ell+m] .$$

Lemma 2.9

(1) $\text{rest}_{\text{lengthe}[ae]}[\text{acom}[ae, n, m] * \tau] = \tau$

(2) $\text{rest}_{\text{lengthb}[be]}[\text{bcom}[be, n, m] * \tau] = \tau$

(3) $\text{rest}_{\text{lengths}[s]}[\text{com2}[s, i, \text{table}[\pi]] * \tau] = \tau$

for arithmetic expressions ae, Boolean expressions be, and statements s . The proof of (1) will be by induction using Def. 1.13.

Let e be an arithmetic expression.

Case 1. isvar[e] .

By Def. 1.17

$$\text{lengthe}[e] = 1 \quad \text{and} \quad \text{acom}[e, n, m] = \text{mkload}[e] .$$

Then

$$\text{rest}_1[\text{mkload}[e] * \tau] = \text{rest}[\text{mkload}[e] * \tau] = \tau .$$

Case 2. isconst[e] .

Similar to Case 1, except 1.15.1 is used in place of 1.15.2. Assume the appropriate induction hypotheses.

Case I. issum[e] .

$$\text{lengthe}[e] = 2 + \text{lengthe}[\text{arg1}[e]] + \text{lengthe}[\text{arg2}[e]]$$

$$\text{acom}[e, n, m] = \text{acom}[\text{arg2}[e], n, m] * \text{mksto}[t[m]] * \text{acom}[\text{arg1}[e], n + \text{lengthe}[\text{arg2}[e]] + 1, m+1] * \text{mkadd}[t[m]]$$

Then

$$\text{rest}_{\text{lengthe}[e]}[\text{acom}[e, n, m] * \tau] =$$

$$\text{rest}[\text{rest}_{\text{lengthe}[\text{arg1}[e]]}[\text{rest}[\text{rest}_{\text{lengthe}[\text{arg2}[e]]}[\text{acom}[\text{arg2}[e], n, m] * \text{mksto}[t[m]] * \text{acom}[\text{arg1}[e], n + \text{lengthe}[\text{arg2}[e]] + 1, m+1] * \text{mkadd}[t[m]] * \tau]]]]$$

$$= \text{rest}[\text{rest}_{\text{lengthe}[\text{arg1}[e]]}[\text{rest}[\text{mksto}[t[m]] * \text{acom}[\text{arg1}[e], n + \text{lengthe}[\text{arg2}[e]] + 1, m+1] * \text{mkadd}[t[m]] * \tau]]]$$

$$= \text{rest}[\text{rest}_{\text{lengthe}[\text{arg1}[e]]}[\text{acom}[\text{arg1}[e], n + \text{lengthe}[\text{arg2}[e]] + 1, m+1] * \text{mkadd}[t[m]] * \tau]]$$

$$= \text{rest}[\text{mkadd}[t[m]] * \tau] = \tau \ .$$

Case II, III, IV. isdiff[e], isprod[e], isquot[e] .

Identical to Case I, except 1.15.4, 1.15.5, and 1.15.6 are used in place of 1.15.3 in reducing the last equality.

Subcase Va. isequal[e] .

lengthe[e] = 6 + lengthe[arg2[prop[e]]] + lengthe[arg1[prop[e]]] +
lengthe[ante[e]] + lengthe[consq[e]]

acom[e, n, m] = acom[arg2[prop[e]], n, m] * mksto[t[m]]
* acom[arg1[prop[e]], n + 1 + lengthe[arg2[prop[e]]], m+1]
* mksub[t[m]] * mktze[T_1] * mkli[1] * mktze[T_2] *
acom[consq[e], n + lengthb[e] +1, m] * mktra[T_3] *
acom[ante[e], T_4, m]

where T_1, T_2, T_3, and T_4 are terms directly obtainable from Def. 1.16. But then, by application of the induction hypotheses and 1.15.1, 1.15.4, 1.15.7, 1.15.8, and 1.15.10,

$$rest_{lengthe[e]}[acom[e, n, m] * \tau] = \tau .$$

Subcase Vb. isless[e] .

Similar to Subcase Va. Hence (1) is true by Def. 1.13.

(2) is proven exactly as Subcases Va and Vb, except that (1) is used, instead of induction hypotheses.

The proof of (3) is by cases.

Case 1. isgoto[s] .

$$rest_{lengths[s]}[com2[s, i, table[\pi]] * \tau] =$$
$$rest[rest[rest[rest_{lengthb[prop[s]]}[bcom[prop[s], i, 0] * mktze$$

$$[i + lengths[s] - 1] * mktra[i + lengths[s]] * mksyntra[$$

$$tlu[dest[s], table[\pi]]] * \tau]]]]$$

$$= rest_3[mktze[i + lengths[s] - 1] * mktra[i + lengths[s]]$$

$$* mksyntra[tlu[dest[s], table[\pi]]] * \tau] = \tau \ .$$

Case 2. $isassign[s]$.

$$lengths[s] = 1 + lengthe[right[s]] \ .$$

$$LHS = rest[rest_{lengthe[right[s]]}[acom[right[s], i, 0] * mksto[$$

$$s]] * \tau]] = rest[mksto[left[s]] * \tau] = \tau \ .$$

Case 3. $isread[s]$.

$$LHS = rest_4[mkrd[arg[s]] * mkli[1] * mkadd[ctr] * mksto[ctr] * \tau] = \tau \ .$$

Case 4. $iswrite[s]$.

$$LHS = rest[mkwrt[arg[s]] * \tau] = \tau \ .$$

This completes the proof.

The next two lemmas and their corollaries show that the functions Compile, presentinst, and table do operate together in fact as one would expect.

<u>Lemma 2.10</u> Let $c[sn, \xi] = n$, $\#[\pi] = p$, $c[IC, \eta] = i$, $1 \leq n \leq p$, and $i = tlu[n, table[\pi]]$. Then

$$\text{rest}_{i-1}[\text{Compile}[\pi]] = \text{com2}[\text{state}[n, \pi], i, \text{table}[\pi]] * \text{mksyntra}[i + 1 + \text{lengths}[\text{state}[n, \pi]]] * \text{coml}[\text{restprog}_n[\pi], i + 1 + \text{lengths}[\text{state}[n, \pi]], \text{table}[\pi]] * \text{mkhalt} .$$

Proof: The proof is by induction upon n for an arbitrary fixed π. Denote $\text{state}[n, \pi]$ by $s[n]$ and $i + 1 + \text{lengths}[s[n]]$ by $j[n]$. Corollary 2.2.1 proves the lemma for $n = 1$.

Now assume the lemma is true for all $n \leq k$. If $k = p$ the lemma is true trivially. Assume $k < p$.

By Lemma 2.5,

$$\text{tlu}[k+1, \text{table}[\pi]] = j[k]$$

Then

$$\text{rest}_{j[k]-1}[\text{Compile}[\pi]] = \text{rest}_{j[k]-1}[\text{rest}_{i-1}[\text{compile}[\pi]]] .$$

By the induction hypothesis,

$$\text{rest}_{j[k]-1}[\text{Compile}[\pi]] = \text{rest}_{j[k]-i}[\text{com2}[s[k], i, \text{table}[\pi]] * \text{mksyntra}[j[k]] * \text{coml}[\text{restprog}_k[\pi], j[k], \text{table}[\pi]] * \text{mkhalt}] .$$

Then by Lemma 2.9,

$$\text{LHS} = \text{rest}[\text{rest}_{\text{lengths}[s[k]]}[\text{com2}[s[k], i, \text{table}[\pi]] * \text{mksyntra}[j[k]] * \text{coml}[\text{restprog}_k[\pi], j[k], \text{table}[\pi]] * \text{mkhalt}]]$$

$$= \text{rest}[\text{mksyntra}[j[k]] * \text{coml}[\text{restprog}_k[\pi], j[k], \text{table}[\pi]] * \text{mkhalt}]$$

$$= \text{com1}[\text{restprog}_k[\pi], j[k], \text{table}[\pi]] * \text{mkhalt}$$

$$= \text{com2}[\text{firstate}[\text{restprog}_k[\pi]], j[k], \text{table}[\pi]] * \text{mksyntra}[$$
$$j[k] + 1 + \text{lengths}[\text{firstate}[\text{restprog}_k[\pi]]]] *$$
$$\text{com1}[\text{restprog}_{k+1}[\pi], j[k] + 1 + \text{lengths}[\text{firstate}[\text{restprog}_k[$$
$$\pi]]], \text{table}[\pi]] * \text{mkhalt} .$$

But by Def. 1.12,

$$s[k+1] = \text{state}[k+1, \pi] = \text{firstate}[\text{restprog}_k[\pi]] .$$

$$\text{LHS} = \text{com2}[s[k+1], j[k], \text{table}[\pi]] * \text{mksyntra}[j[k+1]]$$
$$* \text{com1}[\text{restprog}_{k+1}[\pi], j[k+1], \text{table}[\pi]] * \text{mkhalt} .$$

Hence the lemma is true by induction.

<u>Corollary 2.10.1</u> Let $c[sn, \xi] = n$, $\#[\pi] = p$, $c[IC, \eta] = i$, $1 \leq n \leq p$, and $i = \text{tlu}[n, \text{table}[\pi]]$. Then

$$\text{presentinst}[\text{Compile}[\pi], i] = \text{first}[\text{com2}[\text{state}[n, \pi], i, \text{table}[\pi]]] .$$

Proof: By Lemma 2.10,

$$\text{rest}_{i-1}[\text{Compile}[\pi]] = \text{com2}[\text{state}[n, \pi], i, \text{table}[\pi]] *$$
$$\text{mksyntra}[i + 1 + \text{lengths}[\text{state}[n, \pi]]]$$
$$* \text{com1}[\text{restprog}_n[\pi], i + 1 + \text{lengths}[$$
$$\text{state}[n, \pi]], \text{table}[\pi]] * \text{mkhalt} .$$

By Def. 1.16,

$$\text{presentinst}[\text{Compile}[\pi], i] = \text{first}[\text{rest}_{i-1}[\text{Compile}[\pi]]] .$$

Then from above,

$$presentinst[Compile[\pi], i] = first[com2[state[n, \pi], i, table[\pi]] * \tau]$$

where τ is given the obvious meaning. Applying Lemma 2.2

$$presentinst[Compile[\pi], i] = first[com2[state[n, \pi], i, table[\pi]]] .$$

Lemma 2.11

$$presentinst[Compile[\pi], last[table[\pi]]] = last[Compile[\pi]] .$$

Proof: Let $\#[\pi] = p$ and $i = tlu[p, table[\pi]]$.

