SPECIFICATION AND VERIFICATION OF CONTEXT CONDITIONS FOR PROGRAMMING LANGUAGES

BY

SIMON MARK KAPLAN

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS OF THE
DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF CAPE TOWN

1986

ABSTRACT

Context conditions – also called static semantics – are the constraints on computer programs that cannot be reasonably expressed by a context–free grammar, but that can be statically checked without considering the execution properties – semantics – of the program. Such conditions tend to be arbitrary and complex.

This thesis presents a new specification formalism called CFF/AML. This formalism is designed to be both useful for the specification of programming languages to an environment generator and also simple to use.

The driving insight behind CFF/AML is that a language specifier conceives of the context condition checks associated with a programming language syntax description in procedural terms. CFF/AML supports this view of context condition specification, thus simplifying the task of the language specifier.

CFF/AML has been formally by constructing a temporal proof system for the metalanguage. This proof system can also be used to verify CFF/AML specifications.

The construction of the temporal proof system for CFF/AML uncovered a deficiency in the existing theory, namely that there was no way to prove subprograms, especially recursive subprograms, correct. The theory was extended to handle recursive subprograms. The approach developed in this thesis allows recursive subprograms to be proven correct using the same approach as was used previously for iterative constructs.

This thesis makes a number of contributions to Computer Science. An approach to language specification – CFF/AML – is developed that greatly reduces the problems associated with building a language specification for input to a programming language environment generator. The theory of temporal proof systems is extended to include a methodology for handling proofs of recursive subprograms. A formal description of the CFF/AML metalanguage has been developed using temporal logic as the framework for the description. This is the first attempt to use temporal logic for such a task. As CFF/AML constructs can be dynamically scoped, this development differs from that required for statically scoped languages. We have also used this temporal proof system formally to prove that context condition specifications are correct. These proofs are an advancement on earlier work in the field of formal reasoning about context condition specification as they allow formal proof of the correctness of evaluations, as well as proving termination.

ACKNOWLEDGEMENTS

First and foremost, I gratefully acknowledge the assistance of my supervisor and friend Ken MacGregor, whose insights and intellectual guidance had a major impact on this project.

Tim McDermott's assistance also was invaluable. His keen insights and penetrating questions boosted me on my way, and he motivated me to be interested in the myriad problems of context condition specification in the first place.

The implementation of the prototype of the CFF/AML compiler front end was assisted by Jeremy Druker, Ian Friedman and Sue Olivier.

Special thanks to my friends, colleagues and family who supported me morally and emotionally whilst this project took shape and reached completion.

Finally, most heartfelt thanks to my wife Philippa, for her understanding, love and support at all times.

This thesis was partially supported by the South African Council for Scientific and Industrial Research; additional financial support was provided by the Computer Science Department of the University of Cape Town. Such support is gratefully acknowledged.

TABLE OF CONTENTS

Chapter 1: INTRODUCTION

The process of developing a programming environment for a new programming language is an expensive one, both in terms of funding and of human resources.

As a result, a large amount of research effort has been expended on the construction of metatools such as compiler generators, in order to reduce these overheads. The earliest of these such as YACC [Johnson 75] or the XPL project [McKeeman 70] were essentially parser generators which provided 'hooks' which could be linked to code for performing static semantic evaluation or code generation. This code would need to be written by hand in the target language of the generator.

More latterly, compiler compilers which can take as input a description of the semantics of a programming language as well as a description of the target machine and produce object code generation modules directly, such as the PQCC project [Cattell 78, Wulf et al 80], and the work of Graham [Graham 80]. Mosses developed a compiler front–end and interpreter driven by denotational semantics [Mosses 82]. This system did not perform any context condition checking.

The issue of automation of context condition (or static semantic) checking has also been extensively addressed, albeit less successfully, using techniques such as attribute grammars and W–grammars. The working systems which have been constructed (and which are referenced below) have tended to restrict themselves to 'toy' languages, and the specification formalisms which such systems have required have proven to be extremely difficult to use and understand, thus detracting from their utility.

This lack of success in dealing with the problems associated with automation of context condition checking is in part because of the fact that this is the most difficult part of a programming language: The introduction to [Bjorner & Oest 80] states:

> "The problem of modeling the so–called static semantics of (languages like Euclid, CHILL and) Ada[1] is a difficult, non trivial one. We have come to believe that this problem exceeds, by a manpower estimate alone, 3–4 times that of modeling the dynamic semantics of the sequential parts of (these languages and) Ada."

(We choose to refer to "static semantics" as "context conditions", following the approach of [Rosselet 84], because we feel that the latter phrase captures more accurately the sort

[1] Ada is a registered trademark of the United States Government, Ada Joint Program Office.

of static checks required on a program).

Why should this modeling of the context conditions of a programming language be such a difficult task? There are several reasons for this.

- Almost all languages have a formal definition of their syntax. The semantics of the languages (context conditions and dynamic semantics) however, tend to be informally specified. This means that the definition of the context conditions can be inconsistent and incomplete, thus making the task of the language implementor more difficult as he not only has to build a formal definition, but also has to resolve all the problems with the informal specification first. The sheer size of many informal definitions adds considerably to this problem.
- There is no formalism which is generally acceptable to the compiler construction community as the way to define the context conditions of a programming language. In recent years, attribute grammars have become the most generally used specification formalism, but there are a number of problems with their use which shall be discussed below.
- The nature of the context conditions of a programming language add considerably to the difficulty of the language implementor. When designing a programming language, all those aspects of the syntax which would be inconvenient to implement, as well as any other arbitrary conditions that the language designers desire, tend to be placed as constraints on the syntax. Thus, context conditions can be naively thought of as being the (perfectly acceptable) requirements that all identifiers must be declared before use, and that operands in an expression must be type compatible, and that parameters or subscripts must match in respect of number and type in a procedure call or array reference. However, they also include other arbitrary conditions. For example, just some of the conditions on record aggregates in Ada include: [Ada 83]

  "Named associations can be given in any order (except for the choice OTHERS), but if both positional and named associations are used in the same aggregate, positional associations must occur first, at their normal position. Hence once a named association is used, the rest of the aggregate must only use named associations. Aggregates containing a single component association must always be given in named notation [section 4.3, para 4]. If the type of the aggregate is a record type, the component names of choices must denote components (including discriminants) of the record type. If the choice OTHERS is given in a record aggregate, then it must

represent at least one component. A component association with the choice OTHERS or with more than one choice is only allowed if the represented components are all of the same type. The expression of a component association must have the type of the associated record components. The value for a discriminant that governs a variant part must be given by a static expression[section 4.3.1, paras 1,2]"

This is not to say that such conditions are unnecessary or meaningless; they are neither. But their ad hoc nature makes them difficult enough to implement, let alone formally specify for input to a compiler generator.

There have been a number of attempts to automate context condition checking; most have used attribute grammars, one that we know of has used affix grammars, and the remainder have used some form of programming language. These are discussed in the chapter "Related Work" (chapter 21) below.

This thesis forms part of an ongoing project at the University of Cape Town to evolve a complete language independent programming environment, which will eventually include a full range of tools, including compilers and editors, and the metatools to generate these. The construction of such an environment will facilitate research into programming languages, as researchers will be able rapidly to prototype compilers and editors for new languages, and thereby allow for the testing of new language concepts as quickly as possible.

The focus of this thesis is the specification of the context conditions of programming languages to such metatools. None of the existing specification formalisms seemed ideal for this task.

An ad hoc approach (using hooks from a generator) would require the rewriting of the context condition checking code each time an environment is produced for a new language.

The specification formalism for context conditions should decorate a skeleton describing the syntax of a language. This simplifies the task of constructing a language specification. For this reason two level approaches such as W-grammars are not suitable. Attribute grammars give a formalism where the description of the syntax of a language is decorated by context conditions, but this approach is deficient in several ways.

A major problem with attribute grammar specifications is that they are very large and clumsy, as many of the rules of the grammar contain redundant information needed to describe the inheriting and synthesizing paths taken by the attributes. Watt's 1979

[Watt 79] paper on extended attribute grammars is an excellent example of this. As the language specifier builds a language specification the task becomes more and more difficult as the maze of redundant attribute passing grows and the specification becomes larger.

Another problem with some implementations of the attribute grammar formalism is that the attribute grammar specification of a language often does not specify anything other than the attribute propagation paths through a tree. When a conditional check of some form is required then it is performed by calling an auxiliary function which will then perform the check. This function must then be written in some auxiliary language, usually the target language of the compiler generator. All meaningful work actually is performed by these functions, including symbol table manipulations.

A major reason for using attribute grammars as a specification formalism is that they provide theoretical guarantees providing that the grammar has certain properties. For example, if attributes are noncircular then evaluations will terminate or if certain ordering rules between attributes hold then an upper limit on the evaluation time can be determined [Kastens 80, Reps 84]. If, however, the actual evaluations are performed by auxiliary functions written elsewhere then this theoretical advantage is in fact lost as no guarantees can be made of how the auxiliary functions perform.

Rather than use a specification formalism which suffers from the above shortcomings, we wished to design a formalism which would simplify the task of specifying context conditions as much as possible by providing a mechanism which would be simultaneously powerful, easy to use and support a natural view of the context condition specification process. The formalism should also lend itself to a formal approach to context condition development and specification.

An immediate result of developing such a formalism is that the development of programming languages would be simplified as a simple–to–use specification tool would mean fast prototyping of compilers. This would mean that a language designer would be able to test his design whilst involved in the design process without needing to distract himself unduly from the design task, which should have the benefit of more consistent and precise language specifications being developed.

The formalism developed in this thesis is intended for use with languages such as Modula–2 [Wirth 83], Pascal [Addyman 80, Jensen & Wirth 74], CLU [Liskov et al 81] and Ada. Languages from outside this paradigm such as LISP [McCarthy & Levin 65] tend to have primitive context conditions, or none at all, as the LISP equivalents of

context conditions are checked at run time.

The formalism developed in this thesis has been given a formal definition. As will become clear in the course of the thesis, context condition specifications, especially type evaluation routines, tend to be recursive in nature. An approach is needed that allows easy manipulation of recursive structures which do not lend themselves to traditional fixpoint–type approaches to their verification.

A Temporal Logic based approach allows a proof methodology for dealing with context condition specifications to be evolved. The formal work in this thesis was therefore performed in a temporal logic framework. This framework gives a formal definition of the specification formalism, and allows formal proof of properties of context condition specifications. One of the disadvantages of the attribute grammar approach is that specifications can be shown to have a termination property, but not that the attribute evaluations produce meaningful results. The formal proof approach to context condition evaluations allows proof that context checks are totally correct, ie that they compute the correct result and that they terminate.

This thesis is divided into five parts. These parts reflect a flow in the thesis from the largely informal and explanatory to the formal.

The first part of the thesis introduces the new formalism for specification of context conditions of programming languages. A context free formalism is introduced. The approach to context sensitivity is discussed, and the specification formalism is then informally defined in the style of a programming language manual such as [Ada 83] or [Jensen & Wirth 74], where the syntax of the formalism is given along with an informal discussion of its semantics. This part is rounded off with a number of examples of the use of the formalism.

Throughout the thesis attempts have been made to show how the new specification formalism applies to 'real life' examples. Examples thus range from introductory expositions of small languages to problems of greater complexity, such as element–by–element assignment for record aggregates and operator overloading.

The second part of the thesis is a digression from the topic of context condition specification. This part presents an introduction to the Temporal Logic approach to program verification and extends the pre–existing work to include the verification of subprograms. This includes a new approach to the verification of recursive programs which brings such verifications in line with those used for iterative programs. This work is necessitated by the fact that context condition specifications can be recursive, and the

more traditional approaches to recursive program verification do not lend themselves to verifying specifications.

The third part of the thesis develops a formal definition of the specification formalism using Temporal Logic. Since specifications are to be verified using Temporal Logic, it is the logical choice as the formalism in which to build the formal definition. Once the formal definition is complete, it is used to verify a number of examples.

The fourth part of the thesis gives an overview of the implementation of a language–independent compiler front–end driven by the specification formalism developed in this thesis.

The fifth and final part contains a discussion of related work, the contributions of this thesis, and some conclusions and prospects for the future.

Four full–size programming language specifications using the context sensitive specification formalism developed in this thesis are also included as appendices.

Appendix I contains a description of the syntax of the formalism, together with the context conditions associated with that syntax in the formalism itself. In other words, Appendix I is a demonstration that the specification formalism is sufficiently powerful to describe itself.

Appendix II contains a description of Pascal in the formalism. Appendix III contains a description of Dijkstra's small language from [Dijkstra 76]. This language is particularly interesting because of its unusual scope rules. Finally, Appendix IV contains a description of another full–size programming language, CLU.

PART I – THE CFF/AML METALANGUAGE

This part describes CFF/AML, the metalanguage developed to specify the syntax and context conditions of a programming language.

CFF, the metalanguage for specification of the context free part of a programming language, is introduced first, in chapter 2. Chapter 3 motivates the approach to context condition specification adopted in this thesis, and chapter 4 introduces AML, the context condition specification language.

Chapters 5 through 9 contain several examples, ranging from the specification of simple languages such as Dijkstra's Language and ASPLE, to type evaluation on recursive types, handling record aggregates, and handling operator overloading.

Chapter 2: CFF – A CONTEXT FREE METALANGUAGE

This chapter describes the context free formalism in which the syntax skeleton of a programming language will be specified. The formalism described below was developed by McDermott [McDermott 84], and is designed to be used as a descriptive language for syntax. The major motivation behind the development a new context free language was the desire to be able to express syntax in such a way as to simplify the process of prompting a user in a language independent syntax–directed editor.

In such an editor, the user is guided by the description of the syntax as a program is entered. The syntax must therefore be described in a way which allows a highly user–friendly prompting process to be used. The syntax description formalism should specifically avoid recursion, replacing it instead by lists. This is because a prompting process which uses deep recursion can rapidly confuse a novice programmer, whereas lists, which are conceptually broad rather than deep are simpler for the user to comprehend [McDermott 84]. Options should also be presented in as simple a fashion as possible so as to minimise user confusion. We do not wish to get too deeply into this subject; interested readers are referred to [McDermott 84]. However, the reader should be sufficiently aware of the problems associated with prompting users in an editor to understand the need for the introduction of yet another variant of BNF.

Some variants of BNF exist which attempt to make ease the task of syntax specification in comparison to the effort needed to specify syntax in the original BNF. Such specification methods to not introduce any additional expressive power into BNF; they merely introduce new notational conveniences into the metalanguage. The Ada language definition, for example, uses a BNF variant which places terminals in a bold print and nonterminals in lowercase. No angle brackets are used and optional elements are allowed in a production. However, direct recursion is still allowed, whole chunks of a production may be made optional by enclosing them in square brackets, and lists take a very primitive form where, for example, a list of identifiers separated by commas is represented as

```
identifier {,identifier}
```

This is not suitable as it introduces a structure which is conceptually thought of as "a list of identifiers separated by commas" as "an identifier, then a list of commas and identifiers". We will introduce below a list specification formalism which supports the more conceptual view.

The BNF variant introduced by McDermott is called Context Free Form (CFF) to distinguish it from other variants. Its features may be summarized as follows.

- Nonterminals appear in lowercase and terminals in uppercase. This is a restriction in the language definition only – a user program has the cases that may be used defined in the lexical definition part of a language definition. The name of a production appears in double quotes on the line above the body of the rule.
- Any special symbols that are needed in the language, such as +,<,> etc are entered as is, except where they conflict with the set of metasymbols, in which case they appear in quotes. Quotes are treated using the lexical conventions of Pascal [Jensen & Wirth 74], i.e. a quote can be entered by quoting it.
- There is no need for direct recursion in the language definition as recursion may be replaced by lists. This is the preferable route to follow if language definitions are also to be used to prompt users of a language independent syntax–directed editor. A list has the form `<symbols_symbols>` where

  `<` Indicates the start of a list

  `_` Indicates the 'exit point' of a list. When the exit point of a list is reached, then either the list must be exited from, or all the symbols occurring between the exit point and the end of list indicator must appear, at which point a loop occurs back to the beginning of the list.

  `>` Indicates the end of the list. If this symbol is reached when working with a list (eg when parsing) then it acts as a command to loop back to the start of the list.

  symbols May be any symbols of the grammar being described such as terminals, nonterminals, another list structure, etc.

  Thus, a list of identifiers separated by commas can be rendered as

  ```
  <identifier_,>
  ```

  This list structure has two major advantages over the alternative structures used previously. Firstly, the structure itself is more flexible than the "{...}" list structure, eg as used in the variant of BNF used in Ada. Secondly, a side effect of using this specification method is that language specifications tend to be far smaller. This is significant for syntax–directed editor environments as a smaller language will appear less complex to the novice user.

- Nonterminals in a production may be made optional by prefixing them with a question mark. Lists may also be made optional in the same way.
- A production may have a number of alternate forms. These are separated by a bar character | .
- Certain nonterminals are predefined in the metalanguage. These are `identifier`, `integer`, `number` (real numbers) and `string`. The exact meaning of these nonterminals is fairly flexible and is defined in the lexical definition section of a language specification. This lexical definition section is not relevant to this thesis and is therefore not discussed here. Details can be found in [Kaplan 83a, Kaplan 83b].
- The layout of a production is significant; blanks and newlines are interpreted as prettyprinting control directives, ie wherever a newline appears in the production body, a newline will be inserted by the formatting utilities in the editor and the source code reconstructor. As a result, the special metasymbol % is used as an end of rule indicator.
- The symbol # is used as a margin indicator in the editor.
- The ! is used to separate two rules where a space would cause unwanted prettyprinting
- The : is used to control the degree of prompting in the editor.

Interested readers are referred to McDermott's paper describing a syntax–directed editor driven by CFF [McDermott 84].

EXAMPLES

Consider the following example of the specification of a rule in CFF:

```
"types"
TYPE <identifier = type;_
     > %
```

This is a definition of the syntax for types in Pascal. `types` is the name of the rule. TYPE is a reserved word. Once the list is entered, there must be an `identifier`, an equals sign and then a nonterminal `type` (defined elsewhere in the grammar definition) and finally a semicolon. Then if the list is repeated, in the editor a newline and five spaces will be entered by the editor before accepting the next type definition from the user of the editor. This facilitates prettyprinting. In the compiler, the input layout may be as defined by the user, but the source code reconstructor will prettyprint according to the

convention defined here.

The definition of the nonterminal type would then be

```
"type"
array|record|set|identifier|file|scalar|...%
```

where the ellipsis is used as all the options for a type do not concern us here. In these above cases the definitions do not make use of the full power of the list notation as nothing follows the exit point indicator in the list (except prettyprinting information). The advantages of the notation become more obvious when considering the definition of variables in CFF:

```
"variables"
VAR <<identifier_,> : type;_
    > %
```

The advantages of this notation are made even more apparent by considering the case of expressions:

```
"expression"
?sign<<factor_multop>_addop>%

"factor"
name|integer|number|...%
```

Here the more conventional expression–term–primary chain is replaced by a single production containing all the information of the standard recursive descriptions. The optional sign is a monadic leading operand. The expression itself consists of a list of factors, separated by multiplication operators, which in turn are defined in the multop nonterminal. This list is in turn only a part of a larger list, each entry of which is separated by an addition operator as defined in the addop rule. Thus the precedence of the operators is built into the syntax. The rule describes an entire simple expression (no relational operators in this version) on one conceptual level, simplifying the prompt process in an editor.

The list structure allows the construction of tree representations of programs which are conceptually broad rather than deep. This simplifies both the prompting process in an editor and the specification of context conditions, as one can move from one iteration of a list to another on the same level, as opposed to having to navigate a deep tree created by a recursive syntax.

Also, the mapping from the syntax diagrams of a language to CFF is made trivial by the list structure, which means that if a language has a syntax diagram description then the

CFF description can be extracted very easily.

Chapter 3: TOWARDS CONTEXT SENSITIVITY

Chapter 2 introduced a context free metalanguage. This chapter motivates the development of a new specification formalism for context conditions. This will not be a completely general language but with one which has the power to express the context conditions found in the definition of a programming language. These include:

- Checking for overdeclaration of identifiers in a scope.
- Checking that identifiers used in the statement part of a scope are all visible in that scope.
- Checking that the operands in an expression are type compatible.
- Checking that the types of the left and right hand sides of an assignment are assignment compatible.
- Checking that the formal parameters of a subprocedure match the actual parameters of a call of the subprocedure in respect both of number and type.
- Checking that an array reference has the correct number and type of subscripts.

We believe that when a language specifier is building a specification he tends to think of the operations required to effect some context check in a procedural fashion, thinking of the syntax skeleton of a language together with the syntax tree that would be constructed from such a skeleton.

The purpose of this project was the development of a specification formalism for context condition checking which would be powerful, easy to use and also support the paradigms with which a language specifier is most likely to be familiar. All persons likely to wish to use this specification tool will be familiar with programming, and it is in the nature of the problem of context condition checking to specify the checks in a procedural fashion. For this reason the specification language is procedural in nature.

Despite the earlier criticisms of attribute grammars, the concept of enhancing a syntax skeleton with the required context checks remains an attractive one, so an approach has been adopted in which the syntax skeleton of a programming language is decorated in the relevant places with procedural code written in a special high–level language, the execution of which achieves the required context condition checks.

As discussed in Chapter 1, the context conditions of a programming language tend to be ad hoc in nature, and to be specified in a natural language, which results in a specification which is often inconsistent, incomplete and ambiguous [Pagan 81].

The ad hoc nature of the context checks of a language requires that the specification language must be highly flexible; on the other hand, it must be usable and simplify the language specifiers task of translation from the informal and ambiguous to the formal, otherwise the entire purpose behind the development of a new specification formalism will be lost.

The first phase in constructing a language specification is the construction of a CFF syntax skeleton, usually from the syntax diagrams of a programming language.

Once this phase is complete, the context conditions may be layered on to the syntax. When performing this layering process, the language specifier tends to think of both the grammar and an idealised tree constructed from the grammar, ie one where there is an exact mapping from the constructs of the grammar to the tree. In practice, of course, not all of the elements of this idealised tree need be held; nonetheless, the formalism should support this illusion.

When considering, for example, checking that identifiers are declared before use in a statement, the required operation is thought of as 'go to the place where identifiers are declared and see if this identifier is there. If not, repeat the operation in all visible scopes until the identifier is found or the search of the identifiers in the most global scope fails',

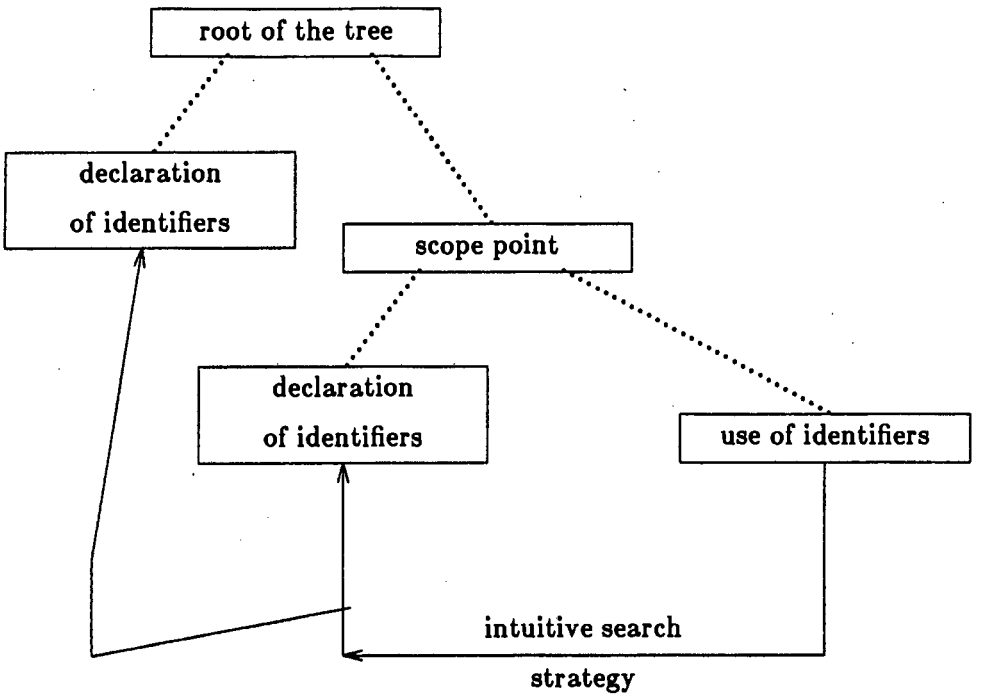


Figure 1

which can be diagrammatically represented as in figure 1. With attribute specifications, however, the language specifier is constrained to think explicitly in terms of a flow only through paths defined by the nodes of the tree, ie, from statements up to the first common ancestor of both the statements and the declarations; then down to the declarations; then search; then, if necessary, go up to the next set of declarations, again via an ancestor node, and so forth. In practice this search strategy may be smoothed by the implementation to conform more closely to the intuitive strategy described above. This is cold comfort to the language specifier who is forced to proceed in the more rigid way, increasing the overhead of developing a specification.

This insight forms the conceptual foundation of the new formalism. The principle issue is to find an effective way of supporting the intuitive search strategy described above. Once the solution to this problem was found, the construction of the remainder of the language was largely pre–ordained.

The first obvious solution, which was adopted in the DIANA intermediate form developed for Ada [Evans et al 82], was to associate with each identifier in the tree a pointer to its declaration point. In a batch–only environment, this solution would have been fine; however, the formalism is intended for application in other utilities, such as intelligent

editors, as well. If each identifier is associated with an explicit pointer, what would happen if the identifier's declaration was deleted for some reason? Or worse, what if another identifier with the same name and another type is declared in an intermediate scope between the declaration of the identifier and an instance of its use, thus changing the type of the identifier at the instance?

An explicit binding between the use of an identifier and its declaration is therefore impractical. Instead, a dynamic binding is needed, one which could, for example, find the most local declaration of a particular identifier only when required to.

Further, the information concerning the type attributes should not be held explicitly in a symbol table, as this means that each time an edit is made, both the parse tree and the symbol table may need modification, which is an unnecessary overhead.

In [Barrett & Couch 79], the authors state

> The compiler data structure that associates identifiers with their attributes is called a SYMBOL TABLE. Often the identifiers are carried in a separate table called a NAME TABLE, with the attributes in a separate table called an ATTRIBUTE TABLE.

All the information needed for a type evaluation is contained on the parse tree. It is therefore not necessary to maintain an attribute table explicitly, which is in line with our argument against symbol tables given above.

A simple modification of the name table concept was adopted, where each entry in the table has associated with it a POINTER to the place on the tree where evaluation of the type attributes of the entry may begin. The attribute table thus becomes an implicit part of the parse tree.

The name table concept is extended to allow multiple name tables at each scope level. Each such table has a user (language specifier) defined name, and the language specifier may determine how many tables are available at each scope level.

Once the identifiers of each scope level are contained in some sort of name table, a simple lookup in the table can be performed to ascertain whether or not the identifier is declared, and on which scope level. When the identifier has been found, the pointer associated with that entry in the table can be used to find the point on the tree where the identifier is declared and start type evaluation if required. This pointer then acts as a 'gateway' onto the tree for type evaluation purposes, and the intuitive strategy then becomes as

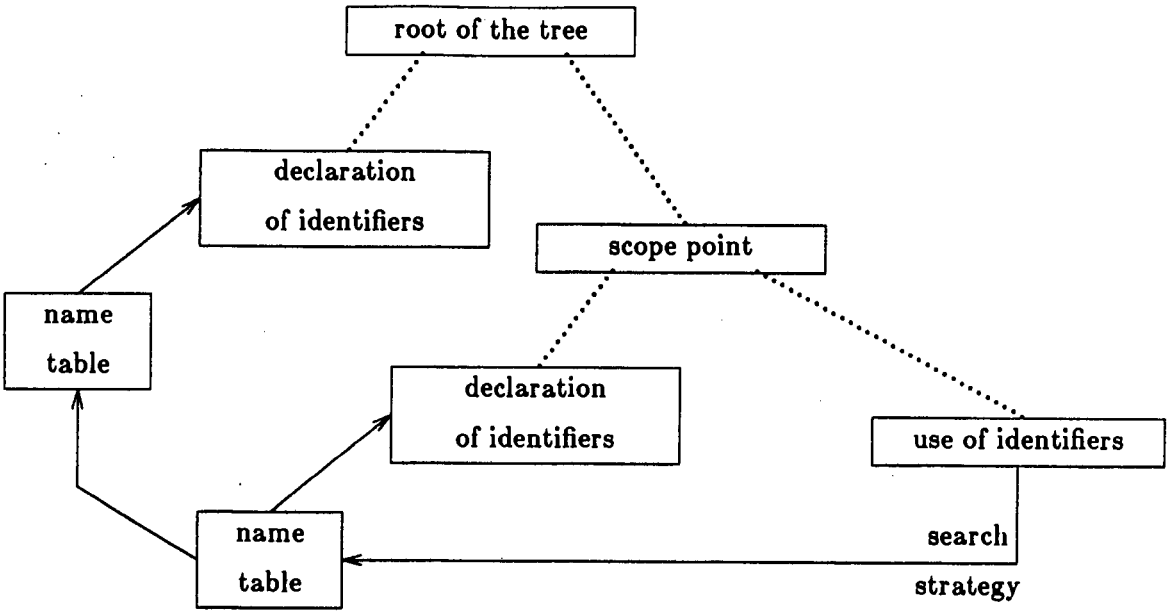


Figure 2

represented in figure 2.