Then by Lemma 2.10,

$$rest_{i-1}[Compile[\pi]] = com2[state[p, \pi]\ i, table[\pi]] * mksyntra[i + 1 + lengths[state[p, \pi]]] * com1[restprog_p[\pi], i + 1 + lengths[state[p, \pi]], table[\pi]] * mkhalt .$$

From Def. 1.19, $\#[\pi] = p$ implies $null[restprog_p[\pi]]$. Then using Def. 1.17, and the fact $\tau * NIL * \pi = \tau * \pi$,

$$rest_{i-1}[Compile[\pi]] = com2[state[p, \pi], i, table[\pi]] * mksyntra[i + 1 + lengths[state[p, \pi]]] * mkhalt .$$

Now, from Lemma 2.6, $tlu[p+1, table[\pi]] = last[table[\pi]]$. On the other hand, Lemma 2.5 gives

$$tlu[p+1, table[\pi]] = i + lengths[state[p, \pi]] + 1 .$$

Hence

$$last[table[\pi]] = i + lengths[state[p, \pi]] + 1 .$$

When

$$presentinst[Compile[\pi], last[table[\pi]]] =$$

$$first[rest[rest_{lengths[state[p, \pi]]}[rest_{i-1}[Compile[\pi]]]]] .$$

By above,

$$LHS = first[rest[rest_{lengths[state[p, \pi]]}[com2[state[p, \pi], i, table[\pi]] * mksyntra[i + 1 + lengths[state[p, \pi]]] * mkhalt]]] .$$

By Lemma 2.9

$$LHS = first[rest[mksyntra[i + 1 + lengths[state[p, \pi]]] * mkhalt]]$$
$$= first[mkhalt] = mkhalt .$$

By Def. 1.10 and 1.17

$$last[Compile[\pi]] = mkhalt .$$

Hence

$$presentinst[Compile[\pi], last[table[\pi]]] = last[Compile[\pi]] .$$

Corollary 2.11.1 $ishalt[presentinst[Compile[\pi], last[table[\pi]]]] .$

From the proof of Lemma 2.11,

$$presentinst[Compile[\pi], last[table[\pi]]] = mkhalt .$$

Hence the corollary follows from 1.15.13.

Corollary 2.11.2 For all π and all $k \geq 1$,

$$\neg\, \text{null}[\text{presentinst}[\text{Compile}[\pi], \text{tlu}[k, \text{table}[\pi]]]] \ .$$

Proof:

$\neg\, \text{null}[\text{Compile}[\pi]]$ implies $\neg\, \text{null}[\text{last}[\text{Compile}[\pi]]]$ from Def. 1.19.

Let

$\text{last}[\text{table}[\pi]] = p$.

By Def. 1.16,

$$\text{presentinst}[\text{Compile}[\pi], p] = \text{first}[\text{rest}_{p-1}[\text{Compile}[\pi]]] \ .$$

By Lemma 2.11,

$$\text{presentinst}[\text{Compile}[\pi], p] = \text{last}[\text{Compile}[\pi]] \ .$$

Hence

$$\neg\, \text{null}[\text{first}[\text{rest}_{p-1}[\text{Compile}[\pi]]]] \ .$$

Then, using 1.14.1, for all i such that $1 \leq i \leq p$,

$$\neg\, \text{null}[\text{first}[\text{rest}_{i-1}[\text{Compile}[\pi]]]]$$

or

$$\neg\, \text{null}[\text{presentinst}[\text{Compile}[\pi], i]] \ .$$

By Lemma 2.6 and Corollary 2.2.1, for all k

$$1 \leq \text{tlu}[k, \text{table}[\pi]] \leq p \ .$$

Hence

$\neg$ null[presentinst[Compile[π], tlu[k, table[π]]]] .

The next lemma and its corollary are the basic result. They fundamentally assert that the compiled programs for arithmetic and Boolean expressions actually compute the correct answers.

Lemma 2.12

Let π be an arbitrary program in Mickey. Let θ be an arbitrary input function. Let e be a proper arithmetic expression contained in π . Let P = Compile[π] . By Lemma 1.12, $P = \tau_1 * acom[e, i, k] * \tau_2$, where $i \geq 1$, $k \geq 0$. Let A = vars[π] $\cup$ {out, ctr} $\cup$ {x|x = t[j] $\wedge$ $0 \leq j \wedge j < k$} . Let η be a p.s.v. of Compile[π] in machine. Let c[IC, η] = i . Let presentinst[P, i] = first[acom[e, i, k]] . Then

$$xeq[P, \eta, \theta] =_A xeq[P, a[ac, avalue[e, \eta], a[IC, i + lengthe[e], \eta]], \theta] .$$

Since the set A is a function of k, the notation A(k) will be used to denote the set for a specific k . Clearly $A(k) \subset A(k+1)$. The proof will be by induction upon e, using Def. 1.13.

Case 1. isvar[e] .

Then by Def. 1.17

acom[e, i, k] = mkload[e]

Hence

presentinst[P, i] = first[mkload[e]] = mkload[e] .

By Def. 1.16,

$$LHS = xeq[P, a[IC, i+1, step[mkload[e], \eta, \theta]], \theta]$$

$$LHS = xeq[P, a[IC, i+1, a[ac, c[e, \eta], \eta]], \theta] .$$

Using Lemma 1.2 and Def. 1.12,

$$LHS = xeq[P, a[ac, avalue[e, \eta], a[IC, i+1, \eta]], \theta]$$

$LHS = RHS$. Hence by Lemma 1.5, $LHS =_A RHS$.

Case 2. isconst[e] .

Then

$$acom[e, i, k] = mkli[e] .$$

$$LHS = xeq[P, a[IC, i + 1, a[ac, arg[mkli[e]], \eta]], \theta]$$

$$= xeq[P, a[ac, avalue[e, \eta], a[IC, i+1, \eta]], \theta]$$

$$= RHS .$$

Ergo,

$$LHS =_A RHS .$$

Assume the lemma is true for arg1[e], arg2[e], ante[e], consq[e], arg1[prop[e]], and arg2[prop[e]] as appropriate.

Case I. issum[e] .

$$acom[e, i, k] = acom[arg2[e], i, k] * mksto[t[k]] *$$
$$acom[arg1[e], i + lengthe[arg2[e]] +1, k+1]$$
$$* mkadd[t[k]] .$$

By the induction hypothesis,

$$\text{LHS} =_{A(k)} \text{xeq}[P, a[ac, \text{avalue}[\text{arg2}[e], \eta], a[IC, i + \text{lengthe}[\text{arg2}[e]], \eta]], \theta] .$$

Define η_1 and η_2 by writing the above partial equation as

$$\text{LHS} = {}_{A(k)}\ \eta_1 = \text{xeq}[P, \eta_2, \theta] .$$

$$\text{presentinst}[P, c[IC, \eta_2]] = \text{first}[\text{rest}_{\text{lengthe}[\text{arg2}[e]]}[\text{rest}_{i-1}[P]]]$$

$$= \text{first}[\text{rest}_{\text{lengthe}[\text{arg2}[e]]}[\text{acom}[\text{arg2}[e], i, k] * \text{mksto}[t[k]]$$

$$* \text{acom}[\text{arg1}[e], i + \text{lengthe}[\text{arg2}[e]] + 1, k+1] * \text{mkadd}[t[k]]$$

$$* \tau_2]] = \text{first}[\text{mksto}[t[k]] * \tau_3] = \text{mksto}[t[k]]$$

where τ_2 and τ_3 are defined by the equality. This used Lemma 2.9.

Then

$$\eta_1 = \text{xeq}[P, a[IC, c[IC, \eta_2] + 1, a[t[k], c[ac, \eta_2], \eta_2]], \theta] .$$

Define η_3 and η_4 by $\eta_1 = \eta_3 = \text{xeq}[p, \eta_4, \theta]$ in the above equation.

Similar to above,

$$\text{presentinst}[P, c[IC, \eta_4]] = \text{first}[\text{acom}[\text{arg1}[e], i + \text{lengthe}[\text{arg2}[e]] + 1, k+1]] .$$

Applying the induction hypothesis again

$$\eta_3 =_{A(k+1)} \text{xeq}[P, a[ac, \text{avalue}[\text{arg1}[e], \eta_4], a[IC, c[IC, \eta_4] + \text{lengthe}[\text{arg1}[e]], \eta_4]], \theta] .$$

Define η_5 and η_6 by rewriting the above partial equality as

$$\eta_3 = {}_{A(k+1)}\eta_5 = \text{xeq}[P, \eta_6, \theta] .$$

Similar to above

$$\text{presentinst}[P, c[IC, \eta_6]] = \text{mkadd}[t[k]] .$$

Then

$$\eta_5 = \text{xeq}[P, a[IC, c[IC, \eta_6] +1, a[ac, c[ac, \eta_6] +c\ [t[k], \eta_6], \eta_6]], \theta] .$$

$$\begin{aligned} c[IC, \eta_6] +1 &= c[IC, \eta_4] + \text{lengthe}[\text{arg1}[e]] +1 \\ &= c[IC, \eta_2] +1 + \text{lengthe}[\text{arg1}[e]] +1 \\ &= i + \text{lengthe}[\text{arg2}[e]] +1 + \text{lengthe}[\text{arg1}[e]] +1 \\ &= i + \text{lengthe}[e] . \end{aligned}$$

$$\begin{aligned} c[ac, \eta_6] &= \text{avalue}[\text{arg1}[e], \eta_4] \\ &= \text{avalue}[\text{arg1}[e], \eta_2] = \text{avalue}[\text{arg1}[e], \eta] . \end{aligned}$$

$$c[t[k], \eta_6] = c[t[k], \eta_4] = c[ac, \eta_2] = \text{avalue}[\text{arg2}[e], \eta] .$$

Substituting these into the equation gives

$$\begin{aligned} \eta_5 &= \text{xeq}[P, a[IC, i + \text{lengthe}[e], a[ac, \text{avalue}[\text{arg1}[e], \eta] + \\ &\qquad \text{avalue}[\text{arg2}[e], \eta], \eta_6]], \theta] \\ &= \text{xeq}[P, a[ac, \text{avalue}[e, \eta], a[IC, i + \text{lengthe}[e], \eta_6]], \theta] . \end{aligned}$$

Combining results gives

$$\text{LHS} = {}_{A(k)}\eta_1 = \eta_3 = {}_{A(k+1)}\eta_5 = \text{xeq}[P, a[ac, \text{avalue}[e], a[IC, i + \text{lengthe}[e], \eta_6]], \theta] .$$

Then by Lemma 1.7,

$\eta \ =_{A(k)} \eta_2 \ =_{A(k)} \eta_4 \ =_{A(k)} \eta_6$.

Using Lemma 1.5

$$LHS =_{A(k)} xeq[P, a[ac, avalue[e, \eta], a[IC, i + lengthe[e], \eta_6]], \theta],$$
$$=_A RHS$$

Cases II, III, IV . isdiff[e], isprod[e] isquot[e] .

The proofs are the same as Case I with the replacement of + and mkadd[t[k]] by -, ×, and ÷ and mksub[t[k]], mkmpy[t[k]], and mkdiv[t[k]] respectively.

Case Va. iscond[e] and isequal[prop[e]] .

```
acom[e, i, k] = acom[arg2[prop[e]], i, k] * mksto[t[k]]
    * acom[arg1[prop[e]], i+1 + lengthe[arg2[prop[e]]],
    k+1] * mksub[t[k]] * mktze[i + lengthe[arg2[prop[e]]] +
    lengthe[arg1[prop[e]]] +4] * mkli[1] * mktze[i +
    lengthb[prop[e]] +2 + lengthe[consq[e]]] * acom[consq[e], i +
    lengthb[prop[e]] +1, k] * mktra[i + lengthb[prop[e]] +
    lengthe[ante[e]] + lengthe[consq[e]] +2] * acom[ante[e], i +
    lengthb[prop[e]] +2 + lengthe[consq[e]], k] .
```

Similar to the proof of Case II, define

η_1 = xeq[P, η_2, θ]

η_2 = a[ac, avalue[arg2[prop[e]], η], a[IC, i + lengthe [arg2[prop[e]]], η]]

η_3 = xeq[P, η_4, θ]

η_4 = a[IC, c[IC, η_2] +1, a[t[k], c[ac, η_2], η_2]]

η_5 = xeq[P, η_6, θ]

η_6 = a[ac, avalue[argl[prop[e]], η_4], a[IC, c[IC, η_4]
+ lengthe[argl[prop[e]]], η_4]]

η_7 = xeq[P, η_8, θ]

η_8 = a[IC, c[IC, η_6] +1, a[ac, c[ac, η_6] -c[t[k], η_6], η_6]] .

Then

presentinst[P, c[IC, η_8]] = mktze[i + lengthe[arg2[prop[e]]] +
lengthe[argl[prop[e]]] +4] .

Then

η_7 = xeq[P, a[IC, if c[ac, η_8] = 0 then i + lengthe[arg2[
prop[e]]] + lengthe[argl[prop[e]]] +4 . else
c[IC, η_8] +1, η_8], θ] .

Define η_9 by η_7 = xeq[P, η_9, θ] .

Then

presentinst[P, c[IC, η_9]] = if c[ac, η_8] = 0 then mktze[i +
lengthb[prop[e]] +2 + lengthe[consq[e]]] else mkli[1] .

Subcase Va-1. c[ac, η_8] = 0 .

Then

η_7 = xeq[P, η_{10}, θ] where η_{10} = a[IC, i + lengthb[prop[e]] +2
+ lengthe[consq[e]], η_9] .

presentinst[P, c[IC, η_{10}]] = first[acom[ante[e], i + lengthb[prop [e]] +2 + lengthe[consq[e]], k]] .

Then by the induction hypothesis

η_7 =A(k) xeq[P, a[ac, avalue[ante[e]+ η_{10}], a[IC, i + lengthb[prop [e]] + 2 + lengthe[consq[e]] + lengthe[ante[e]], η_{10}]], θ] .

Subcase Va-2. c[ac, η_8] $\neq$ 0 .

Then

η_7 = xeq[P, η_{11}, θ] where η_{11} = a[IC, c[IC, η_9] +1, a[ac, 1, η_9]] .

presentinst[P, c[IC, η_{11}]] = mktze[i + lengthb[prop[e]] +2 + lengthe[consq[e]]] .

η_7 = xeq[P, η_{12}, θ]

where

η_{12} = a[IC, <u>if</u> c[ac, η_{11}] = 0 <u>then</u> i + lengthb[prop[e]] +2 + lengthe[consq[e]] <u>else</u> c[IC, η_{11}] +1, η_{11}] .

But

c[ac, η_{11}] = 1 $\neq$ 0, hence c[IC, η_{12}] = c[IC, η_{11}] +1 .

Then

presentinst[P, c[IC, η_{12}]] = first[acom[consq[e], i + lengthb[prop[e]] +1, k]] .

Hence, applying the induction hypothesis,

$$\eta_7 =_{A(k)} \text{xeq}[P, \eta_{13}, \theta]$$

where

$$\eta_{13} = a[ac, \text{avalue}[\text{consq}[e], \eta_{12}], a[IC, i + \text{lengthb}[\text{prop}[e]] + \text{lengthe}[\text{consq}[e]], \eta_{12}]] .$$

$$\text{presentinst}[P, c[IC, \eta_{13}]] = \text{mktra}[i + \text{lengthb}[\text{prop}[e]] + \text{lengthe}[\text{ante}[e]] + \text{lengthe}[\text{consq}[e]] + 2] .$$

$$\text{xeq}[P, \eta_{13}, \theta] = \text{xeq}[P, a[IC, i + \text{lengthb}[\text{prop}[e]] + \text{lengthe}[\text{ante}[e]] + \text{lengthe}[\text{consq}[e]] + 2, \eta_{13}], \theta] .$$

Combining subcases Va-1 and Va-2 and noting $\text{lengthe}[e] = 2 + \text{lengthe}[\text{ante}[e]] + \text{lengthe}[\text{consq}[e]] + \text{lengthb}[\text{prop}[e]]$ gives

$$\eta_7 =_{A(k)} \text{xeq}[P, a[IC, i + \text{lengthe}[e], \underline{\text{if}}\ c[ac, \eta_8] = 0\ \underline{\text{then}}\ a[ac, \text{avalue}[\text{ante}[e], \eta_{10}], \eta_{10}]\ \underline{\text{else}}\ \eta_{13}], \theta] .$$

By Lemma 1.2,

$$\begin{aligned} a[IC, i + \text{lengthe}[e], \eta_{13}] &= a[IC, i + \text{lengthe}[e], a[ac, \text{avalue}[\text{consq}[e], \eta_{12}], \eta_{12}]] \\ &= a[IC, i + \text{lengthe}[e], a[ac, \text{avalue}[\text{consq}[e], \eta_{12}], \eta_{11}]] \\ &= a[IC, i + \text{lengthe}[e], a[ac, \text{avalue}[\text{consq}[e], \eta_{12}], \eta_9]] \end{aligned}$$

while

$$a[IC, i + \text{lengthe}[e], a[ac, \text{avalue}[\text{ante}[e], \eta_{10}], \eta_{10}]] =$$

a[IC, i +lengthe[e], a[ac, avalue[ante[e], η_{10}], η_9]] .

Hence the combined result can be written

$\eta_7 =_{A(k)}$ xeq[P, a[IC, i + lengthe[e], a[ac, <u>if</u> c[ac, η_8] = 0 <u>then</u> avalue[ante[e], η_{10}] <u>else</u> avalue[consq[e], η_{12}], η_9]], θ] .

We have the following chain of partial equalities

xeq[P, η, θ] $=_{A(k)}$ η_1 = xeq[P, η_2, θ] = xeq[P, η_4, θ] $=_{A(k+1)}$

xeq[P, η_6, θ] = xeq[P, η_8, θ] $=_{A(k)}$ xeq[P, a[IC, i + lengthe[e], a[ac, <u>if</u> c[ac, η_8] = 0 <u>then</u> avalue[ante[e], η_{10}] <u>else</u> avalue[consq[e], η_{12}], η_9,]], θ] .

We also have as in Case II,

c[ac, η_8] = c[ac, η_6] - c[t[k], η_6] = avalue[argl[prop[e]], η_4]
- avalue[arg2[prop[e]], η] = avalue[argl[prop[e]], η]
- avalue[arg2[prop[e]], η] .

Then

c[ac, η_8] = 0 iff avalue[argl[prop[e]], η] = avalue[arg2[prop[e]], η]
iff bvalue[prop[e], η] .

$\eta_{10} =_{A(k)} \eta_9 =_{A(k)} \eta_8 =_{A(k)} \eta_6 =_{A(k)} \eta_4 =_{A(k)} \eta_2 =_{A(k)} \eta$.

Then by Lemma 1.10,

avalue[ante[e], η] = avalue[ante[e], η_{10}] .

Similarly

$\eta_{12} =_{A(k)} \eta_{11} =_{A(k)} \eta_9 =_{A(k)} \eta$.

Then

avalue[consq[e], η_{12}] = avalue[consq[e], η] .

Combining these results,

xeq[P, η,θ] $=_{A(k)}$ xeq[P, a[ac, <u>if</u> bvalue[prop[e], η]
<u>then</u> avalue[ante[e], η] <u>else</u> avalue[
consq[e], η], a[IC, i + lengthe[e], η]], θ]

o

LHS $=_{A(k)}$ RHS .

Subcase Vb. iscond[e] and isless[prop[e]] .

acom[e, i, k] = acom[arg2[prop[e]], i, k] * mksto[t[k]] *
acom[argl[prop[e]], i + 1 + lengthe[arg2[prop[e]]],
k + 1] * mksub[t[k]] * mktmi[i + lengthe[arg2[prop[e]]]
+ lengthe[argl[prop[e]]] +5] * mkli[1] * mktra[i +
lengthe[arg2[prop[e]]] + lengthe[argl[prop[e]]] +6]
* mkli[0] * mktze[i + lengthb[prop[e]] + 2 + lengthe[
consq[e]]] * acom[consq[e], i + lengthb[prop[e]] +1, k]
* mktra[i + lengthe[e]] * acom[ante[e], i + lengthb[
prop[e]] +2 + lengthe[consq[e]], k] .