The conceptual 'go to the place on the tree where the identifiers are declared' becomes a matter of name table lookup. This lookup can automatically take care of finding the identifier in the correct scope and finding the gateway point on the tree.

With the above in mind we can introduce an example of a CFF rule with context conditions added. The rule format is as before, except that decorations are added that describe the context conditions to be checked at that point. The conditions appear between the metasymbols [and] . Because a language specifier could conceive of these specifications being interpreted on an 'abstract machine' buried inside a utility of a system using this specification methodology, the language in which the checks are written is called AML (Abstract Machine Language).

```
"types"
TYPE <identifier[ IF unique(vars + types + consts +
                            subroutines)
                  THEN
                    insert(types)
                  ELSE
                    error('Identifier already declared')
                ] = type;_
     > %
```

Immediately some observations can be made about the AML. The AML language is a

Pascal-type language. The control structures are similar to Pascal. `Vars`, `types`, `consts` and `subroutines` are all user (language specifier) defined names of name tables. The `unique` function is a standard boolean function in AML which accepts as argument a list of name table names and checks whether or not the `identifier` is already entered into one of them, in the current scope. If the true value is returned then the result of the `insert` operation is to insert the `identifier` into the name table called `types`. If the false value is returned then an error message is generated.

If the `insert` operation is performed then the pointer part of the name table entry points to the declaration of the `identifier`.

The AML code is called a decoration of the CFF rule; the combined specification is called a CFF/AML specification of the rule.

For the rule `type`, only the `identifier` option needs a check:

```
"type"
identifier[ IF not visible(types)
               THEN
                  error('Not a type identifier')
          ]|array|record|set|file|scalar|...%
```

Here the only check required is that the `identifier` be a type identifier. If it is a type identifier, then it will have been placed in the name table called `types` by the AML code decorating the `types` rule. This will cause the `visible` function to return a true value.

One of the major advantages of this formalism also becomes apparent as a result of this example. There is no need to inherit or synthesize information from level to level of the tree in an explicit fashion as with attribute grammars. This means that language specifications remain small and are also easier to read as they are less cluttered.

Further, the specification method is similar to a programming language. Because of the high-level name table operations built into the AML, the task of writing AML code is significantly less than that required to write the equivalent code in a conventional programming language.

The above examples merely give a flavour of a CFF/AML specification; in the following chapter describes the AML language in some detail before developing some CFF/AML specifications of example languages.

Chapter 4: OVERVIEW OF AML

This chapter introduces informally the features of AML, following a language manual type approach such as [Ada 83] or [Jensen & Wirth 74], with syntax descriptions and informal discussion of the semantics of the language. A collected syntax of CFF/AML, together with the context–sensitive aspects of the language defined in CFF/AML, is to be found in appendix I. The dynamic semantics of the CFF/AML metalanguage are presented informally in this chapter, and formally defined in chapters 14 and 15 below.

When defining context check evaluations, it often becomes necessary to refer to the tree which would be a representation of a program on which actual checks would be carried out. (For example, the `son` command, which moves to the given son of the current node, or the `subtree` function, which returns true if a given subtree is present on the tree). In order that these references can be kept at an abstract level, AML considers the CFF rules as templates of an idealised tree and then allows references to the elements of the tree via the CFF rule names. It is assumed that all possible elements of the tree are present, although in practice a large number of these may be missing. For example, in the UCT implementation terminal symbols other than identifier names and operators are not actually stored on the tree as these can be extracted from the CFF rule templates when needed. However if elements are not present then the implementation must provide a correct 'virtual' view of the actual tree held.

A complete context check of a user program takes place as a number of passes through the virtual tree in a depth–first, left–to–right manner. The number of passes is under AML control, the default being one pass. As the tree is walked through, the CFF rule templates from which the tree was constructed are walked through as well. When a point is reached where an AML decoration is included in the rule, then the tree walker halts and the AML code inside the decoration is executed. When the execution is complete then the tree walking process continues.

The language specifier has access to two sets of 'tree pointers' which indicate positions on the tree. The first are the TREE WALK pointers which can be accessed by the AML code but not changed. These pointers can be used to discover the state of the tree walk, ie the location on the tree.

The second set of pointers are the AML pointers which can be accessed and modified by the AML code. The reason for the second set of pointers is that the TREE WALK pointers navigate through the tree in a fixed fashion as described above. When performing context checks, however, there is often a need to jump around the tree in some

(ostensibly random) fashion. The AML pointers can be used to move around the tree in this way. The usual scenario here is that, when evaluating a type which is in turn made up from some user defined type, it may be necessary to move to the point on the tree where the user defined type is declared in order to continue the type evaluation.

These two different sets of pointers may obviously be pointing to different points in the tree at the same time. Usually when an evaluation starts it starts from the tree walk pointers but it may switch at some time to the AML pointers. This is because the AML pointers are affected by name table lookups. One may then start from an identifier on the tree, (pointed to by the tree walk pointers), do a name table lookup to see if it is declared – which alters the AML pointers to point to the place pointed to by the pointer part of the name table entry for the identifier – and then continue evaluating the type of the identifier using the AML pointers. To support this there is an invisible SWITCH in the abstract machine, which indicates which of the tree walk or AML pointers is the set being used. On starting the execution of any AML decoration, this switch is set to the tree walk pointers, but certain operations, such as name table lookups, alter it to the AML pointers. The effect of certain AML operations depends on the value of the switch, although others operate entirely independently of the switch.

Each of the sets of pointers has a "current node", which is the node pointed to by that set of pointers. Each node on the tree is constructed from a particular rule of the CFF grammar, so "current rule" means the rule which was used as a template for the construction of that node. Also, each node on the tree is in a particular scope, and "current scope" means the scope in which the current node is defined.

The current node, rule or scope can be referred to relative to the AML or tree walk pointers.

When making reference to current node in the context of an AML command, some commands can use both pointers, in which case which of the the pointers is used depends on the value of the switch; others use either one or the other exclusively.

A scope area may be defined to start and end anywhere on the tree. As the pointers move, they may move in and out of the scopes on the tree.

The preceding chapter introduced the idea of an abstract name table structure with a number of high–level operations defined on it. In fact there are three permissible name table structures. These are called SET, LIST and STACK. For the SET name table, no assumptions are made about the internal structure of the name table. The AML implementor may then choose any structure that will give a most efficient implementation, for

example hashing tables or binary trees. For the LIST name table, entries are assumed to be in FIFO order and for the STACK name table, in LIFO order.

A number of generic operations are defined upon these structures. When a key is required, it can be supplied in one of two ways. The first is to supply the key explicitly as an argument to the operation. The second is not to supply it explicitly but instead to perform the operation when the pointer designated by the switch is pointing to a leaf of the tree. This leaf is then used as the key. This second way is the more common way of providing a key for a name table operation and is the way used in the examples above.

Some of the AML operations on name tables are treated as statements in the AML; the rest are treated as AML standard functions. Each of the operations is introduced at a conceptual level here; their syntax and an informal description of their semantics are given below in this chapter. A collected syntax of CFF/AML, with context conditions defined in AML, appears as Appendix I. Formal definitions of the semantics are given in chapters 14 and 15. The operations are:

UNIQUE takes as argument a list of name table structures and returns true if the key is not an element of one of the name tables specified as an argument. The most local instantiation of each table is searched.

LOCAL takes as argument a list of name table structures and returns true if the key is an element of one of the tables specified as an argument. The current local scope is the only scope used.

VISIBLE takes as argument a list of name table structures and returns true if the key is an element of one of the tables specified as an argument. All visible elements of all visible tables are searched.

INSERT takes as argument a single name table structure name, inserts the given key into the table, and associates with the key a pointer to the tree.

DELETE takes as argument a single name table structure name and deletes the most local entry contained therein with the given key.

CLEAR takes as argument a single name table structure name and removes all the elements in it.

TOP takes as argument the name of a single AML name table STACK structure. TOP returns true if the key is the same as the top element of the stack.

UNSTACK takes as argument the name of a stack type name table structure. The effect of the command is to remove the top element of this stack.

FRONT takes as argument a single name table LIST structure name and moves the AML pointers to the front element of the list.

NEXT takes as argument a single name table LIST structure name and moves the AML pointers to the next element of the list.

EOL takes as argument a single name table LIST structure name and returns true if the AML pointers point to the last element in the list.

COPY is used to move the contents of name tables from one scope to another. For example, on a WITH statement in Pascal, the local variables from the record declaration must be copied into the local scope defined by the WITH statement to reflect the scope mutation that the WITH introduces.

EMPTY returns true if the name table given as argument is empty. It is useful for detecting that there are no forward–declared procedures without bodies, or unreferenced labels, provided that auxiliary name tables are instantiated to record this information.

DUMP takes as argument a name table and prints out the contents, in a legible form.

The LOCAL, VISIBLE, TOP, FRONT and NEXT operations all have the effect of setting the switch to the AML pointer value.

As many name tables structures of the various types as required may be declared by the language specifier. Such name table instantiations appear in the AML decorations immediately after the declaration of a new scope. They are nested dynamically. A more local instantiation of a name table does not hide the name tables of the same name declared at more global scopes; rather, an entry in a more local name table hides any entries with the same name in more global instantiations of the name table. Scoping of user program variable names can thus be taken care of automatically.

For example, suppose there exists a name table called `symtab` containing the entries `x`, `y` and `z` at a particular scope, and declare a table with the same name at a nested scope. Then, inserting an entry `y` in the nested table will hide the entry `y` in the more global table whilst leaving the entries `x` and `z` visible. This reflects the way that scoping rules are expected to hide variable names.

In addition to the name tables, the language specifier user may declare simple AML variables. These may be of `integer`, `string` or `flag` (boolean) type. AML simple variables are static in the FORTRAN sense, being declared on a global level only in a declarations block which precedes the first rule of the grammar.

AML has one predeclared constant with the name `std-error`. This constant is used as the result of the type evaluation process when the process fails for some reason, such as attempting to perform a type evaluation which uses a type which does not exist for some reason, such as the user's failure to declare it. There are two possible values for `std-error`. The first is to make std-error compatible with nothing (a zero value) and the second to make it compatible with everything (an identity value). The latter approach has been adopted as it results in fewer spurious errors being generated when performing type compatibility checking.

AML also has provision for the declaration of procedures and functions. The functions may return any simple AML type. Procedures and functions may be recursive. In order to simplify the implementation, they may not take parameters, as parameters introduce significant extra complexity into the implementation of procedure and function call. This restriction will be lifted in later versions of CFF/AML.

In a multi-pass CFF/AML definition, the decorations are prefixed by a pass indicator. Any number of these may be declared in a PASS clause along with the declarations of AML simple variables. The order of the indicators is significant. For example, if there is a pass declaration

```
PASS setup, check, redo
```

then each AML decoration must be prefixed with one of `setup`, `check` or `redo`. All the decorations prefixed with `setup` will be executed in the first pass, those prefixed by `check` in the second, and so on. If a one-pass specification is desired (as is usually the case) then the pass indicators may be entirely omitted.

AML is designed around its concept of a name table. The statements of the language reflect this and the special target problem domain for AML. They are split into four categories for convenience. These are conventional, scope, tree and stack statements. The syntax of the AML static declarations is described first, followed by descriptions of each of the statement categories.

AML STATIC DECLARATIONS

AML simple variables are declared in a special header section which precedes the first rule of the CFF grammar. All `integer`, `flag` and `string` variables, as well as any pass indicators, are declared in this section. The syntax is:

```
"declarations-section"
```

```
'[' DECLARE
    ?passname-declaration
    <declaration;_> ']' %

"passname-declaration"
PASS <identifier_,> ; %
```

The set of identifiers following the reserved word PASS are called pass indicators. The elements of this set must be disjoint. However, they need not be unique with respect to the AML static variables, procedures, functions or the names of the AML name tables.

If a passname-declaration appears in a CFF/AML language specification then all AML decorations must begin with a pass indicator. The order of the pass indicators is significant. If a passname-declaration is present in a CFF/AML language specification, then the context condition checking will be performed as a number of passes. The number of passes is equal to the number of declared pass indicators. On the first pass, all AML decorations prefixed with the first pass indicator are executed, on the second all AML decorations prefixed with the second pass indicator are executed, and so on for each pass indicator in the list.

```
"declaration"
 string-declaration
|integer-declaration
|flag declaration %

"string-declaration"
STRING <identifier_,> %

"integer-declaration"
INTEGER <identifier_,> %

"flag-declaration"
FLAG <identifier_,> %
```

All identifiers in the declarations section must be unique, with respect to other static variable declarations. It is not necessary that they be unique with respect to the pass indicators or AML procedure, function or name table names. Also, all AML variables are entirely independent of CFF rule names.

The identifiers declared in a declaration are called AML simple identifiers, to distinguish them from AML name tables. Each identifier takes on a type defined by the reserved word which precedes the list of which it is a member. There is no compatibility across types in AML.

Procedures and functions in AML can take no parameters, but can be recursive. Functions can return any AML simple type. Functions are assigned a return value using the AML assignment statement. No function or procedure declaration nesting is possible. These restrictions were introduced to simplify the implementation of the AML interpreter. Because AML code is interpreted, the need to be able to optimize the implementation as much as possible was strongly felt. Therefore many language features which have a large overhead in their implementation, such as parameters and nested scopes, were omitted from AML.

The syntax for the AML subprogram declarations is:

```
"aml-subprograms-section"
  '[' <subprogram-declaration;_> ']'

"subprogram-declaration"
 procedure
|function %

"procedure"
PROCEDURE identifier ; block %

"function"
FUNCTION identifier RETURN type ; block %

"type"
 STRING
|FLAG
|INTEGER %
```

No two procedures or functions may be denoted by the same identifier. All procedures and functions are mutually visible. A block is a list of statements and is defined below.

AML CONVENTIONAL STATEMENTS

The conventional statements include the control abstraction mechanisms of the language, similar to that of Pascal, such as if, while, repeat and procedure call statements. Then there are a number of additional statements which are included in the category of conventional statements largely because they do not fit in elsewhere. The full set of AML conventional statements is:

```
    assign | block | error | exit | if | procedure-call | null
  | repeat | reset | restore | save | set | warning | while %
```

The `assign`, `block`, `if`, `procedure-call`, `null`, `repeat` and `while` statements are

almost identical to those of Pascal and are consequently not discussed here. They are included in the definition of CFF/AML to be found in Appendix I.

Consider now the remaining AML conventional statements:

```
"exit"
EXIT %
```

The EXIT statement forces a return from an AML procedure or function. If executed when not in a procedure or function body, the effect is to halt the execution of the AML decoration currently being executed.

```
"error"
ERROR(string) %

"warning"
WARNING(string) %
```

The ERROR and WARNING statements are used to indicate to the user of the CFF/AML system that an error or warning exists. Both take as argument a string which is output as the error or warning message.

```
"set"
SET(identifier) %

"reset"
RESET(identifier) %
```

Rather than assigning the scalar values 'true' and 'false' to boolean variables, the AML simple flag variables are set and reset via the SET and RESET commands, each of which take as argument the name of one flag variable.

```
"save"
SAVE(identifier) %

"restore"
RESTORE(identifier) %
```

A programming language definition such as CFF is inherently (although not necessarily directly) recursive. As each level of recursion is entered, the AML simple variable values often need to be saved and restored after the recursion level terminates. In a conventional programming language, this is taken care of automatically by the operation of the stack at run time. The equivalent in AML would have been to allow simple variable declarations on any rule. This would have led to language definitions becoming excessively cluttered. The problem was therefore attacked from a different angle and the SAVE

and RESTORE statements were introduced into AML. A SAVE takes as argument a simple AML identifier and saves its value, whilst a RESTORE restores the value of the argument identifier to the last value saved. As the previous sentence implies, SAVEs and RESTOREs may be nested.

AML SCOPE STATEMENTS

AML must provide a flexible method for controlling scopes in the definition of a programming language, as not all languages use the same conventions or start a scope in the equivalent place. A series of scope statements allow scopes to be started and ended, and allow for the declaration of the name tables for the scope.

```
"scope-statements"
  start-scope | finish-scope | dump | acomp-table-load
| dcomp-table-load | mcomp-table-load | insert | delete
| copy | clear | unstack | front | next | aml-scope
| tree-scope %
```

Scopes are started by executing a start-scope statement.

```
"start-scope"
NEWSCOPE
?<symbol-table-declaration;_> %

"symbol-table-declaration"
 set-declaration
|stack-declaration
|list-declaration %

"set-declaration"
SET <identifier_,> %

"stack-declaration"
STACK <identifier_,> %

"list-declaration"
LIST <identifier_,> %
```

The effect of the NEWSCOPE is to introduce a new scope level. The declarations which follow the NEWSCOPE instantiate local name tables for the new local scope level, with type from the reserved word preceding the lists of tables. All name table names must be disjoint on a particular scope, but can duplicate names which appear at another scope. If a table is of a given type at one scope, it may be instantiated with another type at another scope.

Note that a programming language must have at least one scope level if name table manipulations are to be used.

```
"finish-scope"
ENDSCOPE %
```

The effect of this statement is to close off the current scope, making it and all the tables contained in it invisible, and to return to the scope into which the current scope was nested by the start-scope statement.

The information in a closed scope is not lost completely. If the AML pointers later walk onto a part of the tree which contains a closed scope, then that scope is reopened with respect to the AML pointers.

```
"load-acomp-table"
LOAD-ACOMP-TABLE(expression, expression) %

"load-dcomp-table"
LOAD-DCOMP-TABLE(expression, expression, expression, expression) %

"load-mcomp-table"
LOAD-MCOMP-TABLE(expression, expression, expression) %
```

Each scope level has associated with it a set of local compatibility tables for dyadic, monadic and assignment compatibility checking. All type compatibility checking is performed relative to these, and more global visible tables. Local compatibility tables (as opposed to just having a set of global pre-supplied tables) are necessary in a language such as Ada, where local overloadings of operators are allowed. Entries in a local table are visible in the scope of the table and all enclosed scopes, unless hidden in the same way that entries in the name tables may be hidden.

Type compatibility tables for standard dyadic and monadic operations, as well as assignment compatibility checking must be pre-supplied by the language specifier. If, as a walk through a walk through a program tree proceeds, overloaded instances of operator definitions are encountered, then these may be loaded into local instances of the compatibility tables. When a compatibility check is required then the tables are searched from the most local out to the standard pre-supplied tables.

Each of the expressions must be of STRING type. The way in which these type compatibility tables are used is discussed in the stack statements section below, where type compatibility checking issues are discussed.

```
"insert"
```

```
INSERT(?stringpart identifier) %

"stringpart"
expression, %
```

The `insert` statement operates on AML name tables and is used to place items in name tables of all types. The identifier must be the name of an AML name table. If present, the expression in the stringpart must have type STRING, and is used as the key for insertion. Otherwise, the leaf pointed to by the pointers is used as the key. The place where the key is inserted in the table depends on the table type, thus for SETs insertion is anywhere in the table, for LISTs at the end of the list (a FIFO order is maintained) and for STACKs at the top of the stack (a LIFO order is maintained).

Each entry in a table has associated with it a pointer to the tree. This pointer can be made to point to the place on the tree pointed to by the tree walk or AML pointers (whichever is relevant) at the time of insertion, or some other arbitrary point on the tree. Where these pointers point depends on the size of a stack known as the MARK stack. This stack is operated on by the MARK and RELEASE statements defined below. If the MARK stack is empty then when an INSERT takes place the pointer associated with the INSERTed key points to the place on the tree pointed to by the (AML or tree walk) pointers. If the stack is not empty then the pointers associated with the key point to the place on the tree pointed to by the top element of the MARK stack.

```
"delete"
DELETE(?stringpart identifier) %
```

The effect of the `delete` statement is to remove the most local occurrence of the given key from the given name table. If the `stringpart` is present then it must be of STRING type (and names the key to be removed). If the `stringpart` is absent then the leaf pointed to by the pointers indicates the key to be removed. The identifier must be the name of a name table.

```
"copy"
COPY(identifier) %
```

The `copy` statement adds the contents of the given name table from the scope of the AML pointers to the instantiation with the same name in the scope of the tree walk pointers. It is particularly useful when dealing with statements such as the pascal WITH statement which mutates scopes.

```
"clear"
CLEAR(identifier) %
```

The identifier in the clear statement must denote a name table. The effect of the CLEAR is dependent on whether or not there is a local instantiation of the table. If there is a local instantiation, then all those elements in the most local instantiation are removed; otherwise all elements of all visible instantiations are removed.

```
"unstack"
UNSTACK(identifier) %
```

The identifier must denote a name table of STACK type. The effect of the statement is to remove from the name table the element on the top of the stack.

Note that because of the dynamic scoping nature of name tables in AML, it is not possible to check at compile time whether or not a name table is of the correct type. Thus name table typechecking is done when the AML decorations are executed.

```
"front"
FRONT(identifier) %
```

The identifier must denote a name table of LIST type. The effect of the statement is to move to the front of the given list. This is achieved by moving an internal pointer to point to the front element of the list. The AML pointers are also modified to point to the place on the tree pointed to by the pointer associated with the front element of the list.

The FRONT operation also causes the invisible switch to indicate the AML pointers.

```
"next"
NEXT(identifier) %
```

The identifier must denote a name table of LIST type. The effect of the statement is to move to the next element of the given list. This is achieved by moving an internal pointer to the next element of the list. The AML pointers are also modified to point to the place on the tree pointed to by the next element of the list. A NEXT on a list which has no more elements is an error.

The NEXT operation also causes the invisible switch to indicate the AML pointers.

An AML standard function, EOL is used to detect whether or not there is another element in a list for use by a NEXT instruction.

```
"aml-scope"
AMLSCOPE %

"tree-scope"
TREESCOPE %
```

The invisible switch always starts off at the beginning of an AML decoration indicating the tree walk pointers. This switch is modified by certain name table operations, namely

LOCAL, VISIBLE, TOP, FRONT and NEXT. The SON and FATHER instructions defined below can also modify the switch. Sometimes the switch can end up indicating the wrong sets of pointers, or sometimes a decoration must start by using the AML pointers. This can be remedied by using the AMLSCOPE and TREESCOPE instructions, which manually override the switch value. AMLSCOPE forces the switch to indicate the AML pointers, whilst TREESCOPE forces the switch to indicate the tree walk pointers.

```
"dump"
DUMP(identifier) %
```

When developing a language specification, it is often useful to be able to keep track of what is in a table by printing the contents of the name table. There are also occasions where, once a specification is being used in a 'live' environment, that one wishes to inform the user of the contents of a name table. For example, all forward declared procedures in Pascal can be held in a name table and, as their corresponding bodies are declared, deleted from the table. If, at the end of the program, there are still entries in the table, then there are forward declared procedures without corresponding bodies. The contents of the table should then be printed to inform the programmer which procedures are in error.

The DUMP command serves to assist under these circumstances. The identifier must denote a name table and the effect of the statement is to print out, in human-readable form, the contents of the name table.

AML TREE STATEMENTS

Features are provided in AML to move explicitly on the virtual tree constructed from the CFF rules. The language specifier can work at an abstract level where all elements of the tree are assumed to be present. The implementation of AML must take care of the mapping to the actual tree held. The statements grouped together as tree statements are:

```
"tree-statements"
 saveposition | restoreposition | son | father
| intolist | nextiteration | mark | release %

"saveposition"
SAVEPOSITION %

"restoreposition"
RESTOREPOSITION %
```

In the same way that SAVE and RESTORE allowed the saving of values if AML simple

variables to facilitate the implicit recursion of a language definition, there are commands that allow similar saves and restores of the pointers used to indicate the position on the tree. These commands are SAVEPOSITION and RESTOREPOSITION. Their effect is analogous to that of SAVE and RESTORE except it is the AML pointers that are saved, not variables. The TREE WALK pointers cannot be saved as their values can be used and not modified, so it would not be meaningful to do so.

The environment of the scope in which the pointers are to be found, such as positions within lists in that scope, are saved along with the pointers.

```
"son"
SON(identifier) | SON %

"father"
FATHER(identifier) | FATHER %
```

The SON and FATHER commands allow explicit moves on the tree. Both take as argument the name of a CFF rule and modify the AML pointers. For a SON to succeed, the name of the rule must be the same as the name of one of the rules which are immediate sons of the current rule. For a FATHER, the rule must be an ancestor of the current rule, although not the immediate ancestor. The effect of both these commands is to move the AML pointers to the relevant node on the tree, and set the switch to the AML pointers. Since the tree is only accessible via the pointers, the effect is the desired move on the tree.

If the rule which was used as a template for the construction of the node pointed to by the pointers before the execution of the SON command is decorated with a NEWSCOPE command then the SON command will move the AML pointers into a nested scope.

A FATHER command could move the AML pointers out into a more global scope, if one of the nodes traversed in executing the FATHER command was constructed from a CFF rule with a NEWSCOPE decoration.

It is possible also to use SON and FATHER without arguments. The effect of FATHER when used in this way is to move to the immediate ancestor of the current node. The effect of SON is to move to the leftmost son of the current node. The remarks above concerning the switch and scope changes remain in effect.

```
"intolist"
INTOLIST %

"nextiteration"
NEXTITERATION %
```

The list structure in CFF rules (not to be confused with the AML LIST name table structure) introduced some notational problems and special statements were introduced to handle lists and their iterations. The INTOLIST command moves the AML pointers to the first element of the list. Here 'the list' means the outermost list in the rule pointed to by the AML pointers. The NEXT-ITERATION command moves the AML pointers to the first element of the next iteration of the list. A standard function, ANOTHERITERATION is used to indicate the presence or absence of further list iterations. These list commands are analogous to the FRONT, NEXT and EOL for name table lists, but apply to the CFF lists.

```
"mark"
MARK %

"release"
RELEASE %
```

When performing an INSERT operation, the pointer associated with the element being inserted in the name table can be made to point elsewhere on the tree by means of a special stack called the MARK stack.

This stack is operated on by the special AML commands MARK and RELEASE. MARK saves the current value of the pointers indicated by the switch on the MARK stack. RELEASE pops the top element of the MARK stack.

INSERT decides where to point the pointer in the name table by examining the MARK stack. If it is empty then the pointer is made to point to the same node on the tree that is pointed to by the set of pointers designated by the switch; otherwise, the pointer is made to point to the same node as that pointed to by the top element of the MARK stack.

AML STACK STATEMENTS

Thus far AML has statements that allow scoping, movement on the tree, and control abstraction, as well as a number of miscellaneous other statements needed for the AML problem domain. The question of including facilities that support typechecking has been ignored so far. This section introduces the typechecking facilities built into AML.

In a stack machine, when evaluating an expression, the operands are placed on the stack and then the operator is applied to the top elements. The result replaces the elements used in the evaluation. Type compatibility checking may be performed in a similar fashion.

Type checking operations fall into three categories, namely dyadic, monadic and assignment. The language specifier must provide, as part of the CFF/AML specification of the language, a set of tables indicating the allowed compatibilities, across each dyadic and monadic operator, and the result of the operation, as well as a set of allowed assignment compatibilities. These tables consist of lists of strings, and in general type compatibility checking in AML is achieved by pattern matching strings.

In many languages these compatibilities cannot be enumerated fully because of user-defined types, so a wildcard character can be used for general cases. This wildcard may be replaced by any string, with the condition that if there are many wildcards in one test, they are all replaced by the same string. Wildcards may not appear in the operator column of a table.

As the type checking of an expression proceeds, the types of the operands are placed on a built–in AML stack. At the relevant points, type checking commands are called which use the top elements of the stack and the relevant table to decide if the types are compatible and, if so, replace them with the type of the result.

The stack statements are:

```
"stack-statements"
push | pop | dcomp | acomp | mcomp %

"push"
PUSH(expression) %

"pop"
POP | POP(identifier) %
```

The push and pop statements are used to manipulate the AML stack explicitly. The PUSH takes as argument an expression which must be of STRING type, and pushes it onto the AML stack.

The POP command can be of two forms. In the first, the AML stack is just popped. In the second, the stack is popped and the top element of the stack is placed in the given identifier, which must be an AML simple variable of STRING type.

```
"dcomp"
DCOMP %

"mcomp"
MCOMP%

"acomp"
```

ACOMP %

Dyadic compatibility is checked by the DCOMP command, which takes the top three elements of the stack, interprets them as an operand–operator–operand triple, decides if the operands are compatible and replaces them by the resultant type of the operation.

Monadic compatibility is checked by the MCOMP command, which takes the top two elements of the stack, interprets them as an operator–operand pair, decides if the operator can be applied to the given operand type, and replaces them by the resultant type of the operation.