The proof is very similar to Va. Define η_1, η_2, η_3, η_4, η_5, η_6 η_7, and η_8 as in Va. All other η_i will be different.

Define

$$\eta_9 = a[IC, \underline{\text{if}}\ c[ac, \eta_8] < 0\ \underline{\text{then}}\ i + \text{lengthe}[\text{arg2}[\text{prop}[e]]] + \text{lengthe}[\text{arg1}[\text{prop}[e]]] + 5\ \underline{\text{else}}\ c[IC, \eta_8] + 1, \eta_8] .$$

Then

$$\eta_7 = \text{xeq}[P, \eta_9, \theta] .$$

Subcase Vb-1. $c[ac, \eta_8] < 0$.

Define

$$\eta_{10} = a[IC, c[IC, \eta_9] + 1, a[ac, 0, \eta_9]] .$$

Then

$$\eta_9 = \text{xeq}[P, \eta_{10}, \theta] .$$

Define

$$\eta_{11} = a[IC, \underline{\text{if}}\ c[ac, \eta_{10}] = 0\ \underline{\text{then}}\ i + \text{lengthb}[\text{prop}[e]] + 2 + \text{lengthe}[\text{consq}[e]]\ \underline{\text{else}}\ c[IC, \eta_{10}] + 1, \eta_{10}] .$$

But

$$c[ac, \eta_{10}] = 0, \text{ hence } \eta_7 = \text{xeq}[P, \eta_{11}, \theta] .$$

Then by the induction hypothesis,

$$\eta_7 =_{A(k)} \text{xeq}[P, a[ac, \text{avalue}[\text{ante}[e], \eta_{10}], a[IC, i + \text{lengthb}[\text{prop}[e]] + 2 + \text{lengthe}[\text{consq}[e]] + \text{lengthe}[\text{ante}[e]], P_{10}]], \theta] .$$

Since

$$\eta_{10} \stackrel{=}{A}(k)\ \eta_9 \stackrel{=}{A}(k)\ \eta_8 \stackrel{=}{A}(k)\ \eta,$$

using Lemma 1.10

$$\eta_7 \stackrel{=}{A}(k)\ \text{xeq[P, a[ac, avalue[ante[e], } \eta\text{], a[IC, i + lengthe[e], } \eta_{10}\text{]], } \theta\text{]} .$$

Subcase Vb-2. $c[ac, \eta_8] \geq 0$.

Define

$$\eta_{12} = a[IC, c[IC, \eta_9] + 1, a[ac, 1, \eta_9]] .$$

Then

$$\eta_7 = xeq[P, \eta_{12}, \theta] .$$

Define

$$\eta_{13} = a[IC, i + lengthe[arg2[prop[e]]] + lengthe[arg1[prop[e]]] + 6, \eta_{12}] .$$

Then

$$\eta_7 = xeq[P, \eta_{13}, \theta] .$$

Define

$$\eta_{14} = a[IC, \underline{if}\ c[ac, \eta_{13}] = 0\ \underline{then}\ 1 + lengthb[prop[e]] + 2 + lengthe[consq[e]]\ \underline{else}\ c[IC, \eta_{13}] + 1, \eta_{13}] .$$

Then

$$\eta_7 = \text{xeq}[P, \eta_{14}, \theta] .$$

But

$$c[ac, \eta_{13}] = c[ac, \eta_{12}] = 1 .$$

Then by the induction hypothesis,

$$\eta_7 =_{A(k)} \text{xeq}[P, a[ac, \text{avalue}[\text{consq}[e], \eta_{14}], a[IC, i + \text{lengthb}[\text{prop}[e]] + 1 + \text{lengthe}[\text{consq}[e]], \eta_{14}]], \theta] .$$

Define η_{15} and η_{16} by rewriting the last partial equality as

$$\eta_7 =_{A(k)} \text{xeq}[P, \eta_{15}, \theta] = \eta_{16} .$$

Define

$$\eta_{17} = a[IC, i + \text{lengthe}[e], \eta_{15}] .$$

Then

$$\eta_{16} = \text{xeq}[P, \eta_{17}, \theta] .$$

Since

$$\eta_{14} =_{A(k)} \eta_{13} =_{A(k)} \eta_{12} =_{A(k)} \eta_9 =_{A(k)} \eta,$$
$$\text{avalue}[\text{consq}[e], \eta_{14}] = \text{avalue}[\text{consq}[e], \eta]$$

Combining subcases Vb-1 and Vb-2, gives

$$\eta_7 =_{A(k)} \text{xeq}[P, \underline{\text{if}}\ c[ac, \eta_8] < 0\ \underline{\text{then}}\ a[ac, \text{avalue}[\text{ante}[e], \eta], a[IC, i + \text{lengthe}[e], \eta_{10}]]\ \underline{\text{else}}\ a[ac, \text{avalue}[\text{consq}[e], \eta], a[IC, i + \text{lengthe}[e], \eta_{15}]], \theta] .$$

But

$$c[ac, \eta_8] = avalue[argl[prop[e]], \eta] - avalue[arg2[prop[e]], \eta]$$

as in Subcase Va. Then

$c[ac, \eta_8] < 0$ iff

avalue[argl[prop[e]], η] < avalue[arg2[prop[e]], η] iff bvalue[prop[e], η] .

Then as in Va,

$$xeq[P, \eta, \theta] =_{A(k)} xeq[P, a[ac, \underline{if}\ bvalue[prop[e], \eta]\ \underline{then}\ avalue[ante[e], \eta]\ \underline{else}\ avalue[consq[e], \eta]\ a[IC, i + lengthe[e], \eta]], \theta] .$$

Hence by induction the lemma is true.

Corollary 2.12.1 Let π, θ, A, η, and i be as in Lemma 2.12. Let e be a proper Boolean expression contained in π . Let P = Compile[π] . By Corollary 1.12.1, $P = \tau_1 * bcom[e, i, k] * \tau_2$ where $k \geq 0$. Let presentinst[P, i] = first[bcom[e, i, k]] . Then

$$xeq[P, \eta, \theta] =_{A(k)} xeq[P, a[ac, \underline{if}\ bvalue[e, \eta]\ \underline{then}\ 0\ \underline{else}\ 1, a[IC, i + lengthb[e], \eta]], \theta] .$$

The proof will be by cases.

Case I. isequal[e] .

The proof is a portion of the proof of Subcase Va in Lemma 2.12, except, appeals to the induction hypotheses can be replaced by appeals to Lemma 2.12.

bcom[e, i, k] = acom[arg2[prop[e]], i, k] * mksto[t[k]] * acom[
arg1[prop[e]], i + lengthe[arg2[prop[e]]], k + 1]
* mksub[t[k]] * mktze[k + lengthe[arg2[prop[e]]] +
lengthe[arg1[prop[e]]] +4] * mkli[1] .

Define η_1, η_2, η_3, η_4, η_5, η_6, η_7, η_8, and η_{11} as in Subcase Va. Then

$$xeq[P, \eta, \theta] =_{A(k)} xeq[P, a[ac, \underline{if}\ bvalue[e, \eta]\ \underline{then}\ 0 \ \underline{else}\ 1, a[IC, i + lengthb[e], \eta]], \theta] .$$

Case II. isless[e] .

As above, this is a portion of the proof of Subcase Vb in Lemma 2.12, replacing the induction hypotheses in that proof by Lemma 2.12, itself. Define η_1, η_2, η_3, η_4, η_5, η_6, η_7, η_8, η_9, η_{10}, and η_{12} as in Subcase Vb. Then, as in Vb,

$$xeq[P, \eta, \theta] =_{A(k)} xeq[P, \eta_8, \theta]$$

and

$$xeq[P, \eta_8, \theta] = xeq[P, a[ac, \underline{if}\ bvalue[e, \eta]\ \underline{then}\ 0\ \underline{else}\ 1, a[IC, i + lengthb[e], \eta]], \theta] .$$

Hence

$$LHS =_{A(k)} RHS$$

This completes the proof. Note in 2.12 and 2.12.1 that the set A(0) equals the set A of Theorem 2.1 (Alternate).

We can now prove the three main theorems. The first asserts that under equivalent starting conditions, a Mickey program π stops iff Compile[π] stops. The second asserts that each stage of a Mickey program affects its state vector similarly to the manner in which the compiled program affects its state vector. The third theorem asserts that the state vectors are strongly partially equivalent.

Theorem 2.13

Let π be an arbitrary program in Mickey. Let θ be an arbitrary input function. Let ξ be a p.s.v. of π in Mickey. Let $A = vars[\pi] \cup \{out, ctr\}$. Let η be a p.s.v. of Compile[π] in machine. Let $\xi =_A \eta$. Let $c[IC, \eta] = tlu[c[sn, \xi], table[\pi]]$. Then

$$end[c[sn, \xi], \pi] \text{ iff } ishalt[presentinst[Compile[\pi], c[IC, \eta]]] .$$

Proof: Let $c[sn, \xi] = k \geq 1$. The theorem will be demonstrated by exhibiting the two equivalences:

(1) $end[k, \pi]$ iff $(\exists j)(0 \leq j < k \text{ and } \#[\pi] = j)$

(2) $ishalt[presentinst[Compile[\pi], tlu[k, table[\pi]]]]$ iff $(\exists j)(0 \leq j < k \text{ and } \#[\pi] = j)$.

The theorem follows by transivity.