Assignment compatibility is checked by the ACOMP command, which takes the top two elements of the stack, interprets them as an lefthand side – righthand side pair, decides if the type of the right–hand side is assignment compatible with the type of the left–hand side, and removes them from the stack.

The decisions as to what constitutes a compatible type are made with reference to the compatibility tables. For example, for a dyadic compatibility check, the top three elements of the stack are treated as an operand–operator–operand triple and are matched against the rows of the dyadic compatibility table. If a match is found then the top three elements of the stack are replaced by the entry in the result field of the row with which the match was made. If no match can be made then the value std–error replaces the top three elements of the stack.

As an example, consider a language which has these entries in its dyadic compatibility

operand	operator	operand	result
integer	+	integer	integer
integer	=	integer	boolean
integer	+	real	real
set*	+	set*	set*

table. This table tells us that `integer` and `integer` are compatible under + with result of `integer`; that `integer` and `integer` are compatible under = with result `boolean`, that `integer` and `real` are compatible under + with result type of `real`, and that any type starting with `set` is compatible under + with the same result type (using the wildcard for specification).

If the stack had entries {integer, +, integer} then a DCOMP operation would replace these by integer after a successful lookup in the first row of the table. If the entries were {integer, +, boolean}, however, a match would not succeed and they would be replaced by the std-error value. If the entries were {setinteger, +, setinteger} then the lookup would succeed with the wildcard in the fourth row of the table being replaced by the string integer, and the result type which would also replace these three entries would also be .ft TA setinteger.

The monadic compatibility check operates in a similar fashion, using only the top two elements of the stack. The assignment compatibility check works in a similar fashion also, except that no result type is placed on the stack.

Note that operator commutativity is not taken into account, so if (real + integer) and (integer + real) were both to be allowed then two entries in the table would be needed. In some languages, such as Ada, operators may be dynamically overloaded in a program. These overloadings may be scoped so that they are only visible to some subset of the program, usually the module in which the overloadings are declared. The compatibility tables are capable of dynamic expansion to reflect new operator overloadings.

Each scope has associated with it a set of local compatibility tables. Often these tables will remain empty and only the pre–supplied tables will be used, but when overloadings are required, then the local tables can be used.

The commands LOAD-ACOMP-TABLE, LOAD-MCOMP-TABLE and LOAD-DCOMP-TABLE are used to add entries into the most local assignment, monadic and dyadic compatibility tables respectively. The number of arguments reflects the number of columns in the table; thus, the first of the load commands takes two arguments, the second takes three and the third takes four. The arguments are all string expressions. The syntax of these commands is given in the conventional statements section above.

When performing a table lookup to decide whether or not a particular set of operands are type compatible under a particular operator, the most local tables are searched first. If no match can be made there, then the remaining tables are searched scope by scope outwards to to most global pre–supplied tables.

STANDARD AML FUNCTIONS

A number of features of AML are defined as functions. These can be presented using a CFF–style syntax:

```
"unique"
UNIQUE(?stringpart <identifier_+>) %
```

UNIQUE takes as argument a list of AML name tables and checks that the key is not an element of the most local instantiation of each table. If the key is not found, then a value equivalent to an AML flag variable which has been SET is returned; otherwise a value equivalent to that of a RESET AML flag variable is returned.

```
"local"
LOCAL(?stringpart <identifier_+>) %
```

LOCAL takes as argument a list of AML name tables and enquires if the key is an element of the name table instantiations at the most local scope only. If the key is found, then a value equivalent to an AML flag variable which has been SET is returned; otherwise a value equivalent to that of a RESET AML flag variable is returned.

LOCAL will also set the switch to indicate the AML pointers.

```
"visible"
VISIBLE(?stringpart <identifier_+>) %
```

VISIBLE takes as argument a list of AML name tables and enquires if the key is an element of the name table instantiations at all visible scopes. If the key is found, then a value equivalent to an AML flag variable which has been SET is returned; otherwise a value equivalent to that of a RESET AML flag variable is returned.

VISIBLE will also set the switch to indicate the AML pointers.

```
"top"
TOP(?stringpart identifier) %
```

TOP takes as argument the name of an AML STACK name table and enquires if the key is the top element of the most local instantiation of this stack name table. If the key is found, then a value equivalent to an AML flag variable which has been SET is returned; otherwise a value equivalent to that of a RESET AML flag variable is returned.

TOP will also set the switch to indicate the AML pointers.

In addition to performing a name table lookup, each of the LOCAL, VISIBLE and TOP functions also has the effect of moving the AML pointers to the place on the tree pointed to by the pointer part of the name table entry, if the lookup succeeds.

```
"empty"
EMPTY(identifier) %
```

EMPTY takes as argument the name of an AML name table and enquires if the most local instantiation of the table is empty or not. If the table is empty, then a value equivalent

to an AML flag variable which has been SET is returned; otherwise a value equivalent to that of a RESET AML flag variable is returned.

EMPTY does not affect any pointers.

EMPTY is most useful in conjunction with the DUMP statement defined above.

```
"eol"
EOL(identifier) %
```

EOL takes as argument an AML LIST name table and returns true if there are no more elements in the list to scan, ie the internal pointer for the list points to the last element in the list.

There are a number of other AML functions. None of these functions takes any argument, unless the contrary is indicated.

CONTENTS returns the contents of a leaf node of the tree, pointed to by the set of pointers designated by the switch.

STACKTOP returns the top element of the type evaluation stack. The stack is left unaltered.

VALUE returns the value of a leaf node of the tree, pointed to by the set of pointers designated by the switch. This function is only effective if the contents of the leaf are numeric.

LENGTH returns the string length of the contents of a leaf node of the tree, pointed to by the set of pointers designated by the switch.

RULE returns the name of the CFF rule from which the current node of the tree was constructed, pointed to by the set of pointers designated by the switch.

SUBTREE is a boolean function which accepts as argument a name of a CFF rule and returns true if there is a subtree created from that rule rooted in the current node of the tree, pointed to by the set of pointers designated by the switch.

ANOTHERITERATION is a boolean function which returns true if the current CFF list has another iteration. The set of pointers designated by the switch must be pointing to a node inside a CFF list.

AML EXPRESSIONS

AML expressions are very similar to expressions in the Pascal family. An expression has the syntax:

```
"expression"
```

```
?sign <<factor_multop>_addop> ?predicate %

"predicate"
relop ?sign <<factor_multop>_addop> %

"factor"
 identifier ?argument
|(expression)
|string
|integer
|negation %

"negation"
NOT factor %

"sign"
+|- %

"addop"
+|-|OR%

"multop"
AND%

"relop"
=|'<>'|'>'|'<'|'>='|'<='|MATCHES %
```

There is only a boolean AND in the multop category as no need for multiplication and division has been found in AML.

The only novel operator is the relational operator MATCHES, which takes as arguments two string expressions, and returns true if the left hand string is a substring of (or the same string as) the right hand string.

Types in AML are only compatible with themselves, which results in very simple type

DYADIC COMPATIBILITY TABLE			
OPERATOR	OPERAND	OPERATOR	RESULT
integer	+	integer	integer
integer	-	integer	integer
string	+	string	string
*	=	*	*
*	<>	*	*
string	matches	string	flag
integer	<	integer	flag
integer	>	integer	flag
integer	<=	integer	flag
integer	>=	integer	flag
flag	AND	flag	flag
flag	OR	flag	flag

compatibility tables. See the dyadic compatibility table for the compatibilities for AML. Note (from the dyadic table) that + is defined on strings as a concatenation operator, and that = and <> are defined on all types, so that the wildcard may be used. Because the same substitution must be made for the wildcard throughout the row, this makes any type equality (and inequality) compatible with itself only. The flag type is a boolean

MONADIC COMPATIBILITY TABLE		
OPERATOR	OPERAND	RESULT
+	integer	integer
-	integer	integer

type.

ASSIGNMENT COMPATIBILITY TABLE	
LEFTHAND SIDE	RIGHTHAND SIDE
*	*

The monadic and the assignment compatibility tables are even simpler.

A `factor` may be an identifier. This can be an AML simple variable or an AML user-defined function, in which case it can take no argument. If it is an AML standard function, then it may take an argument, depending on the definition of the function.

Chapter 5: A SAMPLE LANGUAGE SPECIFICATION

To illustrate the use of the CFF/AML formalism, in this chapter a small sample language based on Dijkstra's small language [Dijkstra 76] is considered. For simplicity the language is considered at a stage in its development before the complex scoping conditions which are a feature of this language were introduced, and with slightly different handling of constants. In appendix III a fuller version of the language with complex scoping is given.

This language cannot demonstrate all the features of AML as its context conditions are not nearly sufficiently complex. It should, however, allow the reader to gain a good insight into the use of AML. Later examples will discuss how more complex context conditions can be specified in CFF/AML.

The statement part of the language consists of a `guarded-if`, a `guarded-do`, a `block`, a `skip` or an `assignment`. The syntax of these various statements (excluding the block for the moment) is:

```
"guarded-if"
IF <expression '->' <statement_;> _
   |>
FI %

"guarded-do"
DO <expression '->' <statement_;> _
   |>
OD %

"skip"
SKIP %

"assignment"
identifier := expression %
```

A `block` opens a new scope. The first part of the `block` consists of a set of declarations. These are followed by a statementlist:

```
"block"
BEGIN
 ?constants ?variables
<statement_;>
END %
```

Constant and variable declarations are similar to those of Pascal. The only types allowed in this language are INT and BOOL.

```
"constants"
CONST <identifier : type = constant_;> %

"constant"
 identifier
|integer
|TRUE
|FALSE %

"variables"
VAR <<identifier_,> = type_;> %

"type"
 INT
|BOOL %
```

An expression is as defined above. Only a limited set of factors are allowed, compared with, say, Pascal. This is due to the paucity of types in the language.

```
"expression"
?sign <<factor_multop>_addop> ?predicate %

"predicate"
relop ?sign <<factor_multop>_addop> %

"factor"
 identifier
|integer
|TRUE
|FALSE
|NOT factor
|( expression ) %

"addop"
+|-|AND %

"multop"
*/|OR %

"relop"
=|<>|>=|<=|<|> %
```

The first part of building an AML specification on top of a CFF syntax is to decide where the scope changes are, what name table instantiations are required and whether these instantiations are to be of `set`, `stack` or `list` type. In the case of this language, two name tables of `set` type are used, one for constants and one for variables. Splitting the declared identifiers up in this way makes assignment checking easier.

Call these tables vars and consts. The block rule is decorated to show the declaration of these tables:

```
"block"
BEGIN
[ newscope ;
  SET vars, consts ;
] ?vars ?consts
  <statement_;>
END [ endscope ]%
```

The next phase is to decorate all places in the CFF rules where information must be inserted into the name tables. In this language, the only places where this is required is on the constants and variables rules. All identifiers must be declared before use. The context conditions may therefore be specified in a one–pass fashion. Some languages may require multiple context condition passes, usually one to set up the name tables and another to perform type checking.

The decorations for the CFF constants rule is:

```
"constants"
CONST <identifier [ if unique(consts + vars) then
                      insert(consts)
                    else
                      error("Constant already declared")
                  ] : type = expression _;> %
```

This decoration is not complete; the type of the constant must match the type of the expression. We will return to this below.

The check for variables is very similar to that for constants:

```
"vars"
VAR <<identifier [ if unique(vars + consts) then
                      insert(vars) ;
                   else
                      error("Identifier already declared")
                 ]_,> : type_;> %
```

The third phase of the decoration process involves inserting the type checks where required. Type evaluations assume the existence of a function type which returns the type of an identifier. This function is described below.

Type checks are required on the assignment, conditional and loop statements, as well as on the constant declaration. Consider the assignment statement. We assume that the decorations for the CFF rule expression have been written in such a way as to

cause the process of walking the tree constructed from the expression rule to result in only one element – the resultant type of the expression – being added to the AML stack. This can be proven to be the case once the formal system of part III is constructed.

If the context check for the left–hand side of the assignment is arranged so as to leave the type of the left–hand side on the stack then the situation which arises is one where, when we are finished with the assignment, we have the type of the expression on the stack immediately above the type of the left–hand side.

An AML acomp command will use the assignment compatibility tables to check that the types are assignment compatible. The decorations are as follows:

```
"assign"
identifier [ if visible(vars) then
                push(type)
             else
               error('identifier not declared')
           ] := expression [ acomp ] %
```

The AML code on the left–hand side identifier checks that the identifier is declared by looking in the AML name table called vars. If the identifier is found then the AML function type is called and the value returned by the function is pushed onto the stack.

As discussed above, we assume that the expression has left one entry – the resultant type of the expression – on the stack. The acomp checks that the two types are compatible, removes the top two stack elements and, if necessary, produces an error message.

Consider now the two guarded command statements. In both cases, the only check needed is that the types of the guard expressions are boolean. The type must then be popped off the stack.

```
"guarded-if"
IF < expression [ if stacktop <> 'boolean' then
                    error('expression not of boolean type') ;
                  pop
                ] '->' <statement_;> _|>
FI %

"guarded-do"
DO < expression [ if stacktop <> 'boolean' then
                    error('expression not of boolean type');
                  pop
                ] '->' <statement_;> _|>
OD %
```

The block and skip CFF rules – the other statements in our sample language – need no

further decoration.

Before looking at the decoration of the expression rule, let us briefly consider the completion of the decorations for the constants. Constants in this language are strongly typed in that the type of the constant is stated directly, not assumed from the type of the expression, and that any identifier may be used in the expression making up the value of the constant, provided only that the identifier is not local.

The relevant rules are:

```
"constants"
CONST <identifier [ if unique(consts + vars) then
                      insert(consts)
                    else
                      error("Constant already declared")
                  ] : type = expression _;> %

"type"
 INT
|BOOL %
```

The obvious decorations are to add in an acomp after the expression, and decorate the type rule to push the type onto the stack. There is only one small problem with this approach, namely that the rule type is used elsewhere in the grammar as well. The types must not be pushed onto the stack each time the types rule is walked through, but only when coming from the constants rule. An AML simple simple variable of flag (boolean) type — INCONSTANT — is introduced to facilitate this. When the flag is set it indicates that type pushing is required; when reset, that it is not required. The context decorations on the constants rule are modified to set and reset the flag. The flag should be reset before the expression to prevent any unexpected side effects when evaluating the expression type.

```
"constants"
CONST <identifier [ if unique(consts + vars) then
                      insert(consts)
                    else
                      error("Identifier already declared") ;
                    set(inconstant)
                  ] : type[ reset(inconstant)
                          ] = expression[ acomp ]_;> %
```

The rule type can now be decorated. If the flag is set, then the requisite type is pushed; otherwise no action is required.

```
"type"
 INT   [ if inconstant then push('integer') ]
|BOOL  [ if inconstant then push('boolean') ] %
```

The last rule group to consider are the rules for expressions. This decoration is quite complex and tends to be very similar for most programming languages. The AML flags monad and dyad are used to indicate whether a monadic or dyadic check, respectively, is required. The precedence of operators is handled by the structure of the syntax. The fully decorated code for an expression is:

```
"expression"
[ reset(dyad) ; reset(monad) ]
?sign<<factor[ if monad then
                  begin
                    mcomp ;
                    reset(monad)
                  end ;
                if dyad then
                  begin
                    dcomp ;
                    reset(dyad) ;
                  end
               ]_multop[ set(dyad)
                       ]>_addop[ set(dyad)
                               ]> ?predicate %
```

The decoration for the predicate follows a similar pattern with extensions for the relational operator.

The decorations of the rule factor must all ensure that the resultant type of the factor is left on the stack.

```
"factor"
identifier [ if visible(vars + consts) then
               push(type)
             else
               error('identifier not declared') ]
|integer    [ push('integer') ]
|TRUE       [ push('boolean') ]
|FALSE      [ push('boolean') ]
|NOT factor[ if stacktop <> 'boolean' then
               error('Factor must be of boolean type') ]
|[ save(monad) ;
   save(dyad)
 ]( expression ) [ restore(monad) ;
                   restore(dyad) ] %
```

The decoration for identifier is as for the left hand side of assignment statement. In the case of an integer or a boolean literal, the correct type is pushed. For a negation, the type of the factor must be boolean. For a subexpression, the flags are saved so that the evaluation can continue in the correct state after the evaluation of the subexpression terminates.

The decoration on the operators is straightforward. Only addop and sign are considered here; the other cases follow by analogy.

```
"addop"
 +   [ push('+')  ]
|-   [ push('-')  ]
|OR  [ push('OR') ] %

"sign"
 + [ push('+') ; set(monad) ]
|- [ push('-') ; set(monad) ] %
```

The remaining parts of the language definition are auxiliary, namely the function type, the compatibility tables and the declaration of the AML simple variables.

The function type is only called after a visible operation has succeeded. The effect of this is to point the AML pointers at the declaration of the identifier. The type can then be found by going down the subtree rooted in the nonterminal type and then determine the type by looking at the contents of the node:

```
FUNCTION type RETURN string ;

BEGIN
  son(type) ;
  IF contents = 'INT'
    THEN
      type := 'integer'
    ELSE
      type := 'boolean'
END
```

MONADIC TABLE		
operator	operand	result
+	integer	integer
-	integer	integer

DYADIC TABLE			
operand-1	operator	operand-2	result
integer	+	integer	integer
integer	-	integer	integer
integer	*	integer	integer
integer	/	integer	integer
boolean	AND	boolean	boolean
boolean	OR	boolean	boolean
integer	<	integer	boolean
integer	<=	integer	boolean
integer	>=	integer	boolean
integer	=	integer	boolean
integer	<>	integer	boolean
integer	>	integer	boolean

ASSIGNMENT TABLE	
lefthand side	righthand side
*	*

The compatibility tables are simple as well; a type is only compatible with itself. Note the use of the '*' in the assignment table. This is a wildcard character and has the feature that it may be replaced by any string with the restriction that all entries in the same row must be replaced by the same string when performing pattern matching.

The declarations come before the first rule of the grammar. Each declaration consists of a type followed by a list of identifiers with that type:

```
DECLARE
  FLAG monad, dyad, inconstant ;
```

Only flag variables need be declared for this language specification.

Chapter 6: ASPLE in CFF/AML

This chapter gives a CFF/AML description of a simple language called ASPLE. In most respects, ASPLE is a simpler language than Dijkstras Language which described in chapter 5. There are, however, two good reasons for looking at ASPLE here.

The first is that ASPLE is often used as an example language to illustrate the use of a particular specification formalism [Ambriola et al 84, Despeyroux 84, Marcotty et al 76] and so is highly suited as an example in order that the reader can get a feel for how concise and usable CFF/AML really is.

The second reason is that although ASPLE is generally a simple language it is a derivative of ALGOL 68, with that languages method of reference of values. This makes the type evaluation and compatibility checking processes somewhat different from those in the previous example and thus illustrates the use of CFF/AML on a slightly different sort of language.

The syntax of ASPLE in CFF is based on the BNF syntax given in [Ambriola et al 84].

```
"program"
BEGIN <decs;_> <stmt_;> END %

"decs"
?mode type <identifier_,> %

"type"
 INT
|BOOL %

"mode"
<REF_> %

"stmt"
 assign
|if
|while
|transput %

"assign"
identifier := expression %

"if"
IF expression THEN <stmt_;> ?else FI %

"else"
ELSE <stmt_;> %
```

```
"while"
WHILE expression DO <stmt_;> END %

"transput"
 INPUT identifier
|OUTPUT expression %

"expression"
<<factor_*>_+> %

"factor"
 identifier
|TRUE
|FALSE
|integer
|(expression)
|(compare) %

"compare"
expression relop expression%

"relop"
 =
|<> %
```

An ASPLE program consists of one scope level with a list of declarations followed by a list of statements. The checks that are needed are that all identifiers used in the body of the program are declared, and that all operators in expressions are applied to operands of compatible type only.

An identifier may be declared directly, or with a preceding list of REFs, each of which indicates a level of indirection. If x is declared as

INT x

it has primitive type INT and one level of indirection, but if declared as being of type

REF INT x

then it has primitive type INT and *two* levels of indirection.

The operators + and * are addition and multiplication operators when applied to integers and OR and AND operations when applied to boolean operators. When applying an operator to its arguments, only the primitive types need be considered to determine if the operands are compatible.

The convention for expressions is that if an expression has only one operand then its type is the type of the operand and the number of levels of indirection associated with the expression is the same as the number of levels of indirection associated with the operand. If the expression has multiple operands then the type of the expression depends on the result types of the operands, but the number of levels of indirection is zero, as the expression simply returns a value. A literal has zero levels of indirection associated with it. The reader should note that this convention is adopted simply as a possible solution to the problem of what happens when an operator is applied to two operands each of which has different levels of indirection, and reflects the solution of [Ambriola et al 84]. Any other solution could also be adopted without prejudice to the specification.

For assignments, the left and right hand sides must both have the same primitive type, and the level of indirection of the left hand side (nl) must be at least one less than the level on the right hand side (nr), ie

$$nl-1 <= nr.$$

The declared identifiers are placed in a name table called vars. The table is declared in a decoration on the program rule.

```
"program"
BEGIN [newscope;
         set vars ;
       ] <decs;_> <stmt_;> [endscope] END %
```

As each identifier is declared, it is checked to see that it is not already in the vars table. If not it is inserted.

```
"decs"
?mode type <identifier[ if unique(vars) then
                           insert(vars)
                        else
                           error('identifier already declared')
                      ] _,> %
```

The nonterminals mode and type require no decoration. Neither does the stmt rule. The various statement rules can now be decorated, starting with the assignment rule. The decoration of this rule is complicated by the fact that as well as having the correct type compatibility, the levels of indirection must satisfy the formula given above.

```
"assign"
identifier[if not visible(vars) then
                error('identifier not declared')
           else
```

```
      begin
        get-type-info ;
        nl := num-refs ;
      end ;
    push(pritype)
  ] := expression [acomp ;
                     if nl-1>nr then
                       error('incompatible referencing')] %
```

Here, nl and nr are AML integer variables, and get-type-info is the type evaluation procedure. Assume for the moment that it returns the primitive type in the string variable pritype, and the number of levels of indirection in num-refs, and also assume that the expression rule leaves the primitive type of the expression on the stack and the number of levels of indirection associated with the type of the result in the integer variable nr.

The if and while statements require only the usual check that the result type of the expression is boolean. Note once again that this is a convention which could be altered without prejudice to the specification. If a number of levels of indirection (or a guarantee of no levels of indirection) were required then these could be introduced with minimal overhead.

```
"if"
IF expression[ if stacktop <> 'bool'
                 then
                   error('Invalid type of expression') ;
                 pop
               ] THEN <stmt_;> ?else FI %

"while"
WHILE expression[ if stacktop <> 'bool'
                    then
                      error('invalid type of expression')
                  ] DO <stmt_;> END %
```

The else rule requires no decoration. The transput statement requires only a check that the identifier is declared, and the expression's result type is cleared off the stack for housekeeping purposes. Once again, if it were required that (say) identifiers had no more than one level of indirection for input and (say) that expressions had no levels of indirection for output (or any other convention) then these could be added with minimal effort.

```
"transput"
 INPUT identifier[ if not visible(vars) then
                     error('identifier not declared')]
```

```
|OUTPUT expression[ pop ] %
```

The only remaining rules to be decorated are those for expressions. The decorations follow a similar pattern to those for Dijkstra's Language introduced above, so minimal explanation is needed.

The function type in the previous example becomes a procedure here, called get-type-info. This procedure must determine the primitive type of the identifier and place this in the AML string variable pritype much as before, but also it must set the AML integer variable num-refs to reflect the number of indirections in the type of the identifier.

```
PROCEDURE get-type-info ;

 begin
   num-refs := 1 ;  { all identifiers have at least one
                      level of indirection }
   if subtree(mode) then  { if there is a REFs list }
     begin
       saveposition ;    { note position on the tree }
       son(mode) ;       { onto the mode list rule }
       intolist  ;       { go into the list itself }
       while another-elt do
         begin
           num-refs := num-refs + 1;  { count the REFs }
           next-iteration ;
         end ;
       restoreposition ; { go back to declaration }
     end ;

   son(type) ;       { get the primitive type }
   if contents = 'BOOL'
     then pritype := 'bool'
     else pritype := 'integer' ; { set the type }
 end
```

DYADIC COMPATIBILITY TABLE			
OPERAND	OPERATOR	OPERAND	RESULT
integer	+	integer	integer
integer	*	integer	integer
bool	+	bool	bool
bool	*	bool	bool
*	=	*	bool
*	<>	*	bool

ASSIGNMENT COMPATIBILITY TABLE	
LEFT HAND SIDE	RIGHT HAND SIDE
*	*

No monadic compatibility table is needed for ASPLE. The AML simple variable declarations for this language are:

```
DECLARE
    STRING pritype ;
    INT    nr, nl, num-refs ;
    FLAG   dyad, no-indirects ;
```

A SAMPLE EVALUATION

Let us at this stage consider how a sample program is evaluated. Given the ASPLE program, taken from an example in [Ambriola et al 84]:

```
BEGIN
   INT a ;
   BOOL b ;
   REF INT c ;
   REF REF int d ;

   a := 16 ;          (1)
   c := d ;           (2)
   a := b ;           (3)
   c := 20 ;          (4)
   d := a ;           (5)
```

```
END
```

where the numbers down the right hand side are for reference purposes and not part of the program.

We outline the process of the AML evaluations of this program. The reader is encouraged to trace the execution in the specification of the ASPLE rules.

When all the declarations have been scanned, the entries `a,b,c` and `d` will all be in the name table `vars`. Scan the assignment numbered (1). The type of `a` is evaluated by `get-type-info`, which will return a `pritype` of `integer` and a `num-refs` of 1. Since 16 is a degenerate expression, little evaluation need be performed; the string `integer` is pushed onto the stack. The `acomp` will succeed, as `nl=1`, `nr=0`, and `1-1<=0`.

For the lefthand side of assignment (2), the `pritype` will evaluate to `integer`. When evaluating the `num-refs`, however, the `subtree(mode)` will succeed and one will be added to the initial value of `num-refs`, making it 2. For the righthand side, the `pritype` of `d` is also `integer`, but `num-refs` will be 3. With `nl=2` and `nr=3`, `nl-1<=nr` still holds, and the assignment is correct.

For the third assignment, the lefthand side has `pritype integer` and `nl=1` and the righthand side has `pritype bool` and `nr=1`. This time, the `acomp` operation will fail as only identical pritypes are assignment compatible.

For assignment 4, the lefthand side has `pritype integer` and `nl=2`, whereas the righthand side, being an integer literal, has `pritype integer` and `nr=0`. This time the `acomp` will succeed but the `nl-1<=nr` predicate does not hold so the assignment is in error.

For the final assignment, the lefthand side has `pritype integer` and `nl=3` and the righthand side has `pritype` of `integer` as well, but `nr` of 1. This assignment is also in error for the same reason as assignment 4.

The power and elegance of the CFF/AML formalism for specifying context conditions should now be apparent. The entire specification of ASPLE is of the order of 120 lines long with the AML code prettyprinted, and took rather less than half an evening to construct from the first sight of the language.

Chapter 7: TYPE EVALUATION OF RECURSIVE TYPES

The examples in chapters 5 and 6 have illustrated the ease of use of AML. The example languages have, however, been strikingly lacking in the sort of complexities that bedevil the context condition specifications in 'real–life' languages. This section considers a more complex example based on Pascal, namely that of type evaluation of recursive records and pointers.

A pointer definition consists of a pointer token followed by an identifier. The identifier must be a type identifier and must be declared locally. The declaration of the pointer may precede that of the type pointed to by the pointer, ie the type identifier may not yet be in a name table at the point of declaration of the pointer. We assume that types in this example are:

```
"types"
TYPE <identifier = type _;> %

"type"
 pointer
|record %

"pointer"
^ identifier %

"record"
RECORD
  fields
END %

"fields"
<<identifier_,> : type_;> %
```

and assume the existence of two name tables, namely `types` and `fwdtypes`. All type identifiers are inserted into the table `types`. If, however, the pointer–identifier on the right–hand–side of the pointer declaration is not yet declared, then that identifier is placed into `fwdtypes`, and deleted when the actual declaration occurs. If, at the end of the declaration part of the scope level, the identifier has not been declared then an error results.

The rule `types` is decorated in a similar manner to that used for constants and variables in the first example above. The additional code handles the declaration of forward–referenced pointer types.

```
"types"
```

```
TYPE <identifier [ if unique(types) then
                     begin
                       insert(types) ;
                       if local(fwdtypes) then
                         delete(fwdtypes)
                     end
                   else
                     error('identifier already declared')
                 ] = type _;> %
```

The rule type requires no decoration; the rule pointer requires a check that the identifier is declared. If it is not, then the identifier must be inserted in fwdtypes:

```
"pointer"
^ identifier [ if not local(types) then
                   insert(fwdtypes) ] %
```

A record is interesting from the context–condition viewpoint. The record defines a new scope. The identifiers in a record are not placed in a set name table but in a list table. This simplifies type evaluation for a record and also aggregate assignment to the components of a record, which is considered in the next example below.

The decoration is:

```
"record"
RECORD [ newscope ;
         list vars ; ]
  fields
END [ endscope ]%

"fields"
<<identifier [ if unique(vars) then
                 insert(vars)
               else
                 error('Component already declared')
             ]_,> : type_;> %
```

The declaration of the rule types also needs code to check that all forward references have been resolved.

```
"types"
TYPE <identifier [ if unique(types) then
                     begin
                       insert(types) ;
                       if local(fwdtypes) then
                         delete(fwdtypes)
                     end
                   else
```

```
        error('identifier already declared')
    ] = type _;> [ if not empty(fwdtypes) then
                     begin
                       error('Pointers unresolved') ;
                       dump(fwdtypes)
                     end ] %
```

The type evaluation of a complete record can now be described. The type of a record consists of the string 'RECORD' concatenated with the types of the components of the record.

For recursive definitions, if the record contains a pointer to itself then that pointer type must not be evaluated as that will cause infinite regression. A name table called recnames is used to prevent this. Each recursive reference is then replaced by the record name rather than the type of the record. The following code fragment would be contained in an evaluation procedure or function. TYPE is a string variable.

Note that the savepositions and restorepositions make it possible for records to be nested. If the record is recursively defined then the self-referential pointers are replaced by the name of the record type as opposed to doing an evaluation of the type pointed to by the record. We assume that the recnames name table has been CLEARed before the call of this procedure.

```
procedure evaluate-record-type ;

  begin
    if rule = 'record' then
      begin
         type := type + 'RECORD' ;
         son(fields) ;
         front(vars) ;
         while not eol(vars) do
           begin
             saveposition ;
             son(type) ;
             evaluate-record-type ;
             restoreposition ;
             next(vars) ;
           end ;
         exit ;
       end ;

    if rule = 'pointer' then
      begin
         type := type + 'PTR' ;
```

```
            if visible(recnames) then
              type := type + contents
            else
              begin
                insert(recnames) ;
                if local(types) then
                  evaluate-record-type
                else
                  error('Invalid identifier') ;
              end ;
        end ;
    end ;
```

A simple example such as

```
TYPE X = RECORD c : ^X END
```

will yield a type of (with periods used to separate the components):

```
RECORD.PTR.RECORD.PTR.X
```

For a more complex example with mutual recursion such as:

```
TYPE R = RECORD a : ^ R ; b : ^ P END

     P = RECORD x : ^ R ; y : ^ P END
```

a type evaluation on record R or record P will yield a type of

```
RECORD.PTR.RECORD.PTR.R.PTR.RECORD.PTR.R.PTR.P
```

which will be acceptable in a language where structural equivalence implies type compatibility. If we did not want structural equivalence to imply type compatibility then simply placing the name of the record into `recnames` before starting the type evaluation, and then calling the evaluation routine will cause evaluations on P and R will yield different types.

This example is significantly more complex than its predecessors; however, so are the context specifications associated with it. Once again, the CFF/AML approach can lead to a simple solution to a complicated problem.

Chapter 8: HANDLING RECORD AGGREGATES

We now extend the example of chapter 7 to see how CFF/AML handles checking that each element of an aggregate record assignment is correct. This is one of the 'new' context features which has entered mainstream programming languages for the first time with its inclusion in Ada. For the sake of simplicity only positional record aggregates with no nestings are considered here; named aggregates can be handled in a similar fashion, and nesting records introduces some small additional complexity.

Suppose the syntax of the previous example is extended to include the aggregate assignment statement:

```
"agg-assign"
identifier := (<expression_,>) %
```

which is a degenerate form of the actual aggregate assignment in Ada. The following context conditions are imposed on the rule:

- The identifier must be the name of a record identifier;
- There must be the same number of expressions in the right hand side list as there are components in the record, and each expression must have the same resultant type as the type of the record component, with the components and expressions matched positionally.

This is a simpler version of the context conditions for record aggregates in Ada.

We assume that the set of types is richer than that of the previous example (although we are not concerned with what these types might be) and that variables are declared using the rule:

```
"vars"
VAR <<identifier_,> : identifier;_> %
```

where the first identifier must be unique in the name table `vars`, and the second must be in the name table `types`. The AML decorations on this rule are:

```
"vars"
VAR <<identifier[ if unique(vars) then
                     insert(vars)
                  else error('Identifier already declared')
                ]_,> : identifier[ if not visible(types) then
                                      error('invalid type')
                                 ];_> %
```

Before showing the solution in AML, some informal explanation of what the solution attempts to do is in order.

First the identifier on the lefthand side of the assignment sign must be checked to ensure that it is a record identifier. The AML pointers are left pointing to the front of the list of components.

Then for each expression in the list, work through the following algorithm:

- Get the type of the next component, and push it onto the AML stack used for type checking.
- Walk the expression. Assume that this leaves the type of the expression on the top of the stack without affecting any other elements of the stack, as usual.
- Check that the top two elements of the stack are assignment compatible.
- Position the AML pointers at the next element of the list of components of the record. If there are no more, and there are more expressions in the list of aggregate elements, then generate an error.
- When the end of the list of aggregate elements is reached, if there are more components in the record then generate an error.

This is a very simple skeleton of the resulting AML code. The lefthand side of the assignment is decorated to get:

```
"agg-assign"
identifier[ if visible(vars)
              then
                check-record-type
              else
                error('Not a record identifier')
          ] := (<expression_,>) %
```

where check-record-type is assumed to leave the AML pointers correctly positioned at the start of the AML list table vars associated with the record. The code to achieve this is:

```
PROCEDURE check-record-type ;

  begin
    son(type) ;
    son ;
    if rule <> 'record'
      then error('not a record type')
      else
        begin
          son(fields) ;
          front(vars) ;
```

```
                    end ;
                end ;
```

The decoration on the remainder of the agg-assign rule is then:

```
"agg-assign"
identifier[ if visible(vars)
              then
                check-record-type
              else
                error('Not a record identifier')
          ] := (<[ amlscope ;
                   saveposition ;
                   type-evaluate ;
                   push(type)
                 ]expression[ acomp
                            ]_,[ amlscope ;
                                 restoreposition ;
                                 if eol(vars) then
                                   error('aggregate overflow')
                                 else next(vars)
                               ]>)[ if not eol(vars) then
                                      error('agg underflow') ]%
```

The group of decorations around the expression list achieve the following purposes:

- o The first group after the '(' get the type of the component and push it onto the stack.
- o Assuming that the expression behaves correctly, the acomp checks that the types are indeed assignment compatible.
- o The next decoration handles the case that there are more elements in the aggregate than the record, and moves the AML pointers to the next element in the list.
- o The final decoration checks that there are not more components in the record than elements in the aggregate list.

The internal operation of the `type-evaluate` procedure is not dealt with.

This example illustrates vividly the advantage of the LIST type. The component list of the record has the form

```
<<identifier_,> : type_;>
```

As far as the aggregate list is concerned, the declarations

```
RECORD a,b : integer; c : integer END
```

and

RECORD a : integer; b: integer; c: integer END

and

RECORD a : integer; b,c : integer END

are the same. It would be complex to write code that will navigate the lists correctly and produce the component types in the correct order regardless of the structure of the lists.

However, the LIST name table structure comes to the rescue as the list and only one simple list walking algorithm is needed.

This is about as complex as AML decorations ever seem to get. A modification of this approach is used for parameter checking and array subscript checking. The array subscript checking uses the CFF list moving primitives, whereas to check procedure and function parameters, the formal parameter identifiers are placed in a list name table and which is then walked.

Chapter 9: HANDLING OPERATOR OVERLOADING

As a final example of the use of CFF/AML, consider the problem of specifying overloading in AML. We loosely base our example on operator overloading in Ada.

Operator overloading involves the definition of a new "meaning" for a particular operator, for example, + could be redefined to mean addition of vectors as well as addition of simple numbers. In Ada this is done by specifying the new overloading of the function symbol as a function, eg:

```
FUNCTION "+"(x,y : IN vector) RETURN vector ;

  begin
    -- code to add the two vectors
  end ;

where vector could be defined to be:

TYPE vector IS ARRAY (1..10) OF integer ;
```

The number of formal parameters must be checked to be correct. Then the types of the formals and the result must be placed into the most local AML compatibility table, working from the scope of the tree walk pointers. Then, any reference to the operator using the overloaded meaning will be correctly resolved in the compatibility tables. Note that the compatibility tables are scoped, so the overloading will only be visible in the correct scope.

A possible syntax for the operator overloading is:

```
"operator-overloading"
FUNCTION operator params RETURN identifier ;
 body %

"operator"
+|-|*|/%

"params"
(<<identifier_,> : typeid_;>) %

"typeid"
identifier %
```

The contents of the rule body are not considered; also, only a subset of the operators is considered, although the other cases can follow by analogy.

The operator rule is decorated first. The name of the operator is placed into a string called opname for later insertion into the relevant compatibility table.

The operators * and / are dyadic but the + and - are either monadic or dyadic in context. A flag mustdyad is set on * and / and reset for the other operators. The operator rule can be decorated to become:

```
"operator"
 + [reset(mustdyad) ; opname := '+']
|- [reset(mustdyad) ; opname := '-']
|* [set(mustdyad) ; opname := '*']
|/ [set(mustdyad) ; opname := '/'] %
```

The parameters rule is decorated next. The integer variable num-params is used to count the number of parameters given thus far, and the string variables ltype and rtype to hold the types of the left and right operands respectively. The list symbol table params is used for checking formal/actual parameter correspondence when the operator is used as a standard function call and the symbol table vars is the usual variables declaration table for a scope. The parameters must be put into this table as they are local to the scope of the function declaration.

```
"params"
([ num-params := 0
 ]<<identifier[ num-params := num-params + 1 ;
                if num-params > 2 then
                  error('Too many parameters')
                else begin
                  if unique(vars) then
                    begin
                      insert(vars);
                      insert(params) ;
                    end
                  else
                    error('Identifier already declared') ;
                  son(typeid) ;
                  if not(visible(types)
                    then
                      begin
                        error('not a type identifier') ;
                        type := std-error ;
                      end
                  else
                    type -evaluate ;
                  if num-params=1 then ltype := type
                                  else rtype := type
```

```
        end
    ]_,> : typeid_;>)
```

This is another long decoration. The length of the decoration reflects the complexity of the context conditions of the overloading, viz. that there must be a specific number of parameters and that the types must be collated for insertion into a compatibility table, as well as all the regular checks that are required for parameter checking, ie that they are unique in the scope, and that the typeid be a valid type identifier. None of the individual components of the checks are difficult to write or understand, but when put together they do tend to become quite bulky.

This bulk can be avoided (or at least hidden) by declaring some more AML procedures and hiding the details of the evaluation away (as has once again been done with the procedure type-evaluate, which we is assumed to set the string variable type to reflect the type of the type identifier).

The operator-overloading rule can now be decorated. Most of the decoration would be required for any function declaration, with only the load statement at the end of the operator-overloading rule being specific to an overloading.

```
"operator-overloading"
FUNCTION[ newscope ;
          SYMBOL vars ;
          LIST params
        ] operator params [endscope
        ]RETURN identifier[ if visible(types)
                              then
                                type-evaluate
                              else begin
                                error('Invalid identifier')
                                type := std-error ;
                              end
                            if num-parms=1 and mustdyad
                              then
                                error('Too few arguments')
                            if num-parms =1 then
                              mcomp-table-load(opname,ltype)
                            else
                              dcomp-table-load(ltype,opname,rtype)
                          ] %
```

The new overloading of the operator can then be used in functional or infix notation, the one way using the parameters and the other using the tables. To overload full procedures and functions a similar procedure is followed, with overloading resolution by building a

PROFILE for each procedure or function and then putting these profiles in a table. Then checking a call becomes a matter of constructing a profile for the procedure name and actual parameters and then seeing if this matches a procedure declaration profile.

PART II – TEMPORAL LOGIC PROOF THEORY

There are two elements lacking in the CFF/AML which described in part I of this thesis. Firstly, the definition of AML is informal. Secondly, a method is required for formally proving correct certain properties of a CFF/AML specification, for example that the decorations on an expression always leave only one element on the AML stack, or that an AML type evaluation procedure always terminates, and with a valid type evaluation each time. This part of the thesis lays the foundation for the solution to these two problems.

The second problem was solved, in principle at least, first. Because of the procedural nature of AML, it would obviously be easiest to achieve proofs of the correctness of specifications using a verification methodology. The problem then became one of how to verify AML specifications. There were several arguments in favour of the use of a verification approach based on temporal logic (called the temporal proof approach in the remainder of the thesis).

- The temporal proof approach is well suited to proving CFF/AML specifications correct because temporal proofs operate on a graph. A CFF rule can be thought of as a graph (the syntax diagram of the rule, in effect), augmented by a number of subgraphs (the graphical representation of the AML code, in effect). The temporal proof approach can therefore easily handle a mix of syntax and context conditions if they are all treated as one graph.
- Temporal proof systems are generic [Manna & Pnueli 83a], *i.e.* that they can be tailored to specify particular programming languages. This is a second major reason for turning to a temporal proof approach to verification of AML code; in building a temporal proof system for a programming language, the semantics of the language become formally specified. Building such a proof system would therefore satisfy the twin objectives of this part of the thesis — the construction of a formal specification of AML, and the creation of a verification system for context conditions of programming languages written in AML. The reader should note that this is not a trivial extension of earlier work using temporal proof systems. The problem domain of AML and the dynamically scoped nature of the symbol tables in AML make the specification significantly more complex than those of previous languages, which have included a simple sequential language (without scope or subprograms) [Manna 82], a simple concurrent language [Manna & Pnueli 83b] and CSP [Manna & Pnueli 83a].

- If we wish to verify recursive procedures and functions which operate without parameters and which perform ostensibly random manipulations on a tree, we need a verification methodology which will support this. The traditional approach to verification of recursive programs would be to use the fixpoint of the function calculated by the program. In this case, however, such fixpoints are not easily attainable. A temporal proof approach, however, allows simple verification of such programs (as we will see below). The temporal proof approach is therefore preferable to other approaches such as that of [Hoare 71].

The temporal proof approach was therefore adopted as both as the mechanism for building a formal specification of AML, and as the means by which to verify that AML specifications are correct. However, there was a problem with this approach that first required rectification.

This part of the thesis therefore digresses from the issues of context condition specification in order to lay a foundation for part III, in which a formal semantics for AML will be constructed and then used for formal proof of the correctness of CFF/AML specifications.

The temporal proof approach developed by Manna and Pnueli is deficient in that it cannot handle procedures or functions. We wish to use this approach both formally to define AML and to prove AML specifications correct. However AML subprograms, especially recursive subprograms are a vital part of the specification language. It is therefore necessary to expand Manna and Pnueli's system to handle subprograms.

This part first gives an overview of Manna and Pnueli's temporal proof system [Manna & Pnueli 83b]. Then this is expanded to handle subprograms, and then expanded further to include the treatment of recursive subprograms. Some simple illustrative examples are also included.

Chapter 10: OVERVIEW OF A TEMPORAL PROOF SYSTEM

Temporal logic (TL) is a first–order language with the usual boolean connectives, equality and quantification. There are also four temporal operators:

□ – The 'box' or always operator

◇ – The 'diamond' or sometimes operator

O – The 'circle' or next operator

U – The until operator

the first three of which are monadic and the fourth dyadic.

The variables of the language are partitioned into global and local sets. Global variables are unchanged with time, but local variables may change their value from instant to instant. Only global variables may be quantified.

A model(I,α,σ) for the temporal language consists of a global interpretation I, a global assignment α and a sequence of states σ.

The interpretation I specifies the domain(s) of operation and assigns 'meaning' to the symbols of the language.

The assignment α defines the values of all global variables.

Temporal formulae are interpreted over infinite sequences of states σ:

$$\sigma: s_0, s_1, s_2, \ldots$$

Each state s contains value–assignments for all the individual variables. Global variables are assigned the values that they held in the previous state, but local variables may change value in the transition from state to state.

The k–shifted sequence of a state σ is denoted:

$$\sigma^{(k)} : s_k, s_{k+1}, \ldots$$

Only the monadic temporal operators are used in this thesis. The interpretation of temporal formulae over sequences of states is given below. The formula "$\sigma \models$ w" should be interpreted as meaning the formula w interpreted over state sequence σ).

For classical formulae, ie ones with no temporal operators, their interpretation is time independent, ie

$$\sigma \models w \text{ iff } s_0 \models w$$

For the monadic temporal operators, the interpretations are:

$$\sigma \models \Box w \text{ iff } \forall k \geq 0,\ \sigma^{(k)} \models w$$
$$\sigma \models \Diamond w \text{ iff } \exists k \geq 0\ \ \sigma^{(k)} \models w$$
$$\sigma \models Ow \text{ iff } \sigma^{(1)} \models w$$

For a discussion of the other temporal operator and formal definitions of the usual operators and quantification, see [Manna & Pnueli 83b].

Vectors are represented using an italic notation, thus:

y represents the vector y

When trying to prove programs correct using the TL approach, it is necessary to construct a temporal proof system (TPS) for the programming language in which the programs are written. Such a TPS consists of three parts. The first is the uninterpreted logic part, which defines the general axioms for our first order TL language. This part does not alter from language to language, and is defined in [Manna 82, Manna & Pnueli 83b].

The second part is the domain part, in which the domains of the programming language must be defined, and any induction rules over those domains given.

A set A with an ordering relation $<<$ is said to be well founded if there exists no infinitely decreasing sequence

$$\alpha_0 << \alpha_1 << \alpha_2 << \ldots$$

If $(A, <<)$ is a well founded set and $w(\alpha)$ a temporal formula dependent on a parameter $\alpha \in A$, then some induction rules vital to proving the termination of any iterative construct can be defined. Such a rule vital to the theory to be developed is the $\Diamond$IND rule, which can be stated as:

$$\frac{w(\alpha) \rightarrow \Diamond[\psi \vee \exists\beta(\beta << \alpha \wedge w(\beta))]}{w(\alpha) \rightarrow \Diamond\psi}$$

An example of a rule which can be deduced from $\Diamond$IND is the rule IND from [Manna 82] for integers, which will be used in the proofs in this thesis.

$$\frac{Q(0) \rightarrow \Diamond\psi \qquad Q(m+1) \rightarrow \psi \vee \Diamond Q(m)}{Q(k) \rightarrow \Diamond\psi}$$

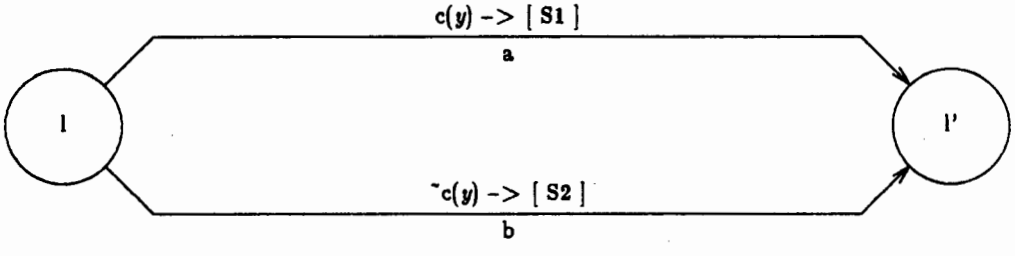
If Q is a predicate associated with a loop, and ψ is the termination condition of the loop then this rule can be used to show loop termination.

The third part of a TPS is the program part, in which the semantics of the programming language are given as a set of axiom templates. In order to understand why this is done, and the whole approach to the program part and the proofs themselves, it is necessary to consider briefly the object on which a proof operates.

When using a TPS, a program is represented as a directed graph. No declarations are shown, and the control abstraction of the program is abstracted into the shape of the graph. An IF statement in a graph would therefore have two edges coming out of a common vertex, one for the THEN part of the IF and the other for the ELSE part. Both edges would join again at a common edge vertex representing the end of the IF statement.

The edges of the graph are each labeled and tagged by a guarded command. The guard represents the condition that must be met for that particular edge to be taken. The statement part of the guarded command may be either a null or an assignment, as these are the only statements left once the control has been abstracted into the shape of the graph. For example, consider the following IF statement and its representation as a graph:

IF c(*y*) THEN S1 ELSE S2



Here, a and b are the edges representing the THEN and ELSE parts of the IF statement respectively. Both have a source vertex of l and sink vertex l'. If S2 was a null statement then this would be represented by a period '.'.

The proof proceeds by assuming that initially we are 'at the first vertex of the graph', and then demonstrating formally that eventually the last vertex of the graph must be reached, with a certain predicate holding.

The TPS approach makes this very simple, and even loop termination is made straightforward by the IND rules described above.

Note that the guards on the edges radiating from a particular vertex must be deterministic.

In order that the notion of 'at a vertex in the graph' can be made more formal, the notion of location variables is introduced. A location variable can point to a location on the graph, and allows concise indication that we are at a particular location, eg 'at l' points to the start vertex of the IF statement subgraph above.

Once a graph has been drawn, the effect of each edge (or transition) in the graph can be axiomatised. These axioms are called transition axioms and reflect exactly the effect of taking a particular transition in a graph.
For example, the IF statement above would have transition axioms:

$$F_a: [\text{at } l \wedge c(y) \wedge y{=}u] \rightarrow \Diamond[\text{at } l' \wedge y{=}F(u)]$$

$$F_b: [\text{at } l \wedge \tilde{}c(y) \wedge y{=}u] \rightarrow \Diamond[\text{at } l' \wedge y{=}G(u)]$$

(assuming that S1 is $y := F(y)$ and S2 is $y := G(y)$)

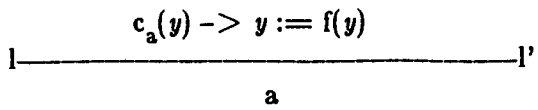
Each transition axiom has a name which relates it to the edge on the graph to which it refers. A transition says that if we are at a particular vertex of the graph, and a certain set of conditions holds, then sometime we will be at some other vertex of the graph with some other set of conditions holding.

In this case, for edge a it can be seen that if we are at l, and $c(y)$ holds, and y is equal to some set of global auxiliary variables u, then sometime we will be at l' and y will have mutated according to the function F. A similar interpretation can be found for the transition axiom for edge b.

The set of global auxiliary variables u used in the transition axioms is a standard trick made necessary by the fact that local variables are subject to mutation from state to state and the fact that these formulae are time dependent. One can not therefore speak meaningfully of '$y{:=}f(y)$' because of the time mutation, as the y on the right of the $:=$ sign must have the same value as the y on the left in the same state in a TPS. Such an assignment is therefore only meaningful if f is an identify function. The global variables therefore 'freeze' the values of y in one state so that the assignment function can have a meaningful application.

In fact the transition axiom schema used here is called weak transition axioms because they use the sometimes operator $\Diamond$, as opposed to a strong axiom which would use the nexttime operator O.

All the edges of a graph can be formalised as axioms in such a way. The general edge will be represented by:

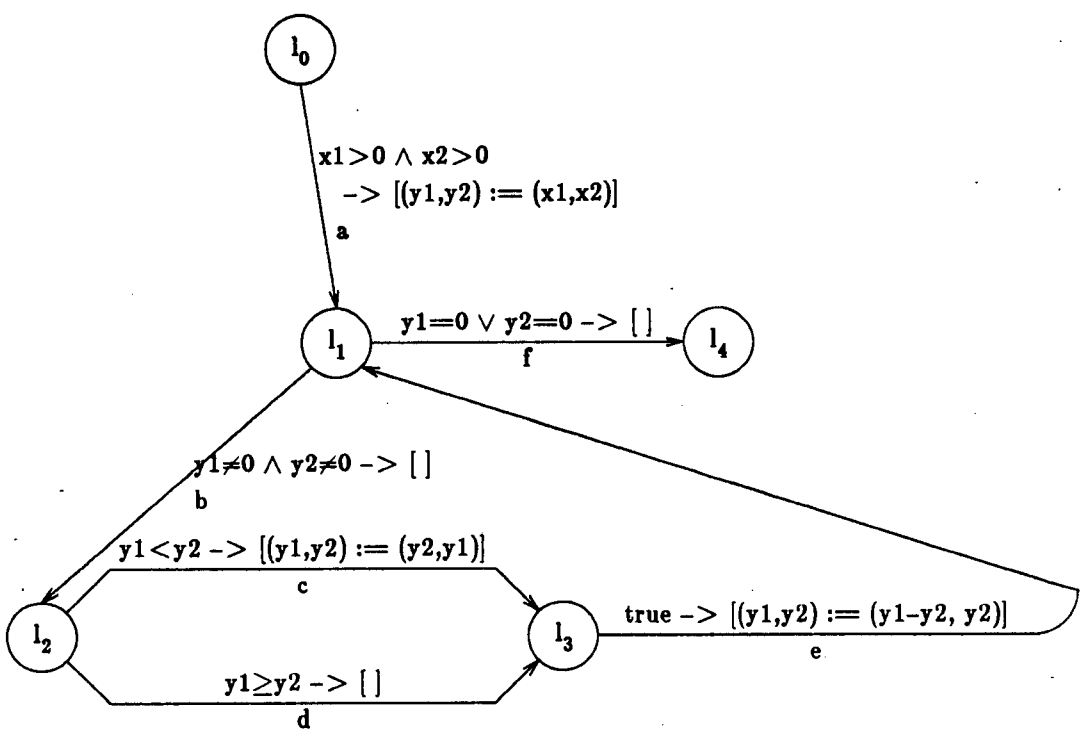


The purpose of the program part of a TPS is to define the axiomatics of the programming language for which the TPS is being built. This includes giving templates for graph construction and defining the statements that could label the command part of the guarded commands on the edges of the graph.

AN EXAMPLE

Let us now prove a simple program correct in order to get a grasp of how temporal logic is used in proofs. A similar example will be used to illustrate the verification of recursive programs.

In this and all other examples, a rulename or theorem given as a justification of a line of the proof which is not defined in this thesis can be found in [Manna 82].



GCD - iterative version

Consider the graph of a program to calculate the GCD of two positive integers x1 and x2. The transition axioms for the proof are:

F_a: [at l_0 ∧ x1>0 ∧ x2>0] -> ◇[at l_1 ∧ (y1,y2)=(x1,x2)]

F_b: [at $l_1 \wedge y1 \neq 0 \wedge y2 \neq 0 \wedge y=u$] -> $\Diamond$[at $l_2 \wedge y=u$]

F_c: [at $l_2 \wedge y1 < y2 \wedge y=u$] -> $\Diamond$[at $l_3 \wedge y1=u2 \wedge y2=u1$]

F_d: [at $l_2 \wedge y1 \geq y2 \wedge y=u$] -> $\Diamond$[at $l_3 \wedge y=u$]

F_e: [at $l_3 \wedge y=u$] -> $\Diamond$[at $l_1 \wedge y1=u1-u2 \wedge y2=u2$]

F_f: [at $l_1 \wedge (y1=0 \vee y2=0) \wedge y=u$] -> $\Diamond$[at $l_4 \wedge y=u$]

The program is proved correct with respect to the following assertions:

ϕ: [at $l_0 \wedge x1>0 \wedge x2>0$]

ψ: [at $l_4 \wedge gcd(x1,x2)=max(y1,y2)$]

The proof is in two parts

(a) ϕ -> $\Diamond \exists k.Q(k,y)$
(b) $Q(k,y)$ -> $\Diamond \psi$

ϕ -> $\Diamond \psi$ by $\Diamond Q$ and $\exists \Diamond Q$ rules of [Manna 82]

Where $Q(k,y)$ is

[at $l_1 \wedge 0 \leq (y1 * y2) \leq k \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge gcd(y1,y2)=gcd(x1,x2)$]

The bound function is (y1*y2). This will be 0 when either y1 or y2 is 0, and decreases with each alteration to y1 or y2 in the program, and is therefore a suitable bound function for this program.

Proof of (a):

1. ϕ -> [at $l_0 \wedge (x1,x2)>0$]
   by definition

2. [gcd(x1,x2) = gcd(x1,x2)]
   by domain

3. [at $l_0 \wedge x1>0 \wedge x2>0$]
   -> [at $l_0 \wedge x1>0 \wedge x2>0 \wedge gcd(x1,x2) = gcd(x1,x2)$]
   by PR

4. [at $l_0 \wedge x1>0 \wedge x2>0 \wedge gcd(x1,x2) = gcd(x1,x2)$]

$-> \diamond[at\ l_1 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge y1 = x1 \wedge y2 = x2 \wedge gcd(x1,x2) = gcd(y1,y2)]$
by F_a, PR

5. $[y1 \geq 0 \wedge y2 \geq 0] -> [y1 * y2 \geq 0]$
by PR

6. $[y1 * y2 \geq 0] -> [0 \leq (y1 * y2) \leq (y2 * y2)]$
by domain

7. $[at\ l_1 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge y1 = x1 \wedge y2 = x2 \wedge gcd(y1,y2) = gcd(x1,x2)]$
$-> [at\ l_1 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge 0 \leq y1*y2 \leq y1*y2 \wedge gcd(x1,x2) = gcd(y1,y2)]$
by PR, domain

8. $[at\ l_1 \wedge 0 \leq y1*y2 \leq y1*y2 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge gcd(x1,x2) = gcd(y1,y2)]$
$-> \exists k.[at\ l_1 \wedge 0 \leq y1*y2 \leq k \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge gcd(y1,y2) = gcd(x1,x2)]$
by T24 of [Manna 82]

9. $\exists k.[at\ l_1 \wedge 0 \leq y1*y2 \leq k \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge gcd(x1,x2) = gcd(y1,y2)]$
$-> \exists k.Q(k,y)$
by def'n of Q

10. $\phi -> \diamond \exists k.Q(k,y)$
by $\diamond \exists Q$ rule of [Manna 82]

Which completes the proof of (a)

The proof of (b) is in two parts:

(b1) $Q(0,y) -> \diamond \psi$
(b2) $Q(m+1,y) -> \diamond \psi \vee Q(m,y)$

$Q(k,y) -> \diamond \psi$ by IND rule defined above.