To prove (1), first assume $end[k, \pi]$. From Def. 1.12

$$end[k, \pi] = \underline{\text{if}}\ k\text{-}1 = 0\ \underline{\text{then}}\ null[\pi]\ \underline{\text{else}}\ (\underline{\text{if}}\ (k\text{-}1)\text{-}1 = 0\ \underline{\text{then}}\ null[restprog[\pi]]\ \underline{\text{else}}\ (\underline{\text{if}}\ \ldots\ \underline{\text{else}}\ (\underline{\text{if}}\ (k\text{-}(k\text{-}1)\text{-}1 = 0\ \underline{\text{then}}\ null[restprog[\ \ldots\ [\pi]]\ \ldots\ ]]\ \underline{\text{else}}\ end[0, restprog[\ \ldots\ [restprog[\pi]]\ \ldots\ ]])\ \ldots\ )) .$$

Then for a fixed k,

$$end[k, \pi] = null[restprog[\ \ldots\ restprog[\pi]\ \ldots\ ]]$$

where the function restprog is applied k-1 times. Then $end[k, \pi] = null[restprog_{k-1}[\pi]]$. From the assumption, then $restprog_{k-1}[\pi] = NIL$. Then, using 1.11.1

$$\begin{aligned} restprog_{k-2}[\pi] &= NIL && \text{if } null[restprog_{k-2}[\pi]] \\ &= s_{k-1} && \text{otherwise} \end{aligned}$$

where s_{k-1} is some statement. Applying 1.11.1 for k-1 times gives

$$\begin{aligned} \pi &= NIL && \text{if } null[\pi] \\ &= s_1 && \text{if } null[restprog[\pi]] \\ &= s_1 * s_2 && \text{if } null[restprog_2[\pi]] \\ &\ \ \vdots && \ \ \vdots \\ &= s_1 * s_2 * \ldots * s_{k-1} && \text{if } null[restprog_{k-1}[\pi]] \end{aligned}$$

Thus, applying Def. 1.19 and 1.11.1

$$\begin{aligned} \#[\pi] &= 0 && \text{if } null[\pi] \\ &= 1 && \text{if } null[restprog[\pi]] \\ &\ \ \vdots && \ \ \vdots \\ &= k\text{-}1 && \text{if } null[restprog_{k-1}[\pi]] \end{aligned}$$

Hence $(\exists j)(0 \leq j < k \text{ and } \#[\pi] = j)$.

To prove the reverse implication for (1), assume $(\exists j)(0 \leq j < k$ and $\#[\pi] = j)$. Then, as above, $end[k, \pi] = null[restprog_{k-1}[\pi]]$. Assume $\neg\, null[restprog_{k-1}[\pi]]$. Then, using 1.11.1

$$restprog_{k-2}[\pi] = firstate[restprog_{k-2}[\pi]] * restprog_{k-1}[\pi]$$

with neither term on the right null. Then, using 1.11.1 for k-1 times,

$$\pi = \text{firstate}[\pi] * \text{firstate}[\text{restprog}[\pi]] * \ldots$$
$$* \text{firstate}[\text{restprog}_{k-2}[\pi]] * \text{restprog}_{k-1}[\pi]$$

with all k terms on the right side not null. Then

$$\#[\pi] > \text{numb}[\text{restprog}_{k-1}[\pi], k-1] \geq k > j .$$

But by assumption $\#[\pi] = j$. Hence there is a contradiction. Therefore $\text{null}[\text{restprog}_{k-1}[\pi]]$, hence $\text{end}[k, \pi]$. This completes the proof of (1).

To prove (2), first assume

$$\text{ishalt}[\text{presentinst}[\text{Compile}[\pi], \text{tlu}[k, \text{table}[\pi]]]] .$$

Also assume $\#[\pi] = j \geq k$. Let $m = \text{tlu}[k, \text{table}[\pi]]$ and $n = \text{tlu}[k+1, \text{table}[\pi]]$. Then by Lemma 2.6, $n > m$.

By Def. 1.17,

$$\text{ishalt}[\text{presentinst}[\text{Compile}[\pi], m]] =$$
$$\text{ishalt}[\text{presentinst}[\text{coml}[\pi, 1, \text{table}[\pi]] * \text{mkhalt}, m]]$$

By Lemma 2.7, the assumption, and 1.15.13,

$$\text{presentinst}[\text{Compile}[\pi], m] = \text{presentinst}[\text{mkhalt}, 1]$$

Then

$$\text{presentinst}[\text{Compile}[\pi], n] = \text{presentinst}[\text{Compile}[\pi], m+(n-m)] =$$
$$\text{presentinst}[\text{mkhalt}, 1+(n-m)] = \text{presentinst}[\text{rest}[\text{mkhalt}], n-m] =$$
$$\text{presentinst}[\text{NIL}, n-m] = \text{NIL} .$$

The third equality comes from Lemma 2.4, the fourth from Def. 1.16, the last two from Def. 1.16 and 1.15.15. Hence

null[presentinst[Compile[π], n]] .

But this is a contradiction of Corollary 2.11.2. Hence

ishalt[presentinst[Compile[π], m]] implies #[π] = j < k .

For the final stage, assume $0 \leq j < k$ and #[π] = j . Then by Lemma 2.6, tlu[k, table[π]] = last[table[π]] . From Corollary 2.11.1, then ishalt[presentinst[Compile[π], tlu[k, table[π]]]] . This is the final implication, and establishes the theorem.

Theorem 2.14

Let π be an arbitrary program in Mickey. Let θ be an arbitrary input function. Let ξ be a p.s.v. of π in Mickey. Let $A = vars[\pi] \cup \{out, ctr\}$. Let η be a p.s.v. of Compile[π] in machine. Let $\xi =_A \eta$. Let c[IC, η] = tlu[c[sn, ξ], table[π]] = i . Then

execute[π, ξ, θ] $=_A$ xeq[Compile[π], η, θ] .

Also,

c[IC, xeq[Compile[π], η, θ]] = tlu[c[sn, execute[π, ξ, θ]], table[π]] .

Proof: The proof will be by cases. Let s = state[c[sn, ξ], π] .

Case 1. isgoto[s] .

By Def. 1.12,

$$LHS = a[sn, \underline{if}\ bvalue[prop[s], \xi]\ \underline{then}\ dest[s]\ \underline{else}\ c[sn, \xi] + 1, \xi] .$$

By Lemma 1.7, since $sn \notin A$,

$$LHS =_A \xi .$$

Corollary 2.10.1 asserts

$$presentinst[Compile[\pi], c[IC, \eta]] = first[bcom[prop[s], i, 0] * mktze[i + lengths[s] -1] * mktra[i + lengths[s]] * mksyntra[tlu[dest[s], table[\pi]]]] .$$

Then by Corollary 2.12.1,

$$RHS =_A xeq[Compile[\pi], a[ac, \underline{if}\ bvalue[prop[s], \eta]\ \underline{then}\ 0\ \underline{else}\ 1, a[IC, i + lengthb[prop[s]], \eta]], \theta] .$$

Let this right-hand term be defined as η_1, and the second argument of xeq in η_1 as η_2. That is, the last partial equality can be written as

$$RHS =_A \eta_1 \quad \text{or as} \quad RHS =_A xeq[Compile[\pi], \eta_2, \theta] .$$

Then by Def. 1.16,

$$\eta_1 = xeq[Compile[\pi], a[IC, \underline{if}\ c[ac, \eta_2] = 0\ \underline{then}\ addr[mktze[i + lengthb[prop[s]] +2]]\ \underline{else}\ c[IC, \eta_2] +1, \eta_2], \theta] .$$

Then using Lemmas 1.1 and 1.2 and 1.15.10

$$\eta_1 = \text{xeq}[\text{Compile}[\pi], a[IC, \underline{\text{if}}\ \text{bvalue}[\text{prop}[s], \eta]\ \underline{\text{then}}\ i + \text{lengthb}[\text{prop}[s]] + 2\ \underline{\text{else}}\ c[IC, \eta_2] + 1, \eta_2], \theta]$$

$$= \text{xeq}[\text{Compile}[\pi], \underline{\text{if}}\ \text{bvalue}[\text{prop}[s], \eta]\ \underline{\text{then}}\ a[IC, \text{tlu}[\text{dest}[s], \text{table}[\pi]], \eta_2]\ \underline{\text{else}}\ a[IC, i + 1 + \text{lengths}[s], \eta_2], \theta] .$$

The last equality used Lemma 1.1 and 1.2, Def. 1.16, and 1.17. Since $IC \notin A$ and $ac \notin A$, then by Lemmas 1.7 and 1.1, $\eta_1 =_A \eta_2 =_A \eta$. Thus, by hypothesis and as shown,

$$\text{LHS} =_A \xi =_A \eta =_A \eta_2 =_A \eta_1 =_A \text{RHS} .$$

Hence

$$\text{LHS} =_A \text{RHS} .$$

Now for the second result of the theorem. By Lemma 1.3,

$$c[sn, \text{execute}[\pi, \xi, \theta]] = \underline{\text{if}}\ \text{bvalue}[\text{prop}[s], \xi]\ \underline{\text{then}}\ \text{dest}[s]\ \underline{\text{else}}\ c[sn, \xi] + 1,$$

and

$$c[IC, \text{xeq}[\text{Compile}[\pi], \eta, \theta] = \underline{\text{if}}\ \text{bvalue}[\text{prop}[s], \eta]\ \underline{\text{then}}\ \text{tlu}[\text{dest}[s], \text{table}[\pi]]\ \underline{\text{else}}\ c[IC, \eta] + 1 + \text{lengths}[s] .$$

Since $\xi =_A \eta$, using Corollary 1.10.1

$$\text{bvalue}[\text{prop}[s], \xi]\ \text{iff}\ \text{bvalue}[\text{prop}[s], \eta] .$$

By Lemma 2.5,

$$tlu[p+1, table[\pi]] = tlu[p, table[\pi]] + lengths[state[p, \pi] + 1 .$$

Hence

$$\begin{aligned} LHS &= \underline{if}\ bvalue[prop[s], \xi]\ \underline{then}\ tlu[dest[s], table[\pi]] \\ &\quad \underline{else}\ tlu[c[sn, \xi] + 1, table[\pi]] \\ &= tlu[\underline{if}\ bvalue[prop[s], \xi]\ \underline{then}\ dest[s]\ \underline{else} \\ &\quad c[sn, \xi] + 1, table[\pi]] = RHS . \end{aligned}$$

Case 2. isassign[s] .