Proof of (b1)

11. $Q(0,y) \rightarrow [\text{at } l_1 \wedge 0 \leq y1*y2 \leq 0 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \gcd(y1,y2)=\gcd(x1,x2)]$

12. $[0 \leq y1*y2 \leq 0] \rightarrow [y1=0 \vee y2=0]$
by PR

13. $[y1=0 \vee y2=0] \rightarrow [\gcd(y1,y2)=\max(y1,y2)]$
by def'n of gcd

14. $[\text{at } l_1 \wedge y1*y2=0 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \gcd(y1,y2)=\gcd(x1,x2)]$
$\rightarrow [\text{at } l_1 \wedge (y1=0 \vee y2=0) \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \gcd(x1,x2) = \max(y1,y2)]$
by 12,13,PR

15. $[\text{at } l_1 \wedge (y1=0 \vee y2=0) \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \gcd(x1,x2)=\max(y1,y2) \wedge y=u]$
$\rightarrow \Diamond[\text{at } l_4 \wedge y=u \wedge \gcd(x1,x2)=\max(u1,u2)]$
by F_f, PR

16. $[\text{at } l_4 \wedge \gcd(x1,x2)=\max(y1,y2)] \rightarrow \psi$
by PR

17. $Q(0,y) \rightarrow \Diamond\psi$
by 11,14,15,16,$\Diamond$Q

Which concludes the proof of (b1)

Proof of (b2)

18. $Q(m+1,y) \rightarrow [\text{at } l_1 \wedge 0 \leq y1*y2 \leq m+1 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \gcd(y1,y2)=\gcd(x1,x2)]$
by PR

There are two cases to consider.

CASE 1: $y1*y2 = 0$

19. $[\text{at } l_1 \wedge o \leq y1*y2 \leq 0 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \gcd(y1,y2)=\gcd(x1,x2)]$
$\rightarrow \Diamond\psi$

by 11,14,15,16,$\diamond$Q

20. $Q(m+1,y) \wedge y1*y2=0 \rightarrow \diamond\psi$

by PR

CASE 2: $y1*y2 > 0$

21. $[y1 * y2 > 0] \rightarrow [y1 > 0 \wedge y2 > 0]$

by PR

22. $[\text{at } l_1 \wedge 0 \leq y1*y2 \leq m+1 \wedge y1*y2>0 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \gcd(x1,x2)=\gcd(y1,y2)]$
$\rightarrow [\text{at } l_1 \wedge 0 \leq y1*y12(<=m+1 \wedge y1>0 \wedge y2>0 \wedge \gcd(x1,x2)=\gcd(y1,y2)]$

by PR

23. $[\text{at } l_1 \wedge 0 \leq y1*y2 \leq m+1 \wedge y1>0 \wedge y2>0 \wedge y=u \wedge \gcd(y1,y2)=\gcd(x1,x2)]$
$\rightarrow \diamond[\text{at } l_2 \wedge 0 \leq y1*y2 \leq m+1 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge y=u \wedge \gcd(x1,x2)=\gcd(y1,y2)]$

by F_b, PR

There are now two subcases:

SUBCASE 1: $y1 < y2$

24. $[\text{at } l_2 \wedge 0 \leq y1*y2 \leq m+1 \wedge y1<y2 \wedge y=u \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \gcd(x1,x2)=\gcd(y1,y2)]$
$\rightarrow \diamond[\text{at } l_3 \wedge 0 \leq y1*y2 \leq m+1 \wedge y1=u2 \wedge y2=u1 \wedge y1 \geq y2 \wedge y1 \geq 0 \wedge y2 \geq 0$
$\wedge \gcd(y1,y2)=\gcd(x1,x2)]$

by F_c

SUBCASE 2: $y1 \geq y2$

25. $[\text{at } l_2 \wedge 0 \leq y1*y2 \leq m+1 \wedge y1 \geq y2 \wedge y=u \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \gcd(x1,x2)=\gcd(y1,y2)]$
$\rightarrow \diamond[\text{at } l_3 \wedge 0 \leq y1*y2 \leq m+1 \wedge y=u \wedge y1 \geq y2 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \gcd(x1,x2)=\gcd(y1,y2)]$

by F_d

End of subcases.

26. $[y1 \geq y2 \wedge y1 \geq 0 \wedge y2 \geq 0] \rightarrow [y1-y2 \geq 0]$
by domain

27. $[0 \leq y1*y2 \leq m+1 \wedge y1 \geq y2 \wedge y1 \geq 0 \wedge y2 \geq 0] \rightarrow 0 \leq (y1-y2)*y2 \leq m]$
by PR

28. $[at\ l_3 \wedge 0 \leq y1*y2 \leq m+1 \wedge y2 \geq y2 \wedge y = u \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge gcd(y1,y2)=gcd(x1,x2)]$
$\rightarrow \Diamond[at\ l!l1 \wedge 0 \leq y1*y2 \leq m \wedge y1 = u1-u2 \wedge y2 = u2 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge gcd(y1,y2)=gcd(x1,x2)]$
by F_e, 26,27,PR, def'n of gcd.

29. $[at\ l_1 \wedge 0 \leq y1*y2 \leq m \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge gcd(x1,x2)=gcd(y1,y2)] \rightarrow Q(m,y)$

30. $Q(m+1,y) \wedge y1*y2 > 0 \rightarrow \Diamond Q(m,y)$
by 18,23,24,25,28,29,$\Diamond$Q

31. $Q(m+1,y) \rightarrow \Diamond\psi \vee \Diamond Q(m,y)$
by 20,30,PR

which completes the proof.

Note that this proof is unnecessarily lengthened by including many intermediate steps which could be combined together, as will be done in the example of the recursive version of the program, in chapter 13 and also in the proofs of AML code in chapters 16, 17 and 18.

Chapter 11: SOME GRAPH THEORY

A final foundation stone which must be laid before starting to extend the proof theory to cope with subprograms is to give a few definitions of what some terminology which relates to graphs means. A background understanding of graphs, such as [Gill 76] is assumed.

The START VERTEX SV of a graph is the first vertex of the graph, ie the one from which execution begins, and which is the sink vertex for no edges.

The HALT VERTEX HV of a graph is the last vertex of the graph, ie the one at which execution terminates, and which is the source vertex for not edges. (In fact, since TL requires an infinite series of states in the model, the halt vertex has one self loop, and a program halting is formally represented by an endless null idle, but this is never actually represented on a graph or taken into practical account when building a proof).

Every edge in the graph has a SOURCE VERTEX, which is the vertex in which it originates, and a SINK VERTEX, which is the vertex in which it ends.

A PATH through the graph from vertex α to vertex β is represented as PATH(α,β), and is defined as a finite sequence of edges such that the first edge in the sequence has source vertex of α, the last edge in the sequence has sink vertex β, and the sink vertex of every other edge is the start vertex of the next edge in the sequence. A path may include cycles. α is called the source vertex of the path, and β the sink vertex of the path.

It is possible that there may be more than one possible PATH(α,β) in a graph. A set of paths PATH(α,β) is called a modular path from α to β and represented by M-PATH(α,β). Not all possible paths from α to β need to be included in the M-PATH.

A COMPLETE PATH through a graph has the source vertex of the graph SV as the start vertex of the path and the halt vertex HV of the graph as the final vertex of the path. A COMPLETE M-PATH may be analogously defined.

Any path p has an EDGE SET E_p which consists of all the edges in the path, and a VERTEX SET V_p which consists of all the vertices in the path.

A vertex enumeration sequence N_p may be constructed for a p=PATH(α,β) by starting from α, with only α in the sequence, and traversing the path to β, placing each vertex into the sequence as it is encountered. If the PATH(α,β) is cyclic, then some vertices could appear in the sequence more than once. There can thus be an infinite number of possible vertex enumeration sequences N_p for any PATH(α,β) which has cycles, although each N_p must be finite.

For a sequence of vertices

$$N_p = v_0, v_1, v_2, \ldots, v_n$$

the i-shifted sequence of vertices is denoted by

$$N_p^{(i)} = v_i, v_{i+1}, \ldots, v_n$$

A vertex v_i can then be said to precede a vertex v_j on a path p if

$$\exists N_p: [v_i, v_j \in N_p \wedge v_j \in N_p^{(i+1)}]$$

where the meaning of the elementhood symbol $\in$ is bent to include membership of a sequence.

Ordering between vertices on a path can now be exactly defined in terms of the above concepts. If the symbol $>>>$ is used to mean 'ordered greater than' as a binary operator on the vertices on a path p, and $<<<$ to mean 'ordered less than', then

$$\alpha <<< \beta$$

means that a is ordered less than b on p if α precedes β on p, and β does not precede α on p, and

$$\alpha >>> \beta$$

means that a is ordered greater than b on p if β precedes α on p, and α does not precede β.

Formally,

$$\alpha <<< \beta \text{ iff } \exists p: [\alpha, \beta \in V_p \wedge \alpha \text{ precedes } \beta \wedge \tilde{}(\beta \text{ precedes } \alpha)]$$

and

$$\alpha >>> \beta \text{ iff } \exists p: [\alpha, \beta \in V_p \wedge \beta \text{ precedes } \alpha \wedge \tilde{}(\alpha \text{ precedes } \beta)]$$

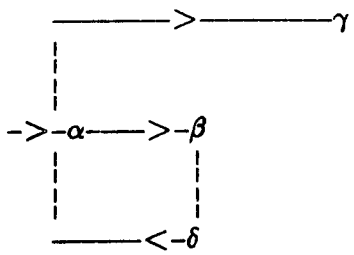
where p is a path.

Two additional operators can also be defined:

$$\alpha <<<= \beta \text{ iff } \alpha <<< \beta \vee \alpha = \beta$$
$$\alpha >>>= \beta \text{ iff } \alpha >>> \beta \vee \alpha = \beta$$

The rather strange caveat on the definitions of $<<<$ and $>>>$ that α precede β and not β precede α, etc is necessary in order that the definitions be sufficiently rigorous on graphs with cycles.

Consider the graph



We can say that $\alpha >>> \gamma$, and that $\beta >>> \gamma$. Also, we can say $\beta >>> \delta$ because there exists a path

$$\{\alpha, \beta, \delta, \alpha, \gamma\}$$

with one iteration of the cycle. However there is no meaningful ordering between α and β. Despite this, the definitions we have are sufficient for the purposes of this thesis.

These definitions and notations will be used when building the proof system for recursive programs.

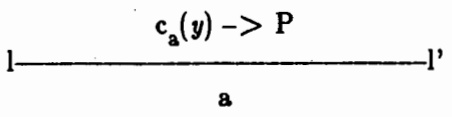
Chapter 12: COPING WITH SUBPROGRAMS

The first chapter of this part of the thesis (chapter 10) introduced the TPS of Manna and Pnueli. This chapter extends their work to cope with subprograms.

Procedures and functions have their own preconditions, postconditions and graphs, representing the code in the body of the subprogram. They can therefore be proved separately from the rest of the program.

Once a procedure has been proven correct, then verifying a call of the procedure requires only a demonstration that the precondition holds in the context of the call, and then deducing that the call must terminate with the effect modeled by the postcondition.

Up until this point, the only permissible bodies of the guarded command decorating an edge of a transition graph have been an assignment command or a null command. This set of commands is extended to include a procedure call command. This command consists of the procedure name followed by a list of actual parameters. In other words, a procedure call looks exactly like one would expect a call to do in a Pascal–like programming language. Diagrammatically a call of a procedure P can be represented by:



where $c_a(y)$ is the guard of edge a, and P means a call of procedure P.

The semantics of procedure call and return can be defined in terms of a stack L of location variables, with operations $|L|$ meaning the size of the stack, and top(L) returning the top element of the stack.

A procedure call is then:

$$[\text{at } l \wedge c_a(y) \wedge \text{PRE}_p(y) \wedge |L|{=}s] \rightarrow \Diamond[\text{at } l_{p0} \wedge |L|{=}s{+}1 \wedge \text{top}(L){=}l']$$

which states that, if we are at l, with PRE_p, the precondition of procedure P holding, and the size of L is s, where s is an auxiliary global variable, then sometime we will be at l_{p0}, the first location of the graph of procedure P, with the size of L increased by 1 and the top of L will be the sink vertex of the edge from which the call was made.

A return can be defined by:

$$[\text{at } l_{ph} \wedge \text{POST}_p(y) \wedge \text{top}(L){=}l' \wedge |L|{=}s] \rightarrow \Diamond[\text{at } l' \wedge |L|{=}s{-}1]$$

where l_{ph} is the halt vertex of the graph of procedure P, and POST_p is the postcondition of procedure p.

This model can cope with parameterless procedures and FORTRAN–like scoping. The model may now be extended to cope with parameters and scoping.

Scoping rules are not an issue, as a suitable renaming convention can transform any scoped language into a static language. Then the only requirement is that at procedure entry time the local variables of the procedure be initialised (usually to the undefined value) by some implicit assignment statement.

If y_l is the set of local variables of procedure p in a language where local variables are undefined at scope entry time, such as Pascal, then the procedure call semantic can be changed to:

$$[\text{at } l \wedge c_a(y) \wedge \text{PRE}_p(y) \wedge |L|=s]$$
$$\text{-> } \Diamond[\text{at } l_{p0} \wedge |L|=s+1 \wedge \text{top}(L)=l' \wedge y_l=\text{undefined}]$$

Parameters are a little more complicated. The introduction of parameters and scoping means that pre and post conditions of a procedure are liable to be expressed in terms of the formal parameters of the procedure. The notation 'a/b' is introduced to mean a replaced by b.

The easiest sort of parameters to handle are value and result parameters. If y_v are the value parameters of procedure p, and y_r are the result parameters, and a_v are representations of the actual value parameters and a_r are the actual result parameters to a call, then procedure call becomes:

$$[\text{at } l \wedge c_a(y) \wedge \text{PRE}_p(y_v/a_v) \wedge a_v=u \wedge |L|=s]$$
$$\text{-> } \Diamond[\text{at } l_{p0} \wedge |L|=s+1 \wedge \text{top}(L)=l' \wedge y_v=u \wedge y_l=\text{undefined} \wedge y_r=\text{undefined}]$$

which now says that if we are at l and $c_a(y)$ holds, and the precondition holds with the formal placemarkers filled in by the actual parameter values, and the size of L is s, then sometime we will be at l', with the size of L increased by 1, the sink vertex of edge a on the top of L, the formal value parameters set to their corresponding actuals, and the local variables and result parameters undefined. If another convention applied to initial and final parameter values then this could be modeled just as simply.

The return semantic becomes:

$$\text{-> } \Diamond[\text{at } l' \wedge |L|=s-1 \wedge a_r=u]$$

A standard restriction [Hoare 71, Hoare & Wirth 73, Igarashi et al 75] on anonymous aliasing is introduced. Such aliasing is disallowed. This is specifically a problem with actual parameters, as one actual parameter variable can be made to refer to several

formal parameters, causing problems for verification. For example, consider this program segment (adapted from [Igarashi et al 75]):

```
PROCEDURE B(RESULT a, b : integer) ;
PRE: true
POST: a=1 ∧ b=2
 begin
  a :=1 ; b := 2 ;
 end ;
```

This procedure may easily be verified. However, consider a call B(q,q). In verifying this call, we would be forced to deduce the postcondition (in terms of actual parameters) $q = 1 \wedge q = 2$, which is an obvious contradiction. For this reason, anonymous aliasing is disallowed. Explict aliasing, such as renaming of variables, can be handled in this proof system, as shall be discussed below.

If the model is extended to include value-result parameters, a new category of parameters y_f is introduced, and the the semantics of call and return are extended as given below. Note that the PRE and POST conditions are extended to take an additional parameter, the value-result parameter list. a_f represents the value-result actual parameters.

$[\text{at } l \wedge c_a(y) \wedge \text{PRE}_p(y_v/a_v, y_f/a_f) \wedge |L|=s \wedge a_f=u1 \wedge a_v=u2]$
$\rightarrow \Diamond[\text{at } l_{p0} \wedge |L|=s+1 \wedge \text{top}(L)=l' \wedge y_v=u2 \wedge y_l=\text{undefined} \wedge y_r=\text{undefined} \wedge y_f=u1]$

$[\text{at } l_{ph} \wedge \text{POST}_p(y_v, y_r, y_f) \wedge \text{top}(L)=l' \wedge |L|=s \wedge y_r=u1 \wedge y_f=u2]$
$\rightarrow \Diamond[\text{at } l' \wedge |L|=s-1 \wedge a_r=u1 \wedge a_f=u2]$

There are two remaining issues to consider, namely side effects and reference parameters. Side effects are not in general disallowed in this system. However, *implicit* side effects (such as those introduced by an actual parameter vector as discussed above) are disallowed. In other words, any side effect which can be explicitly modeled, such as renaming of variables, will be allowed, but implicit side effects are not. Any side effects should be modeled in the pre and postconditions of a procedure in order that the verification be meaningful.

If y_s is the set of variables affected by the side effects of a procedure, then the semantics of procedure call and return can be modified to become:

$[\text{at } l \wedge c_a(y) \wedge \text{PRE}_p(y_v/a_v, y_f/a_f, y_s) \wedge |L|=s \wedge a_f=u1 \wedge a_v=u2]$
$\rightarrow \Diamond[\text{at } l_{p0} \wedge |L|=s+1 \wedge \text{top}(L)=l' \wedge y_v=u2 \wedge y_l=\text{undefined} \wedge y_r=\text{undefined} \wedge y_f=u1]$

$[\text{at } l_{ph} \wedge \text{POST}_p(y_v, y_r, y_f, y_s) \wedge \text{top}(L)=l' \wedge |L|=s \wedge y_r=u1 \wedge y_f=u2]$

$-> \diamond$[at l' $\wedge$ |L|=s-1 $\wedge$ a_r=*u1* $\wedge$ a_f=*u2*]

Reference parameters present an interesting problem, as for verification purposes a reference parameter introduces an alias between the formal and actual reference parameters. Then, any change to one automatically changes the other.

In order to handle reference parameters we must modify our view of assignment. We introduce the concept of an ALIAS SET. Each variable has an alias set. Then, a change to a variable modifies the variable and all the elements of that variable's alias set (and all the elements of their alias sets, and so on). For any variable y, its alias set is designated A_y. An assignment of the form y:=e can then be understood to have the underlying definition:

```
ASSIGN(y,e) : T := {} ;
              PUTVALUE(y,e)

PUTVALUE(y,t) : y := t ;
                T := UNION(T,y)
                ∀x:[x∈A_y ∧ ~(x∈T)]:PUTVALUE(x,t)
```

Here, T is an auxiliary set, used to prevent infinite regressions if alias sets happen to end up being recursive. It is initially set to the empty set, and PUTVALUE (an auxiliary function) is called, which takes as argument a variable and a value, assigns the value to the variable, places the variable into T, and then assigns the same value to all the members of the variable's alias set.

(One of the most attractive aspects of this method of handling reference parameters is that the alias set technique can be applied to any form of aliasing. Thus Ada renames clauses, for example, can be handled in exactly the same way.)

The procedure call and return can be extended to reflect reference parameters. The formal parameters are denoted by y_a, and the actual parameters by a_a. The pre and postconditions are once again altered to reflect the new parameter form.

[at l $\wedge$ $c_a(y)$ $\wedge$ $PRE_p(y_v/a_v, y_f/a_f, y_s, y_a/a_a)$ $\wedge$ |L|=s $\wedge$ a_f=*u1* $\wedge$ a_v=*u2* $\wedge$ a_a=*u3*]
 $-> \diamond$[at l_{p0} $\wedge$ |L|=s+1 $\wedge$ top(L)=l' $\wedge$ y_v=*u2* $\wedge$ y_l=undefined
 $\wedge$ y_r=undefined $\wedge$ y_f=*u1* $\wedge$ y_a=*u3*]

[at l_{ph} $\wedge$ $POST_p(y_v, y_r, y_f, y_s, y_a)$ $\wedge$ top(L)=l' $\wedge$ |L|=s $\wedge$ y_r=*u1* $\wedge$ y_f=*u2* $\wedge$ y_a=*u3*]
 $-> \diamond$[at l' $\wedge$ |L|=s-1 $\wedge$ a_r=*u1* $\wedge$ a_f=*u2* $\wedge$ a_a=*u3*]

and, of course, at procedure entry time all actual reference parameters are placed in the alias sets of the corresponding formals, and vice versa, and removed from the alias sets at procedure return time, and vice versa. Similarly, these bindings are broken at procedure exit time.

Some notation is also needed to indicating the initial value of a parameter in a procedure. A variable by a 0, as in y_0 refers to the value of the parameter at the start of the most recent call. Subscripting with a c, as in y_c, refers to the value of the parameter at the start of the first call of the procedure. For non–recursive procedures, these values will be the same, but for recursive procedures, y_c will continue to give the first value, whilst, y_0 will alter from call to call. A variable with a 0 or c subscript is considered a global variable for verification purposes.

A PROOF RULE FOR PROCEDURE CALL

Now that we have a formal understanding of what a procedure call means, a proof rule to use in verifying calls can be formulated. This rule is called PCALL, and has the form:

PCALL:

1. at l $\wedge$ c(y)
2. PRE(y_v/a_v, y_f/a_f, y_s, y_a/a_a)

$\diamond$[at l' $\wedge$ POST(y_v, y_r/a_r, y_f/a_f, y_s, y_a/a_a)]

This rule says that, if we are at l, the source vertex of an edge decorated by a call, and the enabling condition of the edge holds, and the precondition of the procedure being called holds in the context of the call, then we can deduce that sometime we will be at the sink vertex of the edge, and the postcondition of the procedure will hold. The procedure being called is assumed to have been verified previously. The restrictions on actual variables appearing only once in a parameter list defined above apply.

This rule cannot be used from inside a procedure for a recursive call.

HOW TO HANDLE FUNCTIONS

The traditional view of procedures is that they are functions which return no parameter. In this thesis, however, we follow the view of [Wulf et al 81], where a function is a procedure which returns an additional (possibly invisible) parameter, namely the result.

This view allows us to treat function calls as an extension of the theory for procedure calls just developed above. A function call can only occur on the righthand side of an

assignment (parameter passing and return are treated as assignments in this case). If the lefthand side variable in the assignment is denoted y_l, and the the return value of the function is always contained in a variable constructed of the name of the function subscripted with an r, then the function call rule FCALL as may be defined as:

FCALL:

1. at l $\wedge$ c(y)
2. PRE(y_v/a_v, y_f/a_f, y_s, y_a/a_a)

$\diamond$[at l' $\wedge$ POST($y_v, y_r/a_r, y_f/a_f, y_s, y_a/a_a$) $\wedge$ y_l=f_r]

where we have assumed that the name of the function being called using FCALL is just f. The function result value is not explicitly shown as a parameter.
For example, given an edge:

c(y) -> y1 := func(y2,y3)

l————————————————————————————l'

Func can be shown to operate correctly as a separate exercise. Then, if it can be shown that

[at l $\wedge$ c(y) $\wedge$ PRE(f1/y2, f2/y3)]

where f1 is the first formal parameter and f2 the second, and PRE the precondition of the function, then it can be deduced that:

$\diamond$[at l' $\wedge$ y1=func_r]

where func_r is the returned value of the function with name func.

In general, the return value variable of the function may be assigned to in an explicit fashion. In languages such as Pascal, the return value of a function is defined by assigning the value to the name of the function. In other languages, such as Ada, the return value is passed back via a RETURN statement. In the latter case a special semantic defining the effect of the RETURN statement must be defined along with the semantics of all the other statements in the language, where the effect of the semantic is to assign to the return value variable the return value of the function. (In Ada the RETURN will also cause an exit from the function but this is not relevant here).

The only thing to be careful of is how to handle functions which have side effects being used as operands in complicated expressions. This is not so much a verification issue as a software engineering issue; the system as developed can handle it, but things are apt to get confusing. In that case the approach to take might be to break the expression down

into a number of subexpressions with assignment to temporary variables to simplify matters.

The restrictions on actual variables appearing only once in a parameter list defined above apply.

As with procedures, FCALL cannot be used from within a function for a recursive call.

A RELAXING OF CONVENTION

This chapter has concentrated on building a generic TPS capable of handling any generalised notion of procedure or function call. All the formalism developed here is not strictly needed in the examples that will be coming up in the rest of the thesis. The following simplifications will therefore be used:

Only value–result parameters will be used, so the notion of assignment by alias can be disregarded.

In pre and postconditions parameters will be listed as required as opposed to by category, as has been the implication above.

As AML procedures and functions operate largely by side–effect, we will be working with side effects. They will be modeled in the pre and post conditions as required.

To save on the typography, when a recursive call is made, the saving of the sink vertex of the edge from which the call was made on the location stack, or the increase of the location stack will not explicitly be shown. This is permissible because a case analysis of the possible recursive return points will be performed as part of the proof.

Chapter 13: COPING WITH RECURSION

This chapter builds on the work chapters 10 through 12 and shows how recursive subprograms can be proven correct using a TPS.

The theoretical similarity between loops and recursion is well known [Kfoury et al 82] but has not been exploited to simplify the proofs of recursive subprograms. Conventional proof rules for recursive programs are complex and based on recursive function theory. In this approach, the least fixpoint of a recursive function is calculated and then used for the verification [Manna 74]. Finding the least fixpoint involves rewriting the function to reduce it to a function which will be simpler to verify. For example, the least fixpoint of the function

$$F(x):\ \text{if } x = 0 \text{ then } 1 \text{ else } F(x+1)$$

is

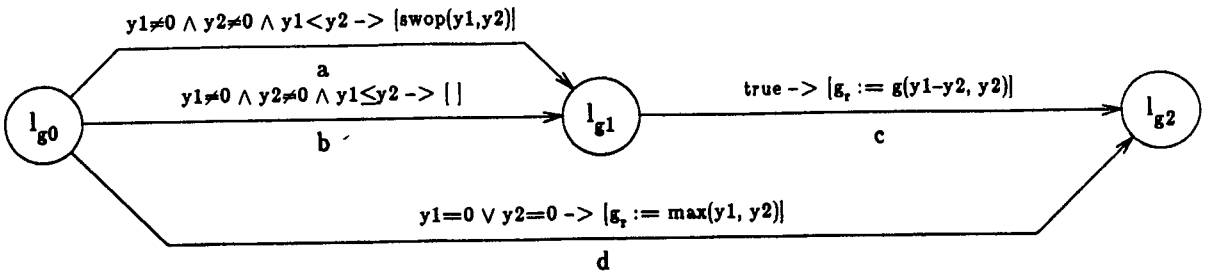
$$\text{if } x\ = 0 \text{ then } 1 \text{ else } \textit{undefined.}$$

because F(x) is undefined on any input except 0.

The process of constructing least fixpoints is described in detail in [Manna 74]. This is a far from trivial process, however, and for recursive AML functions, which terminate because of side effects and perform arbitrary walks on trees, explicit construction of the fixpoint would be extremely difficult. Instead, we propose the proof of correctness of a recursive program by using the approach to iteration described above to prove that the recursive program converges.

In this section of the thesis we illustrate our approach with the proof of correctness of a simple example. This methodology will be applied to the proof of recursive AML programs in part III of the thesis.

Consider a graph of a recursive subprogram, such as this function g which calculates the GCD of two positive integers. g_r is the result of the function and is not explicitly shown



Graph of function g

as a parameter, and y1 and y2 are value parameters. The precondition ϕ_g and postcondition ψ_g are:

$$\phi_g : y1 \geq 0 \wedge y2 \geq 0 \wedge y1_c > 0 \wedge y2_c > 0$$
$$\psi_g : g_r = \gcd(y1_c, y2_c)$$

Procedure swop takes two formal value–result parameters y3 and y4, and has pre and postconditions:

$$\phi_{swop} : \text{true}$$
$$\psi_{swop} : y3 = y4_c \wedge y4 = y3_c$$

If we consider this graph, we can see that for the recursion to finish, eventually edge d of the graph must be taken, as it comprises the only path in the graph which does not have an edge labeled with a recursive call. The whole proof will revolve around showing that eventually this edge must be taken.

If we are at l_{g0}, and either one of y1 or y2 is zero, then we will take edge d and terminate. Otherwise, the values of y1 and y2 are adjusted so that the larger is in y1 and then call g again, passing it the parameters y1–y2 and y2. This adjustment of the parameter values still results in two numbers having the same GCD, but the number of recursions required to reach the GCD is reduced by 1. If neither of y1 or y2 is now zero at the recursive call, then the process is repeated, but must eventually terminate. This strategy is similar to that used in the previous example to prove a program to calculate GCD with a loop.

This is the important point of this chapter; that with the proper analysis, any recursive subprogram can be proven correct using the loop proof approach.

In general, to prove a recursive subprogram totally correct, there must exist at least one path through the graph which does not contain an edge with a recursive call. Such a path is called a NON RECURSIVE PATH (NRP) of the graph. Paths with edges which

contain recursive calls are called RECURSIVE PATHs (RP) of the graph.

When proving a loop correct using a rule such as IND, as we did above, the predicate Q has three components. One of these is a location (the 'hub' of the loop), the second a bound formula that must be shown to converge on the termination condition of the loop with each iteration, and the third an arbitrary predicate, which corresponds to the invariant of classical verification systems.

Equivalent location variables, bound functions and 'invariant' predicates must be identified for the recursive case.

We show in proving a loop that when the loop terminates, some condition γ will hold. γ consists of two parts, the location to be attained when the loop terminates, and a predicate which must hold when the location is attained. This location is called the terminal of the loop.

Consider a general graph of a recursive subprogram. There will be at least one complete path which is an NRP and one which is an RP. Tracing backwards from the call will eventually encounter a vertex common to the RP and the NRP. This distinguished vertex is called the CRITICAL VERTEX (CV) of that particular RP–NRP pair, and fulfills the same function as the 'hub' vertex of the loop.

The edge leading out of the CV on the NRP has a guarded command; this command is also distinguished as the CRITICAL CONDITION of the RP–NRP pair, and forms the basis for the construction of the bound function.

Tracing forwards from the call will eventually encounter a vertex which is common to both the RP and NRP. This distinguished vertex is the JOIN VERTEX of the RP–NRP pair, and is equivalent to the terminal vertex of a loop.

The 'invariant' and predicate associated with the termination condition are constructed in the same way as for loops.

There may be many possible RP–NRP pairs in a subprogram graph; each of these should in theory be considered as a separate set of entities and proved separately; in practice these proofs can be overlaid, especially if RPs and NRPs are constructed as M–PATHs.

The definition of recursive path, non–recursive path, critical vertex and join vertex can be formalised so that they can be unambiguously identified.

DEFINITION: A RECURSIVE PATH (RP) in a graph of a subprogram is a complete path where at least one edge in the path is decorated by a guarded command containing a call of the subprogram.

DEFINITION: A NON RECURSIVE PATH (NRP) in a graph of a subprogram is a complete path where no edge on the path is decorated by a guarded command containing a call of the subprogram.

DEFINITION: A vertex β is the CRITICAL VERTEX (CV) of a particular RP and NRP if it is the last vertex that the paths have in common prior to the edge in the RP containing the recursive call. If V_N is the vertex set of the NRP, V_R is the vertex set of the RP and α is the source vertex of the edge decorated by the call, then β is defined by

$$\beta \in V_N \wedge \beta \in V_R \wedge \neg(\exists \gamma:[\gamma \in V_N \wedge \gamma \in V_R \wedge \beta <<< \gamma <<< \alpha])$$

DEFINITION: The CRITICAL CONDITION of an NRP–RP pair is the enabling condition of the edge in the NRP which has as its source vertex the critical vertex of the NRP–RP pair.

DEFINITION: A vertex β is the JOIN VERTEX (JV) of a particular RP and NRP if it is the first vertex that the paths have in common after the edge in the RP containing the recursive call. If V_N is the vertex set of the NRP, V_R is the vertex set of the RP and α is the source vertex of the edge decorated by the call, then β is defined by

$$\beta \in V_N \wedge \beta \in V_R \wedge \neg(\exists \gamma:[\gamma \in V_N \wedge \gamma \in V_R \wedge \alpha >>> \gamma >>> \beta])$$

The $\diamond$IND–based proof rules can now be used to prove recursive programs correct, with the above interpretations used to identify the various distinguished vertices, conditions and edges.

There is, however, one simple proviso. In a loop, the terminal node of the loop is just another node in the program graph. When this vertex is reached, the proof continues in a normal fashion. When proving recursive programs the JV is used as the terminal node because, once it is reached, the proof can continue as normal. The problem is that once the JV has been reached using the strategy described above, the path from the sink vertex of the recursive call edge to the JV will not have been covered.

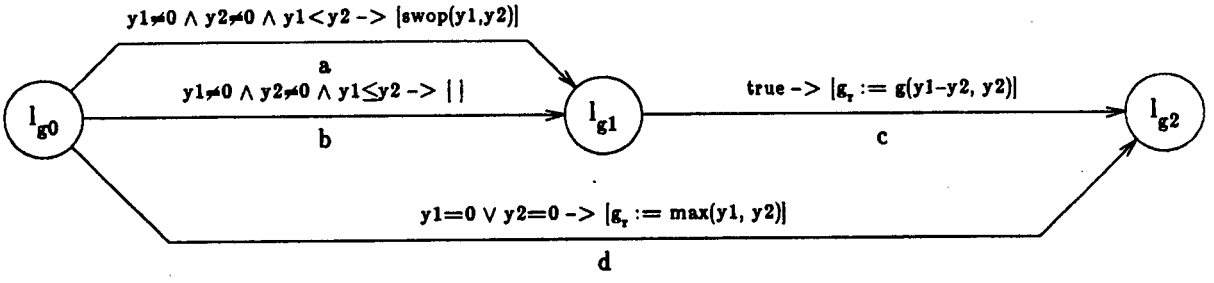
An additional part of the proof is therefore to conduct a case study on the possible call return points (ie sink vertices of recursive call edges) to complete the proof. In a tail recursive program, this is a null operation, but in programs which recurse back to another part of the graph, some additional work must be undertaken, namely proving the edges between the sink vertex of the recursive call edge and the JV.

The examples in part III of this thesis on verification of CFF/AML specifications include examples of non tail recursive procedures.

We now consider a proof of the function g to calculate GCDs, which was given at the start of this chapter.

AN EXAMPLE

We prove that the recursive function g introduced at the start of this chapter calculates the GCD of two positive integers correctly. This is a recursive version of the iterative program proven in chapter 10. The purposes of this example are to illustrate the application of the theory built in this part of the thesis, and to demonstrate that a proof of a recursive program using the techniques developed here is no more complex than proving an equivalent iterative program.



Graph of function g

The graph and conditions of g are repeated for ease of reference: g_r is the result of the function and is not explicitly shown as a parameter, and y1 and y2 are value parameters. The precondition ϕ_g and postcondition ψ_g are:

ϕ_g : $y1 \geq 0 \wedge y2 \geq 0 \wedge y1_c > 0 \wedge y2_c > 0$

ψ_g : at $l_{g2} \wedge g_r = gcd(y1_c, y2_c)$

The last two conjunctions on ϕ_g will be implicit in the proof, to save on typography.

Procedure swop takes two formal value–result parameters y3 and y4, and has pre and postconditions:

ϕ_{swop} : true

ψ_{swop} : $y3{=}y4_c \wedge y4{=}y3_c$

The NRP is just the edge d, and the RP is the M–PATH comprising edges a, b and c. The CV is therefore l_{g0}, and the critical condition is $(y1{=}0 \vee y2{=}0)$. The JV is l_{g2}.

The transition axioms are:

F_a: $[at\ l_{g0} \wedge y1{\neq}0 \wedge y2{\neq}0 \wedge y1{<}y2 \wedge y{=}u] \rightarrow \Diamond[at\ l_{g1} \wedge y1{=}u2 \wedge y2{=}u1]$

F_b: [at l_{g0} $\wedge$ y1$\neq$0 $\wedge$ y2$\neq$0 $\wedge$ y1$\geq$y2 $\wedge$ y=u] -> $\diamond$[at l_{g1} $\wedge$ y=u]
F_c: [at l_{g1} $\wedge$ y=u] -> $\diamond$[at l_{g0} $\wedge$ y1=u1-u2 $\wedge$ y2=u2]
F_d: [at l_{g0} $\wedge$ (y1=0 $\vee$ y2=0) $\wedge$ y=u] -> $\diamond$[at l_{g2} $\wedge$ y=u]

The proof is in two parts:

(a) ϕ -> $\exists$k.Q(k,y)
(b) Q(k,y) -> $\diamond\psi$

ϕ -> $\diamond\psi$ by $\exists\diamond$Q of [Manna 82]

where Q(k,y) is

[at l_{g0} $\wedge$ 0$\leq$y1*y2$\leq$k $\wedge$ y1$\geq$0 $\wedge$ y2$\geq$0 $\wedge$ gcd($y1_c$,$y2_c$)=gcd(y1,y2)]

Note that we say 'at l_{g0}' as this is the CV.

The following shorthand is used in the proof:

α: gcd($y1_c$,$y2_c$)=gcd(y1,y2)

The 'g' in the vertex names, ϕ and ψ is omitted to simplify the typography.

Proof of (a):

1. ϕ_g -> [at l_0 $\wedge$ y1=$y1_c$ $\wedge$ y2=$y2_c$ $\wedge$ y1$\geq$0 $\wedge$ y2$\geq$0 $\wedge$ α]
by domain, PR. y1,y2 are value parameters.

2. [y1$\geq$0 $\wedge$ y2$\geq$0] -> [y1*y2 $\geq$0]
by PR

3. [y1*y2$\geq$0 -> 0$\leq$y1*y2$\leq$y1*y2] by domain

4. [at l_0 $\wedge$ y1=$y1_c$ $\wedge$ y2=$y2_c$ $\wedge$ y1$\geq$0 $\wedge$ y2$\geq$0 $\wedge$ α]
-> [at l_0 $\wedge$ 0$\leq$y1*y2$\leq$y1*y2 $\wedge$ y1$\geq$0 $\wedge$ y2$\geq$0 $\wedge$ α]
by PR

5. $[\text{at } l_0 \wedge 0 \leq y1*y2 \leq y1*y2 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \alpha]$
   $\rightarrow \exists k.[\text{at } l_0 \wedge 0 \leq y1*y2 \leq k \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \alpha]$
   by T24 of [Manna 82]

6. $[\text{at } l_0 \wedge 0 \leq y1*y2 \leq k \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \alpha] \rightarrow Q(k,y)$
   by def'n of Q

7. $\phi \rightarrow \exists k.Q(k,y)$ 1,4,5,6,PR

which concludes the proof of (a)

Once again, the proof of (b) is in two parts:

(b1) $Q(0,y) \rightarrow \diamond\psi$
(b2) $Q(m+1,y) \rightarrow \diamond\psi \vee \diamond Q(m,y)$

$Q(k,y) \rightarrow \diamond\psi$ by IND.

Proof of (b1)

8. $Q(0,y) \rightarrow [\text{at } l_0 \wedge y1*y2=0 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \alpha]$
   by PR.

9. $y1*y2=0 \rightarrow y1=0 \vee y2=0$ by PR

10. $y1=0 \vee y2=0 \rightarrow \max(y1,y2)=\gcd(y1,y2)$
    by def'n of GCD

11. $[\text{at } l_0 \wedge y1*y2=0 \wedge y1 \geq 0 \wedge y2 \geq 0 \wedge \alpha]$
    $\rightarrow [\text{at } l_0 \wedge (y1=0 \vee y2=0) \wedge \gcd(y1_c,y2_c)=\max(y1,y2)]$
    by PR

12. $[\text{at } l_0 \wedge (y1=0 \vee y2=0) \wedge y=u \wedge \gcd(y1_c,y2_c)=\max(y1,y2)]$
    $\rightarrow \diamond[\text{at } l_2 \wedge \gcd(y1_c,y2_c)=g_r \wedge y=u \wedge g_r=\max(y1,y2)]$
    by F_d, PR.

13. $[\text{at } l_2 \wedge \gcd(y1_c,y2_c) = \max(y1,y2) \wedge g_r=\max(y1,y2)]$
$\rightarrow [\text{at } l_2 \wedge g_r=\gcd(y1_c,y2_c)]$
by PR

14. $[\text{at } l_2 \wedge g_r=\gcd(y1_c,y2_c)] \rightarrow \psi$
by PR

15. $Q(0,y) \rightarrow \Diamond\psi$ by $\Diamond$Q.

which completes the proof of (b1)

Proof of (b2):

16. $Q(m+1,y) \rightarrow [\text{at } l_0 \wedge 0\leq y1*y2\leq m+1 \wedge y1\geq 0 \wedge y2\geq 0 \wedge \alpha]$
by PR

There are now two cases:

CASE 1: $y1*y2=0$

17. $[Q(m+1,y) \wedge y1*y2=0] \rightarrow [\text{at } l_0 \wedge y1*y2=0 \wedge y1\geq 0 \wedge y2\geq 0 \wedge \alpha]$
by PR

18. $[\text{at } l_0 \wedge y1*y2=0 \wedge y1\geq 0 \wedge y2\geq 0 \wedge \alpha] \rightarrow \Diamond\psi$
by 17,11..14,$\Diamond$Q,PR

CASE 2: $y1*y2>0$

19. $y1*y2>0 \rightarrow y1>0 \wedge y2>0$ by PR

20. $[\text{at } l_0 \wedge 0\leq y1*y2\leq m+1 \wedge y1*y2>0 \wedge y1\geq 0 \wedge y2\geq 0 \wedge \alpha]$
$\rightarrow [\text{at } l_0\ 0\leq y1*y2\leq m+1 \wedge y1>0 \wedge y2>0 \wedge \alpha]$
by PR

There are now two subcases:

SUBCASE 1: $y1<y2$

21. $[\text{at } l_0 \wedge 0\leq y1*y2\leq m+1 \wedge y1>0 \wedge y2>0 \wedge y1<y2 \wedge \alpha \wedge y=u]$
$\rightarrow \Diamond[\text{at } l_1 \wedge 0\leq y1*y2\leq m+1 \wedge y1=u2 \wedge y2=u1 \wedge y1\geq y2 \wedge y1\geq 0 \wedge y2\geq 0 \wedge \alpha]$
by F_a',
PCALL,
ψ_{swop}',
def'n of GCD,
PR

SUBCASE 2: $y1\geq y2$

22. $[\text{at } l_0 \wedge 0\leq y1*y2\leq m+1 \wedge y1>0 \wedge y2>0 \wedge y1\geq y2 \wedge y=u \wedge \alpha]$
$\rightarrow \Diamond[\text{at } l_1 \wedge 0\leq y1*y2\leq m+1 \wedge y1\geq 0 \wedge y2\geq 0 \wedge y1\geq y2 \wedge \alpha]$
by F_b, PR

End of subcases

23. $[y1\geq 0 \wedge y2\geq 0 \wedge y1\geq y2 \wedge 0\leq y1*y2\leq m+1] \rightarrow [0\leq (y1-y2)*y2\leq m]$
by PR, domain

24. $[\text{at } l_1 \wedge 0\leq y1*y2\leq m+1 \wedge y1\geq 0 \wedge y2\geq 0 \wedge y1\geq y2 \wedge y=u \wedge \alpha]$
$\rightarrow \Diamond[\text{at } l_0 \wedge 0\leq y1*y2\leq m \wedge y1\geq 0 \wedge y2\geq 0 \wedge y1=u1-u2 \wedge \alpha]$
by F_c',
def'n of call,
def'n of GCD,
PR

25. $[\text{at } l_0 \wedge 0\leq y1*y2\leq m \wedge y1\geq 0 \wedge y2\geq 0 \wedge \alpha] \rightarrow Q(m,y)$
by PR

26. $Q(m+1,y) \rightarrow \Diamond\psi \vee \Diamond Q(m,y)$ by 15,18,16,21..25,$\Diamond$Q.

The only possible return point inside the function is l_2, at which point we can deduce that g_r has been assigned the value of g, which we have shown is $GCD(y1_c,y2_c)$. This

completes the proof.

If this proof ends up looking just like the proof for loops, and this appears a trivial result, then the objective of this part of the thesis has been met – to make proving subprograms, especially recursive subprograms, as simple as it was to prove other programming constructs correct using a TPS before these extensions were introduced.

PART III - FORMAL SEMANTICS AND PROOFS OF AML

The first part of this thesis described a language capable of expressing the syntax and context conditions of a programming language. The second part extended the temporal proof system of Manna and Pnueli to cope with procedures and functions, with particular reference to recursive procedures and functions.

In this, the third part of the thesis, these two aspects are combined together, firstly to give a formal definition of the semantics of AML, and secondly to show that a CFF/AML specification can be proven 'correct' in some sense, usually defined by a set of assertions.

The formal definition of AML is given by constructing a TPS for the language. A TPS, as we have seen, consists of an uninterpreted logic part [Manna 82], a domain part, which specifies the domains of the programming language being defined, and a program part, in which the semantics of the programming language are defined.

The issue of WHY temporal logic is employed as the formalism for formal definition of the semantics of AML should also be addressed. This argument runs as follows:

- As discussed in the introduction to Part II, a verification methodology is required which will support the verification of procedures for which the traditional fixpoint approach would not be convenient. The enhanced temporal proof approach of Part II satisfies this requirement.
- Having built this system with the express purpose of using it to verify AML specifications, the next step is to instantiate a TPS for AML. This has to be done in order that AML verifications which can take advantage of the approach to recursion developed above can be performed.
- In building the TPS for AML, a natural side effect is that the semantics of the language become formally specified.
- Thus, temporal logic is the most natural way to define the semantics of AML. Any other additional specification would be redundant.

Chapter 14 gives the domain part of the TPS for AML, thus formally defining the domains in which the language can operate. Chapter 15 gives the program part of the TPS, thus formally defining the language itself. Chapters 16 through 18 give a number of examples of how the TPS can in fact be used to verify CFF/AML specifications.

The examples presented are kernel problems (such as verifying the context conditions on an expression definition) rather than verification of complete language specifications. We believe that this strikes a balance between producing verifications of complete language

specifications (which would be so large as to be of no purpose in a thesis) and producing verifications of trivial examples (which would not convince the reader of the advantages of verifying specifications).

It should be noted that this work is complete in the sense that all of AML is formally defined and that full language specifications can be verified. The application to full languages from the examples is a simple matter of scaling up the length of the proofs as more rules are introduced.

There are a number of benefits which accrue from a formal approach to the correctness of specifications. The first is exactly that advantage which accrues from the verification of programs, namely that one can get a guarantee that the specification of a programming language is correct in some sense. As compilers are critical pieces of software, and CFF/AML is meant to drive compilers, this ability to be sure of the correctness of a specification is vital.

Secondly, once a formal semantics and associated verification methodology exist for AML, it is then possible to develop specifications formally, following techniques such as those espoused in [Gries 81]. Again, the advantage that accrues is that specifications can be developed more accurately and quickly.

Thirdly a benefit specific to context condition specification is accrued. An attribute grammar specification can be shown to be noncircular, which implies that the attribute evaluations must eventually terminate. Further, if certain ordering relations between attributes can be shown to hold then the time for termination can be shown to be bounded. However, nothing can be said about the correctness of the evaluations once they have terminated. When verifying AML, it can be shown that evaluations terminate and also that they terminate with a correct result.

Chapter 14: THE DOMAIN PART

The domain part consists of a description of the domains on which the language for which the TPS is being constructed operates.

AML simple variables operate in the integer, boolean and string domains. The usual operators are defined on boolean and integer variables. String variables have concatenation, assignment, equality and inequality operations as well as a matching operation. A matches B is true if A is a substring of (or the same string as) B.

For AML name tables the situation is somewhat more complex. AML name tables may be of set, list or stack type. A set type can be treated as an ordinary set and is represented in curly brackets {}. A list type is represented in <>, and a stack is represented in ().

To model sets, lists, stacks and scoping, a number of abstract data type operations on sets, lists and stacks are introduced. These operations are almost all polymorphic. The reader is therefore urged to pay close attention to these definitions so as to avoid confusion when they are used later in the thesis.

The usual operators on a set are defined on a set type, namely union(U) and elementhood($\in$). Both of these are binary operators. The union of x and y is represented by

$$x \; U \; y$$

The disjunction of x and y is represented as

$$\tilde{}(x \; U \; y)$$

Two primitive operations, ins and del are defined which insert elements into and delete elements from a set:

$$ins(x,X) = x \; U \; X$$
$$del(x,X) = \tilde{}(x \; U \; X)$$

where x is an element and X a set.

For all the list and stack work, the structure is assumed to have been normalised in that if we have, for example, a list nested in a list:

$$<a,b,...,c>$$

where b is a list

$$<d,e,...,f>$$

Then the normalised representation is:

$$<a,d,e,...,f,...,c>$$

A stack can be normalised in a similar fashion.

Two primitive operators on lists, head and tail, are defined. Note the polymorphic use of "nil":

$$\begin{aligned} \text{head}(<a,b,...,c>) &= a \\ \text{tail}(<a,b,...,c>) &= <b,...,c> \\ \text{head}(<a>) &= a \\ \text{tail}(<a>) &= \text{nil} \\ \text{head}(<>) &= \text{nil} \\ \text{tail}(<>) &= \text{nil} \end{aligned}$$

Elementhood and union are defined using a functional notation for the infix operators:

$$\begin{aligned} \in(x,X) = \text{if } x = \text{head}(X) &\text{ then true} \\ \text{else if } \text{tail}(X) = \text{nil} &\text{ then false} \\ \text{else } \in(x,&\text{tail}(X)) \end{aligned}$$

$$U(X,Y) = <X,Y>$$

The definition of ins on a list is simply:

$$\text{ins}(x,X) = <X,x>$$

The definition of del on a list is somewhat more complex:

$$\begin{aligned} \text{del}(x,X) = \text{temp} &:= <> \\ &\text{delbody}(x,X) \\ &X := \text{temp} \end{aligned}$$

$$\begin{aligned} \text{delbody}(x,X) = \text{if } \text{head}(X) &= x \\ &\text{then temp} := <\text{temp},\text{body}(X)> \\ &\text{else begin} \\ &\quad \text{ins}(\text{head}(X),\text{temp}) \\ &\quad \text{if } \text{tail}(X) \neq \text{nil} \text{ then delbody}(x,\text{tail}(X)) \\ &\text{end} \end{aligned}$$

where x is an element of the list X and temp is an auxiliary list.

For each list s there is an auxiliary list s' which is used in the definition of FRONT, NEXT and EOL.

A stack is analogously defined, with the primitive operators top and body:

$$\begin{aligned} \text{top}((a,b,...,c)) &= a \\ \text{body}((a,b,...,c)) &= (b,...,c) \\ \text{top}((a)) &= a \end{aligned}$$

body((a)) = nil
top(()) = nil
body(()) = nil

Note the slight ambiguity of the brackets; the outer are for the function and the inner represent the stack. The operations of elementhood and union can be defined for stacks, once again using functional notation for the infix operators:

$\in$(x,X) = if top(X) = x then true
else if top(X) = nil then false
else $\in$(x,body(X))

U(X,Y) = (X,Y)

where X and Y are stacks and x is (possibly) an element of a stack.

Again ins and del are defined for a stack, using an approach analogous to that used for lists:

```
ins(x,X) = (x,X)

del(x,X) = temp := ()
           delbody(x,X)
           X := ()
           invert(temp,X)

delbody(x,X) = if top(X)=x
                 then begin
                   invert(body(X),W)
                   temp := (W,temp)
                 end
               else begin
                 ins(top(X),temp)
                 if body(X) ≠ nil then delbody(x,body(X))
               end

invert(Y,Z) = if top(Y) ≠ nil then
                begin
                  ins(top(Y),Z)
                  invert(body(Y),Z)
                end
```

The introduction of the function invert is messy but necessary as delbody will create an inverted stack.

There is also a 'stack of name tables' notation needed, which is represented in square brackets []. This notation will be used when the scoping issue is dealt with below.

Top and body on the stack of name tables structure are defined as:

$$\begin{aligned} \text{top}([a,b,...,c]) &= a \\ \text{body}([a,b,...,c]) &= [b,...,c] \\ \text{top}([a]) &= a \\ \text{body}([a]) &= \text{nil} \\ \text{top}([]) &= \text{nil} \\ \text{body}([]) &= \text{nil} \end{aligned}$$

Elementhood, on this 'stack of name tables' structure is:

$$\begin{aligned} \in(x,s) = \text{if } x \in \text{top}(s) \\ \text{then true} \\ \text{else if body}(s) = \text{nil then false} \\ \text{else } \in(x,\text{body}(s)) \end{aligned}$$

The set of all name tables for a particular specification is denoted by E. The set of name tables local to a scope is denoted L. Each element of L is attributed with a type, which can be set, list or stack. The type attribute is extracted using the function TYP.

For an arbitrary name table s, s denotes the elements of the name table at the local scope, TYP(s) denotes the type of the table at the local scope and s* denotes the elements visible at other scope levels. s* is a stack of name tables, $[s_1,...,s_n]$ where s_1 is the immediately enclosing scope and s_n is the most global scope. Note that the type of a table is a local attribute and may be different in visible but non-local scopes. When it would be unambiguous to do so, s is also used to denote the NAME of the table.

We usually work with two scopes. These are the scope of the tree walk pointers and the scope of the AML pointers. We will differentiate when necessary by subscripting with a T for tree walk pointers and an A for AML pointers, ie L_T means the set of name tables local in the scope of the tree walk pointers. There exists a switch which indicates which pointer is to be used and which takes on values from (treep, amlp) (see chapter 4). When there is no subscript this is interpreted to mean the scope defined by the switch. The only exception is when name table names are bound; then the scope is defined by the binding. For example, in

$$\forall s \in L_T : s \neq \text{nil}$$

s is always in the tree pointer scope as defined by the binding to L_T.

Each entry in a stack of name tables has associated with it an indicator, like L_T or L_A, except that it indicates which tables are declared in each level of the stack (because an entry in a stack of name tables corresponds to a scope level). This indicator is denoted L_S. It takes as argument a stack of tables and returns the local name table set for the top

element of the stack.

The scope of the pointers may be referred to as scope(amlp) or scope(treep) when necessary.

In order that the scoping rules to be defined below can have the proper effect, it is necessary to consider what happens when the scope of the AML pointers changes. This can happen either as the result of a name table lookup or as the result of executing a SON into a scope or executing a FATHER out of a scope.

When the scope changes as the result of a successful name table lookup, the scope of the AML pointers must be the same as the scope of the identifier found in the lookup. Each element of every name table therefore has associated with it a logical attribute SCOPE. A change to any element in a scope changes the SCOPE attribute of every other element in the scope. This logical change is indicated by:

$S := S + x$ where S is a logical scope and x the inserted element

$S := S - x$ where S is a logical scope and x the deleted element

Insertion and deletion into name tables must reflect this logical change. Two update functions, inselt and delelt, are provided which use the auxiliary functions insentry and delentry. All insertions and deletions into the name tables must use inselt and delelt.

```
insentry(x,S): if top(S) ∈ L_S(S) then ins(x,top(S)) else
                   if body(S) = nil then error
                                    else insentry(x,body(S))

delentry(x,S): if top(S) ≠ nil and x∈top(S)
                   then del(x,top(S))
                   else if body(S) = nil then error
                          else delentry(x,body(S))

inselt(x,X,L): insentry(x,[X,X*],[L,l*])
                   ∀s∈L:∀p∈s: scope(p) := scope(p) + x

delelt(x,X,L): delentry(x,[X,X*])
                   ∀s∈L:∀p∈s: scope(p) := scope(p) - x
```

where S is a stack of name tables. Note the use of L_S on the first line of insentry to check if there is a local occurrence of the name table on the local scope. If not, then the entry is inserted into the first local occurrence of the table.

When the scope changes as the result of a SON or FATHER the scope of the AML pointers must also be changed accordingly. Every rule where scope changes therefore has

associated with it two logical attributes, NEW-SCOPE and OLD-SCOPE. A SON off such a rule into a scope moves the scope of the AML pointers into the NEW-SCOPE of the rule; a FATHER moves the scope back into the OLD-SCOPE of the rule.

The scope changes associated with FATHER and SON instructions are taken care of in the construction of the program part of the TPS. A set SC of rules where scope changes must be identified by the language specifier for use in the SON and FATHER commands. A set called RHS is also needed for each rule. This has as elements the nonterminals in the rule. This set is used in the father and son operations. The goal rule of the CFF grammar is distinguished and called the root rule. A quick forward reference to a slightly simplified version of the definition of the SON instruction below will illustrate why these sets are needed. (Some of the terms used in this forward reference may not have been explained as yet; they will be defined below).

SON(u2):
[amlp=u1 $\wedge$ u1$\in$SC $\wedge$ u2$\in$RHS(u1)] -> O[amlp=u2 $\wedge$ scope(amlp)=NEW-SCOPE(u1)]
[amlp=u1 $\wedge$ $\tilde{}$(u1$\in$SC) $\wedge$ u2$\in$RHS(u1)] -> O[amlp=u2 $\wedge$ scope(amlp)=scope(u1)]
[amlp=u1 $\wedge$ $\tilde{}$(u2$\in$RHS(u1))] -> error

This definition tells us that:

(1) If the rule pointed to by the aml pointers is called u1, and u1 is a rule where a scope change occurs (by definition of SC), and u2 is indeed a son of u1 (because it is in RHS(u1)), then in the next state the aml pointer points to u2 and the scope of the aml pointer has altered.