By Def. 1.12,

$$LHS = a[sn, c[sn, \xi] + 1, a[left[s], avalue[right[s], \xi], \xi]] .$$

Since $sn \notin A$ and by Def. 1.18, $left[s] \in A$, then using Lemma 1.7,

$$LHS =_A a[left[s], avalue[right[s], \xi], \xi] .$$

As in Case 1, using Lemma 2.12, and Corollary 2.10.1,

$$\begin{aligned} RHS =_A xeq[&Compile[\pi], a[ac, avalue[right[s], \eta], a[IC, p + \\ &lengthe[right[s]], \eta]], \theta] . \end{aligned}$$

Define η_1 and η_2 by

$$RHS =_A \eta_1 = xeq[Compile[\pi], \eta_2, \theta] .$$

Then, by Def. 1.16 and 1.15.7

$$\eta_1 = \text{xeq}[\text{Compile}[\pi], a[IC, c[IC, \eta_2] + 1, a[\text{left}[s], c[ac, \eta_2], \eta_2]], \theta]$$
$$= \text{xeq}[\text{Compile}[\pi], a[IC, c[IC, \eta_2] + 1, a[\text{left}[s], \text{avalue}[\text{right}[s], \eta], \eta_2]], \theta]$$
$$= a[IC, i + 1 + \text{lengths}[s], a[\text{left}[s], \text{avalue}[\text{right}[s], \eta], \eta_2]] .$$

This last equality used Def. 1.17, 1.16, and 1.15.14 and Lemma 1.1 and 1.2. Since $ac \notin A$ and $IC \notin A$,

$$\eta_1 =_A a[\text{left}[s], \text{avalue}[\text{right}[s], \eta], \eta] .$$

Then by Lemma 1.10 and above

$$\text{LHS} =_A a[\text{left}[s], \text{avalue}[\text{right}[s], \xi], \xi]$$
$$=_A a[\text{left}[s], \text{avalue}[\text{right}[s], \eta], \xi]$$
$$=_A a[\text{left}[s], \text{avalue}[\text{right}[s], \eta], \eta]$$
$$=_A \eta_1 =_A \text{RHS} .$$

The second result in this case is straightforward.

$$c[sn, \text{execute}[\pi, \xi, \theta]] = c[sn, \xi] + 1 .$$
$$c[IC, \text{xeq}[\text{Compile}[\pi], \eta, \theta]] = i + 1 + \text{lengths}[s] .$$

Hence by Lemma 2.5,

$$c[IC, \text{xeq}[\text{Compile}[\pi], \eta, \theta]] = \text{tlu}[c[sn, \text{execute}[\pi, \xi, \theta]], \text{table}[\pi]] .$$

Case 3. $\text{isread}[s]$.

By Def. 1.12,

$$\text{LHS} = a[sn, c[sn, \xi] + 1, a[ctr, c[ctr, \xi] + 1, a[\text{argum}[s], \text{read}[\theta, c[ctr, \xi]], \xi]]] .$$

As above,

$$LHS =_A a[ctr, c[ctr, \xi] +1, a[argum[s], read[\theta, c[ctr, \xi]], \xi]] .$$

Again, as above, using Lemma 2.12 and Def. 1.17, 1.16, 1.15.1, 1.15.3, 1.15.7, and 1.15.14,

$$RHS = a[IC, i + 1 + lengths[s], \eta_4]$$

where

$$\eta_4 = a[IC, c[IC, \eta_3] +1, a[ctr, c[ac, \eta_3], \eta_3]]$$

and

$$\eta_3 = a[IC, c[IC, \eta_2] +1, a[ac, c[ac, \eta_2] +c[ctr, \eta_2], \eta_2]]$$

and

$$\eta_2 = a[IC, c[IC, \eta_1] +1, a[ac, \eta_1]]$$

and

$$\eta_1 = a[IC, c[IC, \eta] +1, a[argum[s], read[\theta, c[ctr, \eta]], \eta]] .$$

Since $IC \notin A$ and $ac \notin A$,

$$RHS =_A \eta_4 =_A a[ctr, c[ac, \eta_3]]$$

$$\eta_3 =_A \eta_2 =_A a[argum[s], read[\theta, c[ctr, \eta]], \eta].$$

By Lemma 1.1

$$c[ac, \eta_3] = c[ac, \eta_2] + c[ctr, \eta_2]$$

$$c[ac, \eta_2] = 1$$

$$c[ctr, \eta_2] = c[ctr, \eta_1] = c[ctr, \eta] .$$

Since $ctr \in A$ and $\xi =_A \eta$, then $c[ctr, \eta] = c[ctr, \xi]$. Then

$$RHS =_A a[ctr, 1 + c[ctr, \xi], \eta_3]$$

and

$$\eta_3 =_A a[argum[s], read[\theta, c[ctr, \xi]], \eta] .$$

Define

$$\xi_1 = a[argum[s], read[\theta, c[ctr, \xi]], \xi] .$$

Then by hypothesis and Lemma 1.6,

$$\eta_3 =_A \xi_1 .$$

Then

$$LHS =_A a[ctr, c[ctr, \xi] +1, \xi_1] =_A a[ctr, 1+ c[ctr, \xi], \eta_3] =_A RHS .$$

The second half of the theorem for this case is proven identically with that of Case 2.

Case 4. iswrite[s] .

$$LHS = a[sn, c[sn, \xi] +1, a[out, c[out, \xi] * c[argum[s], \xi], \xi]] .$$

$$LHS =_A a[out, c[out, \xi] * c[argum[s], \xi], \xi] .$$

As above,

$$RHS = a[IC, i + 1 + lengths[s], \eta_1]$$

where

$$\eta_1 = a[IC, c[IC, \eta] +1, a[out, c[out, \eta] * c[argum[s], \eta], \eta]] .$$

Then

$$RHS =_A a[out, c[out, \eta] * c[argum[s], \eta], \eta] .$$

But by Def. 1.18, hypothesis, Def. 1.8, and Lemma 1.9

$$c[out, \eta] = c[out, \xi] \quad \text{and} \quad c[argum[s], \eta] = c[argum[s], \xi] .$$

Then

$$LHS =_A a[out, c[out, \xi] * c[argum[s], \xi], \xi] = a[out, c[out, \eta] * c[argum[s], \eta], \eta] =_A RHS .$$

This shows the first half of the theorem. The second half is identical with the proof in Case 2. Hence, the theorem is proven.

Theorem 2.1'

Under the hypotheses of Theorem 2.13 and 2.14, then

$$Mickey[\pi, \xi, \theta] \tilde{=}_A machine[Compile[\pi], \eta, \theta]$$

Proof: Define two sequences ξ_i and η_i .

Define

$$\xi_0 = \xi$$

$$\xi_{i+1} = \xi_i \qquad \text{if } end[c[sn, \xi_i], \pi]$$

$$= execute[\pi, \xi_j, \theta] \qquad \text{otherwise}$$

Define

$$\eta_0 = \eta$$

$\eta_{i+1} = \eta_i$ if ishalt[presentinst[Compile[π], c[IC, η_i]]]

$=$ xeq[Compile[π], otherwise

η_i, θ]

By hypothesis, $\xi_0 =_A \eta_0$. Let K be the set of indices such that $k \in K$ implies $\xi_k \neq_A \eta_k$ i.e., K is the set of indices over which partial equality does not hold. Assume K is non-empty. Let $j \in K$ be the smallest index by the well-ordering principle. That is $\xi_{j-1} =_A \eta_{j-1}$ but $\xi_j \neq_A \eta_j$.

If

end[c[sn, ξ_{j-1}], π],

then by Theorem 2.13,

ishalt[presentinst[Compile[π], c[IC, η_{j-1}]]] .

Then by the definitions of ξ_i and η_i, $\xi_j = \xi_{j-1} =_A \eta_{j-1} = \eta_j$. Then $\xi_j =_A \eta_j$, a contradiction.

Then

$\neg$ end[c[sn, ξ_{j-1}], π] .

Again by Theorem 2.13,

$\neg$ ishalt[presentinst[Compile[π], c[IC, η_{j-1}]]] .

Then by the definitions,

ξ_j = execute[π, ξ_{j-1}, θ]

and

$$\eta_j = \text{xeq}[\text{Compile}[\pi], \eta_{h-1}, \theta] .$$

But then by Theorem 2.14, $\xi_j =_A \eta_j$, a contradiction. Hence K must be empty, i.e., for all i, $\xi_i =_A \eta_i$.

By the definitions we have

$$\text{Mickey}[\pi, \xi_i, \theta] = \underline{\text{if}}\ \text{end}[c[\text{sn}, \xi_i], \pi]\ \underline{\text{then}}\ \xi_i\ \underline{\text{else}}\ \text{Mickey}[\pi, \xi_{i+1}, \theta]$$

also

$$\text{machine}[\text{Compile}[\pi], \eta_i, \theta] = \underline{\text{if}}\ \text{ishalt}[\text{presentinst}[\text{Compile}[\pi], c[\text{IC}, \eta_i]]]\ \underline{\text{then}}\ \eta_i\ \underline{\text{else}}\ \text{machine}[\text{Compile}[\pi], \eta_{i+1}, \theta] .$$

Since

$$\text{Mickey}[\pi, \xi, \theta] = \text{Mickey}[\pi, \xi_0, \theta],$$

LHS is undefined iff for all i,

$$\neg\ \text{end}[c[\text{sn}, \xi_i], \pi] .$$

But then by Theorem 2.13 that is true iff

$$\neg\ \text{ishalt}[\text{presentinst}[\text{Compile}[\pi], c[\text{IC}, \eta_i]]],$$

i.e.,

$$\text{machine}[\text{Compile}[\pi], \eta_0, \theta] = \text{machine}[\text{Compile}[\pi], \eta, \theta]$$

is undefined.

If Mickey[π, ξ, θ] is defined, then for some smallest n, Mickey[π, ξ, θ] = ξ_n, also end[c[sn, ξ_n], π] . But from above $\xi_n =_A \eta_n$, hence also ishalt[presentinst[Compile[π], c[IC, η_n]]] . Hence machine[Compile[π], η, θ] = η_n . Then Mickey[π, ξ, θ] $=_A$ machine[Compile[π], η, θ] . Combining the undefined and defined cases gives,

$$\text{Mickey}[\pi, \xi, \theta] \tilde{=}_A \text{machine}[\text{Compile}[\pi], \eta, \theta] .$$

Now Theorem 2.1 (Alternate) can be proven. Actually Theorem 2.15 is a stronger result, and Theorem 2.1 follows immediately once it is shown that c[IC, η] = c[sn, ξ] = 1 implies c[IC, η] = tlu[c[sn, ξ], table[π]] .