(2) If the rule pointed to by the aml pointers is called u1, and u1 is not a rule where a scope change occurs (by definition of SC), and u2 is indeed a son of u1 (because it is in RHS(u1)), then in the next state the aml pointer points to u2 and the scope of the aml pointer is unchanged.

(3) If the rule pointed to by the aml pointers is called u1 and u2 is not a son of u1 then an error results.

A name table entry consists of a string (usually an identifier) and a pointer (to a node on the tree - at this abstract level this can be considered as a pointer to a rule). The language specifier must construct, for each name table in the specification, a set IP of insertion points for that name table. The elements of IP are the rules which will be pointed to as a result of an insert into the table. Then, when a name table lookup succeeds we know that the AML pointers must be pointing to one of the rules in the IP

for the set on which the lookup was made. (In the event that a lookup across multiple tables succeeds the IP must be treated as the set created from the union of the IP's for each name table).

For example, if a name table 'vars' is inserted into on the rules vars, fields and parameters (as is the case with Pascal), then the IP of vars is:

$$IP(vars) = \{vars, fields, parameters\}$$

If a lookup on table vars succeeds, then we know that after the lookup,

$$amlp = \text{'vars'} \vee amlp = \text{'fields'} \vee amlp = \text{'parameters'}$$

where amlp is the aml pointer. If a table called types has IP defined as

$$IP(types) = \{types\}$$

and a lookup across vars and types is performed then if the lookup succeeds we have the predicate:

$$amlp=\text{'vars'} \vee amlp=\text{'fields'} \vee amlp=\text{'parameters'} \vee amlp=\text{'types'}$$

which is obtained from IP(vars) U IP(types).

AML also operates on an number of stacks, namely the AML stack, denoted AML (operated on by the PUSH and POP instructions), the mark stack, denoted MARK (operated on by the MARK and RELEASE instructions) and the AML positional stack, denoted P (operated on by the SAVEPOSITION and RESTOREPOSITION instructions). Each simple variable in AML may have its value saved and restored. There is therefore a stack V–id for each simple AML variable id in an AML specification (operated on by the SAVE and RESTORE instructions).

The operations TOP and SIZE (indicated by placing the name of the stack between $||$) are defined on each stack.

Note that when the AML pointer is saved on P the associated scope and the state of each list name table in that scope are saved as well, and both the pointer, scope and list states are restored together.

The type compatibility tables and commands, and their exact effect, as well as the fact that a type compatibility can change along with scope changes if the language being specified allows user–definable operator overloading, must also be considered.

A type compatibility table can be thought of as being a matrix, with size denoted by the table name between $||$, eg the size of the dyadic table is $|dcomp|$, and two columns. The

first column is a composite of the types of the operands and the operator, and the second is the result type (a null column in the case of the assignment table). Operations to insert elements into a table, and a way of handling table scoping analogous to name table scoping as handled above, must be defined. A generic insert operation for the tables can be defined as:

```
ins-table(tabname,x,y) : |tabname|:=|tabname|+1 ;
                         tabname(|tabname|) := (x,y)
```

The lookup operation on the type compatibility tables is defined as:

```
look(tabname,x) : if ∃i:tabname(i,1)=x
                    then tabname(i,2)
                    else undef.
```

where undef is a special unique value, and the equality is understood after any wildcard substitutions have been made.

Now, suppose the concept of multiple scopes is introduced. We can allow a 'stack of type tables' concept similar to the stack of name tables concept that we introduced for handling scopes above, use the same top and body constructs. The local table is called by the name of the operation, and the global tables by the name suffixed with a *, eg the dcomp table is called dcomp, and the nonlocal entries are in dcomp*.

The lookup operation with scopes is:

```
lookup(tabname,x) : if look(tabname,x) ≠ undef
                       then look(tabname,x)
                       else
                        if body(tabname)=nil
                          then std-error
                          else lookup(body(tabname),x)
```

The elements of the AML stack, which are used for type checking, will also be referred to. The top element of the stack will be referred to as S(0), and the second and third elements as S(1) and S(2) respectively.

Chapter 15: THE PROGRAM PART

This chapter considers the second part of a TPS that must be supplied by the TPS constructor, namely the program part. This part consists of a set of templates which describe exactly the effect of each statement in the language being specified.

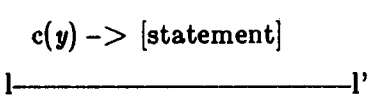
The CFF part of the language is not considered overmuch here. CFF lists will translate to loops in the graphical representation of CFF/AML rules to be used when proving the correctness of AML decorations. Only syntactically correct programs are considered to have AML applied to them, so these loops (the CFF lists) can be presumed to terminate.

A CFF list is similar to a REPEAT construct in a programming language and we may associate with the exit point of the list an 'intermittent assertion' which must hold each time the exit point of the list is reached. When a CFF list exit point is reached, there will therefore be a NONDETERMINISTIC choice at verification time as to whether or not the list is repeated. However, because of the assumption that syntactic loops terminate, this nondeterminism can be ignored.

In the body of any rule the name of another nonterminal may be found. These nonterminals may be considered to be to be SYNTACTIC PROCEDURES. Any syntactic procedure may have pre and post–conditions associated with it.

The formal definition of AML statements and functions is performed by category, ie conventional, scope, tree and stack statements.

The conventional if, block, assignment, repeat, while, null and procedure call statements have the conventional meanings and are not considered any further here. [Manna 82, Manna & Pnueli 83b] discuss conventional statements and procedure and function calls were described in part II above. The remaining statements are all defined as if they are labels on an arbitrary transition. Once the semantics of an AML statement has been formally defined, it may decorate an edge of a graph, for example SON(x) means the definition of SON given below. The arbitrary edge could be:



In these definitions the notation of [Manna 82] and part II is followed, so that any variable beginning with y is a local variable and any variable beginning u is a global variable. Other characters may be used from time to time as variable names; these are explicitly mentioned where they are used.

set(y_i):

$[\text{at } l \wedge y{=}u \wedge c(y)] \rightarrow O[\text{at } l' \wedge y_i{=}\text{true} \wedge \forall j{:}j{\neq}i{:}y_j{=}u_j]$

reset(y_i):

$[\text{at } l \wedge y{=}u \wedge c(y)] \rightarrow O[\text{at } l' \wedge y_i{=}\text{false} \wedge \forall j{:}j{\neq}i{:}y_j{=}u_j]$

save(y_i):

$[\text{at } l \wedge c(y) \wedge y{=}u \wedge |V{-}y_i|{=}S] \rightarrow O[\text{at } l' \wedge y{=}u \wedge \text{top}(V{-}y_i){=}u_i \wedge |V{-}y_i|{=}S{+}1]$

where S is an auxiliary global variable

restore(y_i):

$[\text{at } l \wedge c(y) \wedge y{=}u \wedge \text{top}(V{-}y_i){=}v \wedge |V{-}y_i|{=}S \wedge S{>}0]$

$\rightarrow O[\text{at } l' \wedge |V{-}y_i|{=}S{-}1 \wedge y_i{=}v \wedge \forall j{:}j{\neq}i{:}y_j{=}u_j]$

where S and v are auxiliary global variables

error(S), warning(S):

$[\text{at } L \wedge c(y) \wedge y{=}u] \rightarrow O[\text{at } l' \wedge y{=}u]$

where S is a string literal

The semantics of the scope statements can be given in the same way. The definitions will be interspersed with discussion designed to illuminate them.

The scope statements do not affect the AML simple variables. The definitions to be given below are simplified by assuming that we start at l with $c(y)$ and $y{=}u$', and that in the next state we are at l' with $y{=}u$'. Where the definition has multiple lines each of which is numbered, this means that the statement being defined has a number of operations which are performed in the order given. When the lines are not numbered then it means that these are a number of alternative definitions, one of which will be used.

newscope:

(1) OLD-SCOPE(treep) := scope(treep)

(2) $\forall s{:}s{\in}L_T{:}[s^*{=}u^* \wedge s{=}u] \rightarrow O[s^*{=}[u, u^*] \wedge L_S(\text{top}(S^*)){=}L_T]$

(3) $\forall s{:}\tilde{}(s{\in}L_T){:}[s^*{=}u^*] \rightarrow O[s^*{=}[\text{nil}, u^*] \wedge L_S(\text{top}(S^*)){=}L_T]$

(4) $\forall s{:}s{=}(\text{mcomp,dcomp,acomp}){:}[s{=}u \wedge s_*{=}u^*] \rightarrow O[s^*{=}[u,u^* \wedge s{=}\text{nil}]$

(5) $O[L_T = \{\}]$

Scope should be thought of as a stacklike object. Creating a newscope pushes all the current local name tables down a level in the stack, and exiting from the scope pops them all back up again. In order that tables which were not local do not get popped back on to

the wrong level, a nil value is pushed for them instead so that the endscope works properly.

The newscope definition therefore says that the oldscope is recorded, and all the local name tables are saved. Then, all the nonlocal tables have a nil value pushed into the stack of name tables, the compatibility tables are also pushed one level, and the new local scope becomes empty.

endscope:
(1) NEW-SCOPE(treep) := scope(treep)
(2) temp := {}
(3) $\forall$s:(s$\in$E $\wedge$ s*=[u1,u2] $\wedge$ u1 $\neq$ nil): O(temp=tempUs $\wedge$ s=u1 $\wedge$ s*=[u2])
(4) $\forall$s:(s$\in$E $\wedge$ s*=[u1,u2] $\wedge$ u1=nil):O(s*=[u2])
(5) $\forall$s:(s=mcomp,dcomp,acomp):s*=[u1,u2] -> O(s=u1 $\wedge$ s*=u2)
(6) L_T := temp
where temp is a global set

Endscope has the opposite effect to newscope; the newscope is saved and the name tables and compatibility tables are all popped back to their previous state.

settype(s):
L_T := L_T U s $\wedge$ TYP(s) = set

listtype(s):
L_T := L_T U s $\wedge$ TYP(s) = list

stacktype(s):
L_T := L_T U s $\wedge$ TYP(s) = stack

dump(s):
no noticeable effect from the verification viewpoint

clear(s):
s$\in$L $\wedge$ s=u1 $\wedge$ s*=[u2] -> O[s=nil $\wedge$ s*=[u2]]
~(s$\in$L) -> O(s* = [])

insert(w,s):
inselt(w,s,L)

where w is a string and s a name table

In actual fact w is not a string but a combination of string and pointer p, where p can be defined by:

p= if |MARK|=0 then
    if switch = amlp then amlp else treep
  else top(MARK)

The pointer is not a factor when doing lookup, other than to set the scope of the AML pointer and the place it points to on the tree.

delete(w,s):
delelt(w,s,L)
where w is a string and s a name table

copy(s):
$[s \in L_T \wedge s \in L_A \wedge s_T = u_T \wedge s_A = u_A] \rightarrow O[s_T = u_T U u_A]$

unstack(S):
$[top(S) \neq nil \wedge top(S) = (u_1, u_2, \ldots, u_n) \wedge TYP(top(S)) = stack] \rightarrow O[top(S) = (u_2, \ldots, u_n)]$
$[top(S) = nil \wedge body(S) \neq nil] \rightarrow [\ unstack(body(S))\ ]$
$[top(S) = nil \wedge body(S) = nil] \rightarrow$ error
where S is a stack of name tables $[s_1, s_2, \ldots, s_n]$

front(s):
$[TYP(s) = list] \rightarrow O[s' = s \wedge amlp \in IP(s) \wedge switch = aml]$

next(s):
$[tail(s') \neq nil \wedge tail(s') = u1 \wedge TYP(s) = list] \rightarrow O[s' = u1 \wedge amlp \in IP(s) \wedge switch = aml]$

treescope:
O[switch = treep]

amlscope:
O[switch = amlp]

load-acomp-table(s1,s2):
ins-table(acomp,s1s2,nil)
where s are global auxiliary string expressions

load-mcomp-table(s1,s2,s3):
ins-table(mcomp,s1s2,s3)
where s are global auxiliary string expressions

load-dcomp-table(s1,s2,s3,s4):
ins-table(dcomp,s1s2s3,s4)
where s are global auxiliary string expressions

The tree statements can be similarly defined. Again, these statements cannot affect the AML simple variables or the fact that we must arrive at l' in the next state. For all the tree statements, S represents a global auxiliary variable. Savedlist is an auxiliary attribute of top(P) used in saveposition and restoreposition to represent the list statuses saved during the saveposition operation and restored by a restoreposition.

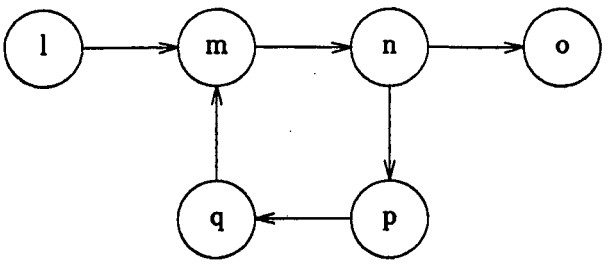
saveposition:
[¦P¦=S ∧ amlp=u1]
-> O[¦P¦=S+1 ∧ top(P)=u1 ∧ scope(top(P))=scope(u1)
∧ ∀s:(s∈L_A ∧ TYP(s)=list):s'=savedlist(s,top(P))]

restoreposition:
[¦P¦=S ∧ top(P)=u1 ∧ S>0]
-> O[amlp=u1 ∧ scope(amlp)=scope(u1) ∧ ¦P¦=S-1
∧ ∀s:(s∈L_A ∧ TYP(s)=list):s'=savedlist(s,u1)]

mark:
[¦MARK¦=S ∧ amlp=u1] -> O[¦MARK¦=S+1 ∧ top(MARK)=u1]

release:
[¦MARK¦=S ∧ S>0] -> O[¦MARK¦ = S-1]

intolist, next-iteration:
if we have a structure of a CFF list represented



then

intolist: [at l] -> O[at m]

next–iteration: [at m ∨ at n ∨ at p ∨ at q] -> O[at m]

The remaining two AML tree statements are SON and FATHER which are both complicated by the fact that there may be a scope change involved if the rule is in the set SC.

son(x):

(1) [if switch=amlp then ptr=amlp else ptr=treep]

(2) [ptr∈SC ∧ x∈RHS(ptr) ∧ ptr=u1]

-> O[ptr=x ∧ scope(ptr)=NEW-SCOPE(u1) ∧ switch=amlp]

[~(ptr∈SC) ∧ x∈RHS(ptr) ∧ ptr=u1]

-> O[ptr=x ∧ scope(ptr)=scope(u1) ∧ switch=amlp]

[~(x∈RHS(ptr))] -> error

Intuitively, the FATHER command has the effect of moving the AML pointers to an ancestor of the current node of the tree. In practice, FATHER must also take into account scope changes, and the fact that the argument rule of the command may not in fact be an ancestor of the current node. The trivial part of the command has the effect "move one level up the tree", ie to the immediate ancestor of the current node. We assume that there is a primitive function "uplevel" which performs this operation by changing the AML pointers, and then define FATHER as:

father(x):

(1) [if switch=amlp then ptr=amlp else ptr=treep]

(2) uplevel ;

switch := amlp ;

if ptr∈SC then scope(amlp) := OLD-SCOPE(ptr) ;

if amlp = root and amlp ≠ x then error ;

if amlp ≠ x then father(x)

The pointer 'ptr' is first set to point to the current node. Then an uplevel is performed,

which moves us to the immediate ancestor of the current node. If we execute a FATHER off a node in the set SC of scope changes, then the current scope must change along with the current node. If we reach the root of the tree and we are not on the argument node then an error results. If the goal of the search has not been reached, then the operation is repeated, thus stepping up the tree until the root or the goal node is reached. The stepping is needed to correctly perform scope changes.

The stack statements are extremely simple to define. The location and simple variable information are included here here as they can be altered as a result of a stack statement:

push(y_i):
[at l $\wedge$ c(y) $\wedge$ y=u $\wedge$ |AML|=S] -> O[at l' $\wedge$ y=u $\wedge$ |AML|=S+1 $\wedge$ top(AML)=y_i]

pop:
[at l $\wedge$ c(y) $\wedge$ y=u $\wedge$ |AML|=S $\wedge$ S>0] -> O[at l' $\wedge$ y=u $\wedge$ |AML|=S-1]

pop(y_i):
[at l $\wedge$ c(y) $\wedge$ y=u $\wedge$ |AML|=S $\wedge$ top(AML)=u' $\wedge$ S>0]
-> O[at l' $\wedge$ |AML|=S-1 $\wedge$ y_i=u' $\wedge$ $\forall$j:i$\neq$j:y_j=u_j]

acomp:
[at l $\wedge$ c(y) $\wedge$ y=u $\wedge$ |AML|=S $\wedge$ S>1 $\wedge$ AML=V $\wedge$ acomp=v1 $\wedge$ acomp*=v2]
-> O[at l' $\wedge$ y=u $\wedge$ |AML|=S-2 $\wedge$ top(AML)=lookup([v1,v2], V(0)V(1))]
where v are auxiliary global variables

mcomp:
[at l $\wedge$ c(y) $\wedge$ y=u $\wedge$ |AML|=S $\wedge$ S>1 $\wedge$ AML=V $\wedge$ mcomp=v1 $\wedge$ mcomp*=v2]
-> O[at l' $\wedge$ y=u $\wedge$ |AML|=S-1 $\wedge$ top(AML)=lookup([v1,v2],V(0)V(1))]
where v are auxiliary global variables

dcomp:
[at l $\wedge$ c(y) $\wedge$ y=u $\wedge$ |AML|=S $\wedge$ S>2 $\wedge$ AML=V $\wedge$ dcomp=v1 g$\wedge$ dcomp*=v2]
-> O[at l' $\wedge$ y=u $\wedge$ |AML|=S-2 $\wedge$ top(AML)=lookup([v1,v2], V(0)V(1)V(2)]
where v are auxiliary global variables

The AML functions that do not reference stacks and name tables can be adequately defined at the abstract level. Here tree is the tree from which the information would be

extracted, switch indicates whether to use the amlp or treep, and string is the string type described at the start of the domain section above,

contents: tree X switch -> string
value: tree X switch -> integer
length: tree X switch -> integer
rule: tree X switch -> string
subtree: tree X switch -> boolean
another-iteration: tree -> boolean

The last function is unusual in that it returns information concerning the syntactic structure of a particular program. For this reason it is assumed to eventually become false (if used in conjunction with next-iteration) on the grounds that only syntactically correct programs, and correct programs must have finite CFF lists.

The one stack-related function, stacktop, can be defined as:

stacktop:
if $|AML| > 0$ then top(AML) else error

The operations on name tables via functions are defined as:

visible(x,X):
if $x \in ([X,X^*])$ then true $\wedge$ amlp$\in$IP(X) $\wedge$ switch=amlp
$\wedge$ scope(amlp) = scope(x)
else false

Scope(x) refers to the LOGICAL SCOPE attribute of the x which is an element of $[X,X^*]$.

local(x,X):
if $x \in X$ then true $\wedge$ amlp$\in$IP(X) $\wedge$ switch=amlp
else false

unique(x,X):
if $X \in L$ then
if $\tilde{}(x \in X)$ then true else false
else if top(X^*)$\in L_S$(top(X^*))
then unique(x,top(X^*)
else if body(X^*)=nil then false
else unique(x,body(X^*))

Unique is a particularly interesting function. Its definitio
is local then unique looks in the local scope only. If how
then search the global scopes until the most local instan
search that instantiation. Unique thus uses the MOST L
name for the check.

top(X):
if top(X) $\neq$ nil
 then
  if TYP(top(X))=stack
   then top(top(X)) $\wedge$ amlp$\in$IP(X) $\wedge$ switch=amlp
   else error
 else
  if body(X) $\neq$ nil then top(body(X))
where X is a stack of name tables type of form [X,x*]

empty(X):
if X = nil then true else false

eol(x):
if TYP(x) $\neq$ list
 then error
 else if x' = nil $\wedge$ TYP(x)=list
  then true
  else false

which completes the TPS for AML and the formal specifi

We turn now to applying the TPS to proving CFF/AM
cated in the introduction to this part of the thesis,
language specifications proofs of selected parts of speci
that the reader will be convinced that these can be scalec

Chapter 16: VERIFYING EXPRESSIONS

Consider the rule "expression" decorated in the first example of part I. The CFF/AML for the rule is:

```
"expression
?sign<<factor[ if monad then begin mcomp ; reset(monad) end ;
                 if dyad  then begin dcomp ; reset(dyad)  end
              ]_multop[ set(dyad)
                      ]>_addop[ set(dyad)
                              ]> %

"factor"
 identifier[ if visible(vars) then
               push(type)
             else begin
                    push(std-error) ;
                    error('Invisible Identifier')
                  end ]
|integer [ push('integer') ]
|TRUE    [ push('boolean') ]
|FALSE   [ push('boolean') ]
|[ save(dyad) ;
   save(monad) ;
 ](expression) [ restore(dyad) ;
                 restore(monad) ] %

"addop"
 + [ push('+') ]
|- [ push('-') ] %

"multop"
 * [ push('*') ]
|/ [ push('/') ] %

"sign"
 + [ set(monad) ; push('+') ]
|- [ set(monad) ; push('-') ] %
```

where the predicate and associated relative operators are ignored without prejudice to the utility of the proof.

We must now consider what checks are required on the AML code of the expression.

From the external viewpoint, the rule must satisfy the requirement that it leaves only one piece of information on the stack, namely the type of the expression. The postcondition of the 'syntactic procedure' which is the CFF expression rule is therefore that the size of

the AML stack is one more than the size when the expression was entered.

Within the rule, we must check that DCOMP and MCOMP are only executed when there are the correct number of elements on the stack and that the correct type is put on the stack as the result of the MCOMP or DCOMP. We associate with the rule factor the postconditions that it increases the stack size by 1, leaves the values of dyad and monad unaltered and returns on the top of the stack the correct type of the factor.

We associate similar conditions with the rules multop and addop, namely that they increase the stack size by 1 and leave an operator of the correct type on the top of the stack. The rule sign operates similarly except that is must also set the flag monad.

With the above in mind we look at the proof. By convention, each syntactic object is represented in the line of the vertices of the graph, and AML code labels the vertices. With each syntactic object is associated a precondition and postcondition, and, having shown that the precondition holds, we assume that the postcondition holds (effectively an application of the procedure call rule PCALL developed in part II). The graph for the expression rule is given in figure 1.

The preconditions and postconditions for the various rules are:

expression:

ϕ: $dyad_0$=false $\wedge$ $monad_0$=false $\wedge$ $|AML|_0 \geq 0$

ψ: $|AML| = |AML|_0 + 1$

factor:

ϕ: true

ψ: $|AML| = |AML|_0 + 1 \wedge dyad = dyad_0 \wedge monad = monad_0$

sign:

ϕ: true

ψ: $|AML| = |AML|_0 + 1 \wedge monad = true \wedge (top(AML) = '-' \vee top(AML) = '+')$

addop:

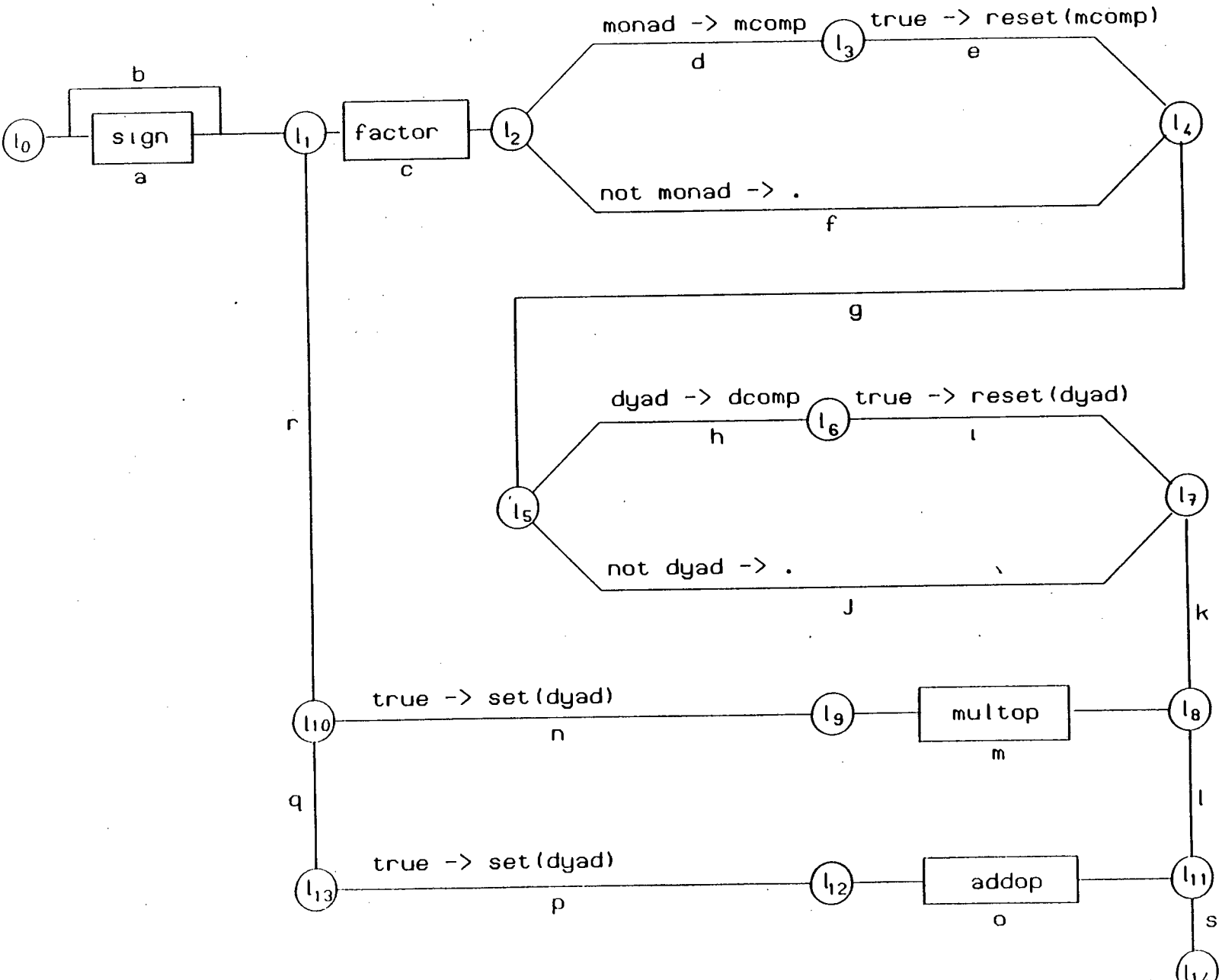
ϕ: true

ψ: $|AML| = |AML|_0 + 1 \wedge (top(AML) = '-' \vee top(AML) = '+')$

multop:

ϕ: true

figure 1 - expression



ψ: $|AML| = |AML|_0 + 1 \wedge (top(AML) = '*' \vee top(AML) = '/')$

By convention, if a variable is not mentioned in a transition axiom it is assumed to be unaltered by that transition. Further, all variable names from the AML code are used in the proof and considered local; global variables still start with a 'u'. The transition axioms for expression (using the weaker version[Manna 82]) are:

F_a: $[at\ l_0 \wedge |AML|=u] \rightarrow \Diamond[\ at\ l_1 \wedge |AML| = u+1 \wedge monad=true\ ]$

F_b: $[at\ l_0] \rightarrow \Diamond[at\ l_1]$

F_c: $[at\ l_1 \wedge dyad=u \wedge monad=u' \wedge |AML|=u'']$
$\rightarrow \Diamond[at\ l_2 \wedge dyad=u \wedge monad=u' \wedge |AML|=u''+1]$

F_d: $[at\ l_2 \wedge monad \wedge |AML|=u] \rightarrow \Diamond[at\ l_3 \wedge |AML|=u-1]$

F_e: $[at\ l_3] \rightarrow \Diamond[at\ l_4 \wedge monad=false]$

F_f: $[at\ l_2 \wedge \tilde{}monad] \rightarrow \Diamond[at\ l_4]$

F_g: $[at\ l_4] \rightarrow \Diamond[at\ l_5]$

F_h: $[at\ l_5 \wedge dyad \wedge |AML|=u] \rightarrow \Diamond[at\ l_6 \wedge |AML|=u-1]$

F_i: $[at\ l_6] \rightarrow \Diamond[at\ l_7 \wedge dyad=false]$

F_j: $[at\ l_5 \wedge \tilde{}dyad] \rightarrow \Diamond[at\ l_7]$

F_k: $[at\ l_7] \rightarrow \Diamond[at\ l_8]$

F_l: $[at\ l_8] \rightarrow \Diamond[at\ l_{11}]$

F_m: $[at\ l_8 \wedge |AML|=u] \rightarrow \Diamond[at\ l_9 \wedge |AML|=u+1]$

F_n: $[at\ l_9] \rightarrow \Diamond[at\ l_{10} \wedge dyad=true]$

F_o: $[at\ l_{11} \wedge |AML|=u] \rightarrow \Diamond[at\ l_{12} \wedge |AML|=u+1]$

F_p: $[at\ l_{12}] \rightarrow \Diamond[at\ l_{13} \wedge dyad=true]$

F_q: $[at\ l_{13}] \rightarrow \Diamond[at\ l_{10}]$

F_r: $[at\ l_{10}] \rightarrow \Diamond[at\ l_1]$

F_s: $[at\ l_{11}] \rightarrow \Diamond[at\ l_{14}]$

In addition to the transition axioms we postulate two intermittent assertions concerning the program, one for each CFF list. In general it is always possible to make such assertions concerning a list, which reflect the values of the AML variables, stacks etc each time the exit point of the list is reached.