Proof: From the hypotheses, and Theorem 2.15 it is sufficient to show tlu[1, table[π]] = 1 for all π . But this is Corollary 2.2.1. Hence 2.1 is proven, i.e., Compile[π] is correct.

An examination of the previous proof shows that it can be modified to include various extensions and restrictions upon Mickey and the target machine.

Corollary 2.1.1 There cannot exist a correct compiler for an arbitrary extension of Mickey with machine as the target computer.

Let Mickey be extended by adding the clause "if isnocomp[e] then noncomp[avalue[arg1[e], ξ], avalue[arg2[e], ξ]] else" to the definition of avalue[e, ξ], where noncomp[e, ξ] is some arbitrary noncomputable function and isnocomp[e] is its associated predicate. By the definition of a noncomputable function, machine cannot compute its value, hence

there can exist no compiler which can correctly translate this extension of Mickey to machine.

Corollary 2.1.2 Compile is a correct compiler for any language which is a subset of Mickey with machine as target.

More accurately, let Mick be any language which satisfies the analytic syntax and semantics of Mickey, albeit certain clauses are satisfied vacuously. Then, under the standard hypotheses of Theorem 2.1 modified in the obvious manner, $\text{Mick}[\pi, \xi, \theta] \tilde{=}_A \text{machine}[\text{Compile}[\pi], \eta, \theta]$.

But clearly the proof of Theorem 2.1 can be modified to a proof of this corollary by deleting in each proof by cases, those cases which cannot occur in Mick .

Corollary 2.1.3 Compile is a correct compiler for Mickey to any machine which includes machine as a subset.

More accurately, let outcomeX be the semantic definition of a machine X which includes machine as part of its definition together with more instructions and their definitions. Then

$$\text{Mickey}[\pi, \xi, \theta] \tilde{=}_A \text{outcomeX}[\text{Compile}[\pi], \eta, \theta] .$$

Again the proof of Theorem 2.1 is sufficient to prove this corollary. The existence of instructions which cannot be generated by the compiler clearly cannot affect the proof.

Corollary 2.1.4 There cannot exist a correct compiler for Mickey with an arbitrary restriction of machine as the target.

Consider machineX defined by deleting the instruction rd and its associated predicate and defining clause in the semantics of machine.

Now take the Mickey program read[s] * write[s] . Clearly for different read functions θ, i.e., sets of input data, c[out, ξ] differs. But since the compiled program is a constant for any compiler, say compileY, and machineX is deterministic; outcomeX[compileY[read[x] * write[x]], η, θ] and c[out, ξ] are constant, ergo, cannot always equal Mickey[π, ξ, θ] . Hence compileY is not correct. Then a correct compiler cannot exist.

It must be noted that Corollaries 2.1.1 to 2.1.4 do not preclude all extensions to Mickey nor all restrictions to machine. For example, Mickey could be extended to permit disjunction, conjunction, and negation upon the values of the predicates. This could be accomplished by representing true(false) by 0(1) and representing the functions $x \vee y$, $x \wedge y$, and $\neg x$ by $x \times y$, $(x \times y) + x+y$, and $x+1$ (all modulo 2), and a correct compiler could then be described directly. Likewise, the instruction add could be deleted and all sums x+y performed as x-((y-y)-y) using only subtraction. Another possible class of problems is generated by extending (or restricting) both the source language and target machines. A trivial set is where both equivalent operations are added or deleted, e.g., $\vee$ and $\times$ (mod2), $\neg$ and +1 (mod2), read and rd, sum and +, etc. The more interesting cases are where only one of an equivalent pair is changed. For instance, if mpy is deleted, while isprod remains, then if the range of the function avalue is the integers or digital numbers, repeated additions can be used to substitute for mpy . However, if the range of avalue is the reals, there can be no equivalence.

APPENDIX I

M. Davis [1] presents a definition of Turing machines, and then proceeds to prove that they are universal. To this end, he proves the equivalence of the classes of (partially) computable functions, i.e., those represented by computations of Turing machines, and (partially) recursive functions. He then uses the normal form theorem to deduce the existence of a Turing machine, which, given a representation of any other Turing machine and its data, can compute the same result as the simulated machine. To prove the universality of Mickey it is sufficient to demonstrate that, given any Turing machine, a Mickey program can be written which produces the same results. Then the existence of a universal Turing machine implies the existence of a universal Mickey program.

A Turing machine consists of a finite set of quadruples, a finite set of internal configurations or states, (including a distinguished one), and a linear tape arbitrarily long in both directions, containing symbols from a finite alphabet such that one symbol can be scanned or written over, or the position changed such that the symbol adjacent to the left (right) can be scanned or written. Clearly, the Mickey input/output facilities are not compatible with the Turing machine tape. The original contents of the Turing tape can be assumed to be upon the Mickey input (called tape for convenience), but the Turing tape operations cannot be performed upon Mickey tape. This means a simulated tape is required. It is known that an alphabet of two symbols is sufficient in the sense there exists a universal Turing machine using a two character alphabet only [2]. This fact simplifies the discussion, so an alphabet

of two characters will be assumed. The identity of the characters is clearly immaterial, so we will use the set $\{0,1\}$. 0 is the special symbol such that it is assumed to exist on the tape arbitrarily far in both directions. Thus the input tape is a sequence of 0's and 1's, beginning and ending with arbitrarily many 0's. Having selected an arbitrary point as the center of the Turing tape, the contents of the tape can be represented in Mickey by two cells, say <u>lefttape</u> and <u>righttape</u>, by

$$\underline{\text{lefttape}} = \prod_{i=1}^{\infty} P_i^{n_i}$$

$$\underline{\text{righttape}} = \prod_{i=1}^{\infty} P_i^{m_i}$$

where P_i is the $i+1^{\underline{st}}$ prime, i.e., $P_1 = 3$, $P_2 = 5$, ... and $n_i(m_i)$ is the symbol 0 or 1, occupying the $i^{\underline{th}}$ position on the tape to the left (right) of center. For example, the tape ... 0011 [center] 10100 ... is represented by

$$\underline{\text{lefttape}} = 3^1 \cdot 5^1 \cdot 7^0 \cdot 11^0 \ldots = 3.5$$
$$\underline{\text{righttape}} = 3^1 \cdot 5^0 \cdot 7^1 \cdot 11^0 \cdot \ldots = 3.7$$

Three cells, <u>pointer</u>, <u>side</u>, and <u>realtape</u>, are used to mark the present position i on the simulated tape and to indicate if more data must be actually read from the real Mickey tape. For the present position i, <u>pointer</u> contains P_i. <u>side</u> is Boolean and contains +1 (-1) if <u>righttape</u> (<u>lefttape</u>) is the cell containing the appropriate half tape.

realtape is also Boolean and contains +1 if the next data to the right is upon the real Mickey tape. There is no way to determine if all the real data has been read from the Mickey tape or not, so only one character at a time is read from the Mickey tape, if needed. Otherwise, the Turing tape movement is simulated by arithmetic operations. The operation starts by reading a character from the real Mickey tape and forming a copy of the Turing tape in lefttape and righttape . Any rewriting or moving left is done solely by arithmetic simulation upon lefttape and righttape . Any reading to the right beyond previously read tape is done by reading Mickey tape and copying the data into lefttape and righttape . Any movement right over previously read tape is performed solely upon lefttape and righttape .

The set of quadruples in a Turing machine contain the following 3 types:

(1) $q_i \; S_j \; S_k \; q_\ell$

(2) $q_i \; S_j \; R \; q_\ell$

(3) $q_i \; S_j \; L \; q_\ell$

Davis defines a fourth type, but it is only used in defining relative computability. This extension is not required, and hence the fourth type will be ignored. The S_j and S_k are elements of the alphabet, i.e., are 0's or 1's . The q_i and q_ℓ are internal configurations of the Turing machine. R(L) indicates a scanning motion one character right (left). The first quadruple indicates that in configuration q_i and observing S_j on the tape, replace S_j by S_k and enter

configuration q_ℓ . The second (third) quadruple indicates that in configuration q_i and observing S_j, move one character right (left) and enter configuration q_ℓ .

The Mickey program for representing a Turing machine then will consist of an initialization of the tape simulation, a collection of subprograms (one for each quadruple), and a terminal section to decipher the simulated tape in lefttape and righttape and produce the proper output. Since Mickey is so rudimentary, it is not possible to program directly in it without an explicit Turing machine to simulate. Instead a model language will be used, so that given a specific set of quadruples, the Mickey program could be generated. For instance, Mickey does not have symbolic labels for statements, only statement numbers. The actual assignment of statement numbers is only possible with a complete program. Since prototype Mickey programs will be displayed for each quadruple, a symbolic model language will be used. Hence in the model language *l*:S will be allowable, where *l* is a symbol and S is any Mickey statement. Also the conditional go to will be of the form

if boolean expression then go to *l*

where *l* is a symbol. The intention is that to translate from the model language to Mickey, all labels in labeled statements will be deleted, and all references to them in go to statements will be replaced by the actual statement number. The other statements will be of the form

variable ← arithmetic expression

read (variable)

write (variable)

Now for the universal Mickey program, the initialization consists of:

lefttape ← 1

righttape ← 1

realtape ← 1

pointer ← 3

side ← 1

read (char)

if char = 0 then go to $q_1 0$

righttape ← righttape × pointer

if 0 = 0 then go to $q_1 1$

For each quadruple, there is a block of statements, depending upon the type of quadruple. For the first type, assume it is $q_i S_j S_k q_\ell$.

Then there are three cases.

Case I. $S_j = S_k$. Then the code is

$q_i S_j$: if 0 = 0 then go to $q_\ell S_j$

Case II. $S_j < S_k$. That is $q_i 0 1 q_\ell$. Then the code is

$q_i S_j$: char ← 1

if side = 1 then go to $q_i S_j \alpha$

lefttape ← lefttape × pointer

if 0 = 0 then go to $q_\ell 1$

$q_i S_j \alpha$: righttape ← righttape × pointer

if 0 = 0 then go to $q_\ell 1$

Case III. $S_j > S_k$. That is $q_i 10 q_\ell$. Then the code is

$q_i S_j$: char ← 0

if side = 1 then go to $q_i S_j \alpha$

lefttape ← lefttape ÷ pointer

if 0 = 0 then go to $q_\ell 0$

$q_i S_j \alpha$: righttape ← righttape ÷ pointer

if 0 = 0 then go to $q_\ell 0$

Certain inline subprograms are used in the models for the second and third types of quadruples. These are FIND SMALLER PRIME, FIND LARGER PRIME, and FIND EXPONENT . The expansions are given below. It is intended that the proper expansions be inserted in the model blocks at the appropriate places. These subprograms do basically what their names imply. They have the function of resetting the cells pointer and char in addition. The models are as follows:

FIND SMALLER PRIME

α : pointer ← pointer -2

if pointer = 3 then go to exit

data ← pointer

```
β : data ← data -2
    datal ← data
    data2 ← datal
    if  data = 1  then go to  exit
γ : if  pointer = data2  then go to  α
    if  pointer < data2  then go to  β
    data2 ← datal + data2
    if  0 = 0  then go to  γ
exit: pointer ← pointer
     FIND LARGER PRIME
α : pointer ← pointer +2
    data ← pointer
β : data ← data -2
    datal ← data
    data2 ← datal
    if  data = 1  then go to  exit
γ : if  pointer = data2  then go to  α
    if  pointer < data2  then go to  β
    data2 ← datal + data2
    if  0 = 0  then go to  γ
exit : pointer ← pointer
     FIND EXPONENT
    char ← 0
    if  side = 1  then go to  α
    data ← lefttape
```

```
β : if  data < pointer  then go to  exit
    if  data = pointer  then go to  γ
    data ← data - pointer
    if  0 = 0  then go to  β
α : data ← righttape
    if  0 = 0  then go to  β
γ : char ← 1
exit : char ← char
```

Now the prototypes for the second and third types of quadruples can be exhibited. The second type of quadruple can be typified by $q_iS_jRq_\ell$. The prototype for this quadruple is

```
q_iS_j  : if  realtape = 1  then go to  q_iS_jα
          realtape ← realtape +1
          if  side = 1  then go to  q_iS_jβ
          if  pointer = 3  then go to  q_iS_jγ
          FIND SMALLER PRIME
          FIND EXPONENT
          if  char = 0  then go to  q_ℓ 0
          if  0 = 0  then go to  q_ℓ 1
q_iS_jα : FIND LARGER PRIME
          read (char)
          if  char = 0  then go to  q_ℓ 0
          righttape ← righttape x pointer
          if  0 = 0  then go to  q_ℓ 1
```

$q_i S_j \beta$: FIND LARGER PRIME

FIND EXPONENT

if char = 0 then go to q_ℓ 0

if 0 = 0 then go to q_ℓ 1

$q_i S_j \gamma$: side ← 1

FIND EXPONENT

if char = 0 then go to q_ℓ 0

if 0 = 0 then go to q_ℓ 1

Let the third type of quadruple be represented as $q_i S_j L q_\ell$. The Mickey code for the prototype is

$q_i S_j$: realtape ← realtape -1

if side = 1 then go to $q_i S_j \alpha$

FIND LARGER PRIME

FIND EXPONENT

if char = 0 then go to q_ℓ 0

if 0 = 0 then go to q_ℓ 1

$q_i S_j \alpha$: if pointer = 3 then go to $q_i S_j \beta$

FIND SMALLER PRIME

FIND EXPONENT

if char = 0 then go to q_ℓ 0

if 0 = 0 then go to q_ℓ 1

$q_i S_j \beta$: side ← -1

FIND EXPONENT

if char = 0 then go to q_ℓ 0

if 0 = 0 then go to q_ℓ 1

By using these prototypes, Mickey code can be produced for each quadruple in a Turing machine. There are $4m^2$ possible distinct quadruples of the first type assuming 2 alphabetic characters and m internal configurations. There are in addition $4m^2$ possible distinct quadruples of the second and third types. Since only deterministic (or consistent) Turing machines are being considered, there are $2m$ legal possible quadruples out of the possible $8m^2$ quadruples. If all $2m$ quadruples exist for a Turing machine, it can never terminate. Therefore, except for the Turing machine which computes the totally undefined function, at least one quadruple is not defined, and is a terminating case. After producing the blocks of statements for each given quadruple, for all remaining legal combinations of the $2m$ legal ones, say configuration q_i and character S_j, produce the code

$$q_iS_j : \underline{\text{if}}\ \ 0 = 0\ \ \underline{\text{then}}\ \underline{\text{go to}}\ \ \text{terminal}$$

"terminal" is the symbol for the first statement in the terminal phase.

The only thing remaining is the terminal phase to translate from the internal form of lefttape and righttape into the proper output. This is straightforward, but tedious, and will not be presented.

Given these prototypes, and a specific implementation of the universal Turing machine, it is mechanical to generate a Mickey program to simulate the same function. Hence Mickey is a universal language.

APPENDIX II

The following is an alphabetical list of all functions and predicates used in this thesis. The "P" appearing before the definition number indicates the function or predicate is primitive to this thesis, i.e., not defined in terms of other functions or predicates.

Function or Predicate		Definition
a		1.7
acom		1.17
addition	P	1.0
addr	P	1.14
ante	P	1.11
arg	P	1.14
argum	P	1.11
argl	P	1.11
arg2	P	1.11
arithmetic contained in arithmetic		1.11
arithmetic contained in program		1.11
avalue		1.12
bcom		1.17
Boolean contained in program		1.11
bvalue		1.12
c		1.6
coml		1.17
com2		1.17
Compile		1.17
compiler definition		1.17

Function or Predicate (continued)		Definition
concatenation	P	1.0
conditional expression	P	1.0
conjunction	P	1.0
consq	P	1.11
correct compiler		1.10
dest	P	1.11
difference	P	1.0
disjunction	P	1.0
division	P	1.0
empty string	P	1.0
end		1.12
equal	P	1.0
e.s.v.		1.3
execute		1.12
existential quantification	P	1.0
explicitvars[π,machine]		1.18
explicitvars[π,Mickey]		1.18
extended state vector		1.3
first	P	1.14
firstate	P	1.11
implication	P	1.0
implicitvars[π,machine]		1.18
implicitvars[π,Mickey]		1.18
inclusion	P	1.0
induction for Mickey arithmetic expression	P	1.13
intersection	P	1.0

Function or Predicate (continued)		Definition
iovariables[π,machine]		1.18
iovariables[π,Mickey]		1.18
isadd	P	1.14
isae		1.11
isassign	P	1.11
isbe		1.11
iscond	P	1.11
isconst	P	1.11
isdiff	P	1.11
isdiv	P	1.14
isequal	P	1.11
isgoto	P	1.11
ishalt	P	1.14
isless	P	1.11
isli	P	1.14
isload	P	1.14
ismpy	P	1.14
isprod	P	1.11
isprogram	P	1.14
isprogramM	P	1.11
isquot	P	1.11
isrd	P	1.14
isread	P	1.11
issto	P	1.14
issub	P	1.14
issum	P	1.11
issyntra	P	1.14

Function or Preciate (continued)	Definition	
istmi	P	1.14
istra	P	1.14
istze	P	1.14
isvar	P	1.11
iswrite	P	1.11
iswrt	P	1.14
last		1.19
left	P	1.11
lengthb		1.17
lengthe		1.17
lengths		1.17
less	P	1.0
machine		1.16
machine analytic syntax		1.14
machine regularity		1.15
machine semantic definition		1.16
machine synthetic syntax		1.14
machvars		1.18
machvarsa		1.18
membership	P	1.0
Mickey		1.12
Mickey analytic syntax		1.11
Mickey semantic definition		1.12
mkadd	P	1.14
mkdiv	P	1.14

Function or Predicate (continued)	Definition	
mkhalt	P	1.14
mkli	P	1.14
mkload	P	1.14
mkmpy	P	1.14
mkrd	P	1.14
mksto	P	1.14
mksub	P	1.14
mksyntra	P	1.14
mktmi	P	1.14
mktra	P	1.14
mktze	P	1.14
mkwrt	P	1.14
multiplication	P	1.0
negation	P	1.0
non-membership	P	1.0
num		1.19
numb		1.19
ordered pair	P	1.0
partial equality		1.8
p.e.s.v.		1.4
presentinst		1.16
principal state vector		1.2
prop	P	1.11
proper extended state vector		1.4
p.s.v.		1.2

Function or Predicate (continued)		Definition
range[x, machine]		1.18
range[x, Mickey]		1.18
rest	P	1.14
restprog	P	1.11
right	P	1.11
sequencer[machine]		1.18
sequencer[Mickey]		1.18
state		1.12
statement		1.11
state vector		1.1
step		1.16
strong partial equality		1.9
subtraction	P	1.0
t		1.17
tabl		1.17
table		1.17
tlu		1.17
union	P	1.0
universal quantification	P	1.0
value of constant	P	1.0
variables		1.5
vars		1.18
varsa		1.18
varsb		1.18
varsc		1.18
xeq		1.16

Function or Predicate (continued)	Definition
#	1.19
#inst	1.19

REFERENCES

1. Davis, M., Computability and Unsolvability. McGraw-Hill Book Co., Inc., New York, N. Y., 1958.

2. Kleene, S. C., Introduction to Metamethematics. D. Van Nostrand Co., Inc., Princeton, N. J., 1950.

3. McCarthy, J., Towards a Mathematical Theory of Computation. In Proceedings of the IFIP Congress, Munich, 1962. North-Holland Publishing Co., Amsterdam, 1963, pp. 21 - 28.

4. McCarthy, J., A Formal Description of a Subset of Algol. In Formal Language Description Languages for Computer Programming, T. B. Steel (Editor). North-Holland Publishing Co., Amsterdam. 1966, pp. 1 - 7.

5. McCarthy, J., and Painter, J. A., Correctness of a Compiler for Arithmetic Expressions. To be published in the proceedings of the symposium on Mathematical Aspects of Computer Science of the American Mathematical Society, 1966.

6. Rabin, M. O., and Scott, D., Finite Automata and Their Decision Problems. IBM Journal of Research and Development, vol. 3, no. 2, April 1959, pp. 114 - 125.

7. Schutzenberger, M. P., Context-free Languages and Push-down Automata. Information and Control, vol. 6, 1963, pp. 246 - 264.

8. Sheperdson, J. C., and Sturgis, H. E., Computability of Recursive Functions. Journal of the Association for Computing Machinery, vol. 10, 1963, pp. 217 - 255.