I_1: $[at\ l_8] \rightarrow [dyad=false \wedge monad=false \wedge |AML|=|AML|_0+1]$

I_2: $[at\ l_{11}] \rightarrow [dyad=false \wedge monad=false \wedge |AML|=|AML|_0+1]$

Recall the remarks concerning nondeterminism of CFF lists; as long as the nondeterminism is purely syntactic, it introduces no problems into the verification. An example of

such a nondeterminism is the edges l and m of figure 1.

The proof is a straightforward application of the TL proof methodology in [Manna 82].

1. ϕ -> [at l_0 $\wedge$ dyad=false $\wedge$ monad=false $\wedge$ $|AML| = |AML_0|$]
   by PR

2. [at l_0 $\wedge$ dyad=false $\wedge$ monad=false $\wedge$ $|AML| = |AML|_0$]
   -> $\Diamond$[at l_1 $\wedge$ dyad=false $\wedge$ ((monad=false $\wedge$ $|AML| = |AML_0|$) $\vee$
   (monad=true $\wedge$ $|AML| = |AML|_0+1$))]
   by F_a, F_b, PR

3. [at l_1 $\wedge$ dyad=false $\wedge$ ((monad=false $\wedge$ $|AML| = |AML_0|$) $\vee$ (monad=true $\wedge$ $|AML = |AML|_0+1$))]
   -> $\Diamond$[at l_2 $\wedge$ dyad=false $\wedge$ ((monad=false $\wedge$ $|AML| = |AML|_0+1$) $\vee$
   (monad=true $\wedge$ $|AML| = |AML|_0+2$)]
   by F_c

4. [at l_2 $\wedge$ dyad=false $\wedge$ monad=false $\wedge$ $|AML| = |AML|_0+1$]
   -> $\Diamond$[at l_4 $\wedge$ dyad=false $\wedge$ monad=false $\wedge$ $|AML| = |AML|_0+1$]
   by F_f

5. [at l_2 $\wedge$ dyad=false $\wedge$ monad=true $\wedge$ $|AML| = |AML|_0+2$]
   -> $\Diamond$[at l_3 $\wedge$ dyad=false $\wedge$ monad=true $\wedge$ $|AML| = |AML|_0+1$]
   by F_d, def'n of mcomp

6. [at l_3 $\wedge$ dyad=false $\wedge$ monad=true $\wedge$ $|AML| = |AML|_0+1$]
   -> $\Diamond$[at l_4 $\wedge$ dyad=false $\wedge$ monad=false $\wedge$ $|AML| = |AML|_0+1$]
   by F_e, def'n of reset

7. [at l_4 $\wedge$ dyad=false $\wedge$ monad=false $\wedge$ $|AML| = |AML|_0+1$]
   -> $\Diamond$[at l_5 $\wedge$ monad=false $\wedge$ dyad=false $\wedge$ $|AML| = |AML|_0+1$]
   by F_g

8. [at l_5 $\wedge$ dyad=false $\wedge$ monad=false $\wedge$ $|AML| = |AML|_0+1$]
   -> $\Diamond$[at l_7 $\wedge$ dyad=false $\wedge$ monad=false $\wedge$ $|AML| = |AML|_0+1$]
   by F_j

9. [at l_7 ∧ dyad=false ∧ monad=false ∧ $|AML|=|AML|_0+1$]
   -> ◇[at l_8 ∧ monad=false ∧ dyad=false ∧ $|AML|=|AML|_0+1$]
   by F_k

There are now two cases:

CASE: The "addop" exists and the inner list is taken:

10. [at l_8 ∧ monad=false ∧ dyad=false ∧ $|AML|=|AML|_0+1$]
   -> ◇[at l_9 ∧ dyad=false ∧ monad=false ∧ $|AML|=|AML|_0+2$]
   by F_m

11. [at L_9 ∧ monad=false ∧ dyad=false ∧ $|AML|=|AML|_0+2$]
   -> ◇[at l_{10} ∧ monad=false ∧ dyad=true ∧ $|AML|=|AML|_0+2$]
   by F_n

12. [at L_{10} ∧ monad=false ∧ dyad=true ∧ $|AML|=|AML|_0+2$]
   -> ◇[at l_1 ∧ monad=false ∧ dyad=true ∧ $|AML|=|AML|_0+2$]
   by F_r

13. [at l_1 ∧ monad=false ∧ dyad=true ∧ $|AML|=|AML|_0+2$]
   -> ◇[at l_2 ∧ monad=false ∧ dyad=true ∧ $|AML|=|AML|_0+3$]
   by F_c

14. [at l_2 ∧ monad=false ∧ dyad=true ∧ $|AML|=|AML|_0+3$]
   -> ◇[at l_4 ∧ monad=false ∧ dyad=true ∧ $|AML|=|AML|_0+3$]
   by F_f

15. [at l_4 ∧ monad=false ∧ dyad=true ∧ $|AML|=|AML|_0+3$]
   -> ◇[at l_5 ∧ monad=false ∧ dyad=true ∧ $|AML|=|AML|_0+3$]
   by F_g

16. [at l_5 ∧ monad=false ∧ dyad=true ∧ $|AML|=|AML|_0+3$]
   -> ◇[at l_6 ∧ monad=false ∧ dyad=true ∧ $|AML|=|AML|_0+1$]
   by F_h, def'n of dcomp

17. $[\text{at } l_8 \wedge \text{monad=false} \wedge \text{dyad=true} \wedge |AML|=|AML|_0+1]$
$\rightarrow \Diamond[\text{at } l_7 \wedge \text{monad=false} \wedge \text{dyad=false} \wedge |AML|=|AML|_0+1]$
by F_i, def'n of reset

18. $[\text{at } l_7 \text{ monad=false} \wedge \text{dyad=false} \wedge |AML|=|AML|_0+1]$
$\rightarrow \Diamond[\text{at } l_8 \wedge \text{monad=false} \wedge \text{dyad=false} \wedge |AML|=|AML|_0+1]$
by F_k

19. $\Box[I_1]$
by 1..18, $\Diamond Q$, PR

CASE: The multop does not exist and the inner list is exited:

20. $[\text{at } l_8 \wedge \text{monad=false} \wedge \text{dyad=false} \wedge |AML|=|AML|_0+1]$
$\rightarrow \Diamond[\text{at } l_{11} \wedge \text{monad=false} \wedge \text{dyad=false} \wedge |AML|=|AML|_0+1]$
by F_l

once again there are two cases:

CASE: an addop is present:

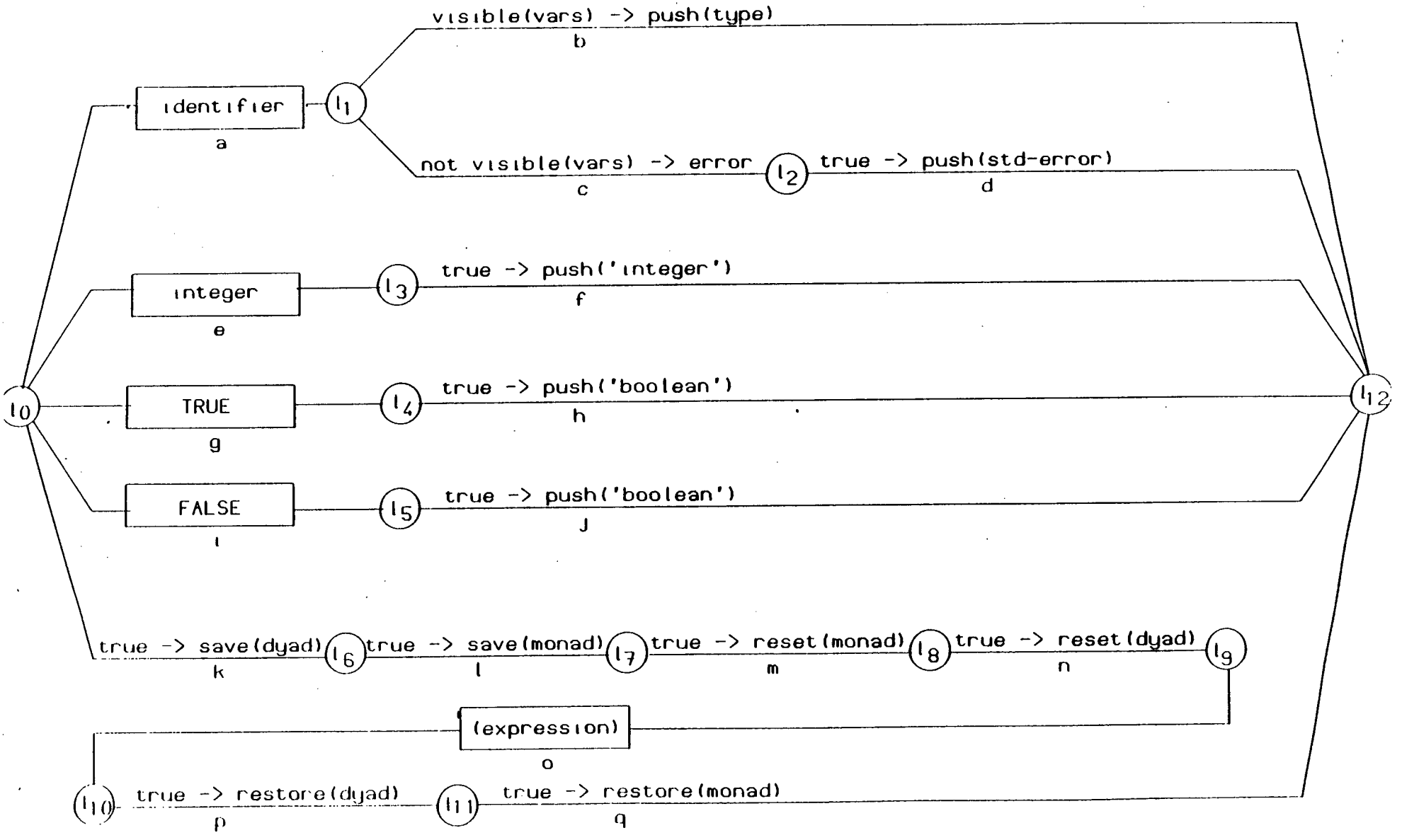
21. $[\text{at } l_{11} \wedge \text{monad=false} \wedge \text{dyad=false} \wedge |AML|=|AML|_0+1]$
$\rightarrow \Diamond[\text{at } l_{12} \wedge \wedge \text{monad=false} \wedge \text{dyad=false} \wedge |AML|=|AML|_0+2]$
by F_o

22. $[\text{at } l_{12} \wedge \text{monad=false} \wedge \text{dyad=false} \wedge |AML|=|AML|_0+2]$
$\rightarrow \Diamond[\text{at } l_{13} \wedge \text{monad=false} \wedge \text{dyad=true} \wedge |AML|=|AML|_0+2]$
by F_p

23. $[\text{at } l_{12} \wedge \text{monad=false} \wedge \text{dyad=true} \wedge |AML|=|AML|_0+2]$
$\rightarrow \Diamond[\text{at } l_{10} \wedge \text{monad=false} \wedge \text{dyad=true} \wedge |AML|=|AML|_0+2]$
by F_q

24. $[\text{at } l_{10} \wedge \text{monad=false} \wedge \text{dyad=true} \wedge |AML|=|AML|_0+2]$
$\rightarrow \Diamond[\text{at } l_{11} \wedge \text{monad=false} \wedge \text{dyad=false} \wedge |AML|=|AML|_0+1]$

figure 2 - factor



by 12..18, 20, ◇Q

25. $\Box[I_2]$

by 19, 22..24, ◇Q, PR

CASE: There is no addop

26. $[at\ l_{11} \wedge dyad=false \wedge monad=false \wedge |AML|=|AML|_0+1]$
$\rightarrow \Diamond[at\ l_{14} \wedge dyad=false \wedge monad=false \wedge |AML|=|AML|_0+1]$

by F_s

27. $[at\ l_{14} \wedge dyad=false \wedge monad=false \wedge |AML|=|AML|_0+1] \rightarrow \psi$

by PR

28. $\phi \rightarrow \Diamond\psi$

by 1..27, ◇Q

which completes the proof.

The proof for factor proceeds similarly. The graph is contained in figure 2. The transition axioms are:

F_a: $[at\ l_0] \rightarrow \Diamond[at\ l_1]$
F_b: $[at\ l_1 \wedge visible(vars) \wedge |AML|=u'] \rightarrow \Diamond[at\ l_{12} \wedge |AML|=u'+1]$
F_c: $[at\ l_1 \wedge \tilde{}visible(vars)] \rightarrow \Diamond[at\ l_2]$
F_d: $[at\ l_2 \wedge |AML|=u'] \rightarrow \Diamond[at\ l_{12} \wedge |AML|=u'+1]$
F_e: $[at\ l_0] \rightarrow \Diamond[at\ _3]$
F_f: $[at\ l_3 \wedge |AML|=u'] \rightarrow \Diamond[at\ l_{12} \wedge |AML|=u'+1]$
F_g: $[at\ l_0] \rightarrow \Diamond[at\ l_4]$
F_h: $[at\ l_4 \wedge |AML|=u'] \rightarrow \Diamond[at\ l_{12} \wedge |AML|=u'+1]$
F_i: $[at\ l_0] \rightarrow \Diamond[at\ l_5]$
F_j: $[at\ l_5 \wedge |AML|=u'] \rightarrow \Diamond[at\ l_{12} \wedge |AML|=u'+1]$
F_k: $[at\ l_0 \wedge |V\text{-}dyad|=u' \wedge dyad=u''] \rightarrow \Diamond[at\ l_6 \wedge |V\text{-}dyad|=u'+1 \wedge top(V\text{-}dyad)=u'']$
F_l: $[at\ l_6 \wedge |V\text{-}monad|=u' \wedge monad=u'']$
$\rightarrow \Diamond[at\ l_7 \wedge |V\text{-}monad|=u'+1 \wedge top(V\text{-}monad)=u'']$
F_m: $[at\ l_7] \rightarrow \Diamond[at\ l_8 \wedge monad=false]$
F_n: $[at\ l_8] \rightarrow \Diamond[at\ l_9 \wedge dyad=false]$

F_o: [at l_9 $\wedge$ |AML|=u' $\wedge$ dyad=u'' $\wedge$ monad=u''']
-> $\diamond$[at l_{10} $\wedge$ |AML|=u'+1 $\wedge$ dyad=u'' $\wedge$ monad=u''']
F_p: [at l_{10} $\wedge$ |V-dyad|=u' $\wedge$ top(V-dyad)=u] -> $\diamond$[at l_{11} $\wedge$ |V-dyad|=u'-1 $\wedge$ dyad=u]
F_q: [at l_{11} $\wedge$ |V-monad|=u' $\wedge$ top(V-monad)=u]
-> $\diamond$[at l_{12} $\wedge$ |V-monad|=u'-1 $\wedge$ monad=u]

we also use the following shorthands in the proof:

α: monad=$monad_0$
β: dyad=$dyad_0$
γ: |AML|=$|AML|_0$
δ: |AML|=$|AML|_0$+1

The proof:

1. ϕ -> [at l_0 $\wedge$ α $\wedge$ β $\wedge$ γ]
by PR

There are a number of cases, one corresponding to each of the syntactic structures in a factor.

CASE: identifier

2. [at l_0 $\wedge$ α $\wedge$ β $\wedge$ γ] -> $\diamond$[at l_1 $\wedge$ α $\wedge$ β $\wedge$ γ]
by F_a

There are now two subcases.

SUBCASE: The identifier is in the name table vars.

3. [at l_1 $\wedge$ visible(vars) $\wedge$ α $\wedge$ β $\wedge$ γ] -> $\diamond$[at l_{12} $\wedge$ α $\wedge$ β $\wedge$ δ]
by postcondition of function type, def'n of push, F_b

4. [at l_{12} $\wedge$ α $\wedge$ β $\wedge$ δ] -> ψ
by PR

SUBCASE: The identifier is not in the table.

5. $[\text{at } l_1 \wedge \tilde{}\text{visible(vars)} \wedge \alpha \wedge \beta \wedge \gamma] \rightarrow \Diamond[\text{at } l_2 \wedge \alpha \wedge \beta \wedge \gamma]$
by F_c, def'n of error

6. $[\text{at } l_2 \wedge \alpha \wedge \beta \wedge \gamma] \rightarrow \Diamond[\text{at } l_{12} \wedge \alpha \wedge \beta \wedge \delta]$
by F_d, def'n of push

7. $[\text{at } l_{12} \wedge \alpha \wedge \beta \wedge \delta] \rightarrow \psi$
by PR

CASE: integer

8. $[\text{at } l_0 \wedge \alpha \wedge \beta \wedge \gamma] \rightarrow \Diamond[\text{at } l_3 \wedge \alpha \wedge \beta \wedge \gamma]$
by F_e

9. $[\text{at } l_3 \wedge \alpha \wedge \beta \wedge \gamma] \rightarrow \Diamond[\text{at } l_{12} \wedge \alpha \wedge \beta \wedge \delta]$
by F_f, def'n of push

10. $[\text{at } l_{12} \wedge \alpha \wedge \beta \wedge \delta] \rightarrow \psi$

CASES: TRUE, FALSE: by analogy to the integer case above.

CASE: subexpression

11. $[\text{at } l_0 \wedge \alpha \wedge \beta \wedge \gamma \wedge |\text{V-dyad}| = |\text{V-dyad}|_0 \wedge |\text{V-monad}| = |\text{V-monad}|_0]$
$\rightarrow \Diamond[\text{at } l_6 \wedge \alpha \wedge \beta \wedge \gamma \wedge |\text{V-dyad}| = |\text{V-dyad}|_0 + 1$
$\wedge \text{top(V-dyad)} = \text{dyad}_0 \wedge |\text{V-monad}| = |\text{V-monad}|_0]$
by F_k. def'n of save

12. $[\text{at } l_6 \wedge \alpha \wedge \beta \wedge \gamma \wedge |\text{V-dyad}| = |\text{V-dyad}|_0 + 1 \wedge \text{top(V-dyad)} = \text{dyad}_0 \wedge |\text{V-monad}| = |\text{V-monad}|_0]$
$\rightarrow \Diamond[\text{at } l_6 \wedge \alpha \wedge \beta \wedge \gamma \wedge |\text{V-dyad}| = |\text{V-dyad}|_0 + 1 \wedge \text{top(V-dyad)} = \text{dyad}_0$
$\wedge |\text{V-monad}| = |\text{V-monad}|_0 + 1 \wedge \text{top(V-monad)} = \text{monad}_0]$
by F_l, def'n of save

13. $[\text{at } l_7 \wedge \alpha \wedge \beta \wedge \gamma \wedge |\text{V-dyad}| = |\text{V-dyad}|_0 + 1 \wedge \text{top(V-dyad)} = \text{dyad}_0$
$\wedge |\text{V-monad}| = |\text{V-monad}|_0 + 1 \wedge \text{top(V-monad)} = \text{monad}_0]$
$\rightarrow \Diamond[\text{at } l_8 \wedge \text{monad} = \text{false} \wedge \beta \wedge \gamma$
$\wedge |\text{V-dyad}| = |\text{V-dyad}|_0 + 1 \wedge \text{top(V-dyad)} = \text{dyad}_0$
$\wedge |\text{V-monad}| = |\text{V-monad}|_0 + 1 \wedge \text{top(V-monad)} = \text{monad}_0]$
by F_m, def'n of reset

14. $[\text{at } l_8 \wedge \text{dyad} = \text{false} \wedge \beta \wedge \gamma$
$\wedge |\text{V-dyad}| = |\text{V-dyad}|_0 + 1 \wedge \text{top(V-dyad)} = \text{dyad}_0$
$\wedge |\text{V-monad}| = |\text{V-monad}|_0 + 1 \wedge \text{top(V-monad)} = \text{monad}_0]$
$\rightarrow \Diamond[\text{at } l_9 \wedge \text{dyad} = \text{false} \wedge \text{monad} = \text{false} \wedge \gamma$
$\wedge |\text{V-dyad}| = |\text{V-dyad}|_0 + 1 \wedge \text{top(V-dyad)} = \text{dyad}_0$
$\wedge |\text{V-monad}| = |\text{V-monad}|_0 + 1 \wedge \text{top(V-monad)} = \text{monad}_0]$
by F_n, def'n of reset

15. $[\text{at } l_9 \wedge \text{dyad} = \text{false} \wedge \text{monad} = \text{false} \wedge \gamma$
$\wedge |\text{V-dyad}| = |\text{V-dyad}|_0 + 1 \wedge \text{top(V-dyad)} = \text{dyad}_0$
$\wedge |\text{V-monad}| = |\text{V-monad}|_0 + 1 \wedge \text{top(V-monad)} = \text{monad}_0]$
$\rightarrow \Diamond[\text{at } l_{10} \wedge \delta \wedge |\text{V-dyad}| = |\text{V-dyad}|_0 + 1$
$\wedge \text{top(V-dyad)} = \text{dyad}_0 \wedge |\text{V-monad}| = |\text{V-monad}|_0 + 1$
$\wedge \text{top(V-monad)} = \text{monad}_0]$
by F_o,
pre/post conditions
for expression

16. $[\text{at } l_{10} \wedge \delta \wedge |\text{V-dyad}| = |\text{V-dyad}|_0 + 1$
$\wedge \text{top(V-dyad)} = \text{dyad}_0 \wedge |\text{V-monad}| = |\text{V-monad}|_0 + 1$
$\wedge \text{top(V-monad)} = \text{monad}_0]$
$\rightarrow \Diamond[\text{at } l_{11} \wedge \alpha \wedge \delta$
$\wedge |\text{V-dyad}| = |\text{V-dyad}|_0 \wedge |\text{V-monad}| = |\text{V-monad}|_0 + 1$
$\wedge \text{top(V-monad)} = \text{monad}_0]$
by F_p, def'n of restore

17. $[\text{at } l_{11} \wedge \alpha \wedge \delta$
$\wedge |\text{V-monad}| = |\text{V-monad}|_0 + 1 \wedge \text{top(V-monad)} = \text{monad}_0]$

$-> \Diamond[\text{at } l_{12} \wedge \alpha \wedge \beta \wedge \delta$
$\wedge\ |\text{V-monad}| = |\text{V-monad}|_0 \wedge \text{top(V-monad)} = \text{monad}_0]$
by F_q, def'n of restore, PR

18. $[\text{at } l_{12} \wedge \alpha \wedge \beta \wedge \delta] -> \psi$
by PR

19. $\phi -> \Diamond\psi$ by 1..19, $\Diamond$Q, PR

which concludes the proof.

The rule sign is proved next. The transition diagram is figure 3.

The transition axioms are:

F_a: $[\text{at } l_0] -> \Diamond[\text{at } l_1]$
F_b: $[\text{at } l_1] -> \Diamond[\text{at } l_2 \wedge \text{monad=true}]$
F_c: $[\text{at } l_2 \wedge |\text{AML}| = u'] -> \Diamond[\text{at } l_5 \wedge |\text{AML}| = u'+1]$
F_d: $[\text{at } l_0] -> \Diamond[\text{at } l_3]$
F_e: $[\text{at } l_3] -> \Diamond[\text{at } l_4 \wedge \text{monad=true}]$
F_f: $[\text{at } l_4 \wedge |\text{AML}| = u'] -> \Diamond[\text{at } l_5 \wedge |\text{AML}| = u'+1]$

The proof:

1. $\phi -> [\text{at } l_0 \wedge |\text{AML}| = |\text{AML}_0]$
by PR

CASE: the sign is a +

2. $[\text{at } l_0 \wedge |\text{AML}| = |\text{AML}|_0]$
$-> \Diamond[\text{at } l_1 \wedge |\text{AML}| = |\text{AML}|_0]$
by F_a

3. $[\text{at } l_1 \wedge |\text{AML}| = |\text{AML}|_0]$
$-> \Diamond[\text{at } l_2 \wedge \text{monad=true} \wedge |\text{AML}| = |\text{AML}|_0]$

---

procedure which is proven as the next example.

by F_b, def'n of set

4. [at l_2 ∧ monad=true ∧ |AML|=|AML|$_0$]
   -> ◇[at l_5 ∧ monad=true ∧ |AML|=|AML|$_0$+1 ∧ top(AML)='+']
   by F_c, def'n of push

5. [at l_5 ∧ monad=true ∧ |AML|=|AML$_0$+1] -> ψ
   by PR

CASE: the sign is a -

6. [at l_0 ∧ |AML|=|AML|$_0$]
   -> ◇[at l_3 ∧ |AML|=|AML|$_0$]
   by F_d

7. [at l_3 ∧ |AML|=|AML|$_0$]
   -> ◇[at l_4 ∧ monad=true ∧ |AML|=|AML|$_0$]
   by F_e, def'n of set

8. [at l_4 ∧ monad=true ∧ |AML|=|AML|$_0$]
   -> ◇[at l_5 ∧ monad=true ∧ |AML|=|AML|$_0$+1 ∧ top(AML)='-']
   by F_f, def'n of push

9. [at l_5 ∧ monad=true ∧ |AML|=|AML$_0$+1] -> ψ
   by PR

10. ϕ -> ◇ψ
   by 1,2..5,6..7,◇Q,PR

The remaining rules, addop and multop are simplifications of the proof for sign.

The whole of the rule expression and all the nonterminals it uses, have now been proven correct. We now know that any evaluation of an expression will terminate correctly with only one element on the stack. The function type must also be proven correct. This is however a trivial exercise and is subsumed into the proof a more complex type evaluation procedure which is proven as the next example.

Before going on to this next example, some reflections on the meaningfulness and usefulness of the verifications are in order.

The verification process uncovered a number of errors and redundancies in the expression decoration. For example, three flags were used where two are adequate and it was not clear that dyad and monad both false were preconditions of the expression rule. Verifications at the metalanguage level are extremely helpful in the debugging of a language specification.

Chapter 17: VERIFYING A TYPE EVALUATION ALGORITHM

This chapter considers the verification of a type evaluation algorithm for Dijkstra's Language as presented in chapter 5, extended to have Pascal-style arrays. Arrays can have one subscript only, but may be of type 'array', so a 2-dimensional array has form A[i][j]. The extensions to the previous example are:

An identifier in a factor now has an optional continuation. A continuation is a list of subscripts. It must leave the flags dyad and monad, as well as the AML stack unaltered. The type of a subscripted array may itself be array, i.e. slicing is allowed.

The syntax of the new nonterminals is:

```
"continuation"
<[ if not( 'array' matches type )
   then error('invalid type')
 ]subscript_> %


"subscript"
[saveposition ;
 save(monad) ;
 save(dyad) ;
 reset(monad) ;
 reset(dyad) ;
 ]'['expression']'[ pop ;
                    amlscope ;
                    if rule = 'array'
                     then son(typeid) ;
                    type-evaluate ] %
```

The type evaluation procedure is:

```
procedure type-evaluate;

  begin
    type := '' ;
```

```
    evaluate
  end ;

procedure evaluate ;

  begin
    if (rule = 'vars') or (rule = 'types')
      then
       begin
         son(type) ;
         evaluate ;
         exit ;
       end ;

    if rule = 'type' then
      begin
        if contents = 'INT' then
                                  begin
                                    type := type + 'integer' ;
                                    exit ;
                                  end ;
        if contents = 'BOOL' then
                                  begin
                                    type := type + 'boolean'
                                    exit ;
                                  end ;
        if (contents <> 'INT') and (contents <> 'BOOL')
          then
            begin
              son(array)
              evaluate ;
              exit ;
            end ;
      end;

     if rule = typeid then
       begin
         if visible(types)
           then
             begin
               evaluate ;
               exit ;
             end
          else
            begin
              error('Invisible identifier') ;
              type := std-error ;
```

```
        end
    end ;

  if rule = 'array' then
    begin
      type := type + 'array'
      saveposition ;
      son(lbound) ;
      type := type + contents ;
      father(array)
      son(ubound)
      type := type + contents ;
      father(array) ;
      son(typeid) ;
      evaluate ;
      restoreposition ;
      exit ;
    end ;

end
```

The transition diagram for continuation is figure 7; for subscript, figure 8; and for the evaluate code, figure 6. The evaluation code is proven first.

The evaluation procedure works on structural equivalence, ie any two arrays [1..10] of INT, say, are the same type regardless of where they are declared, or if they are anonymous.

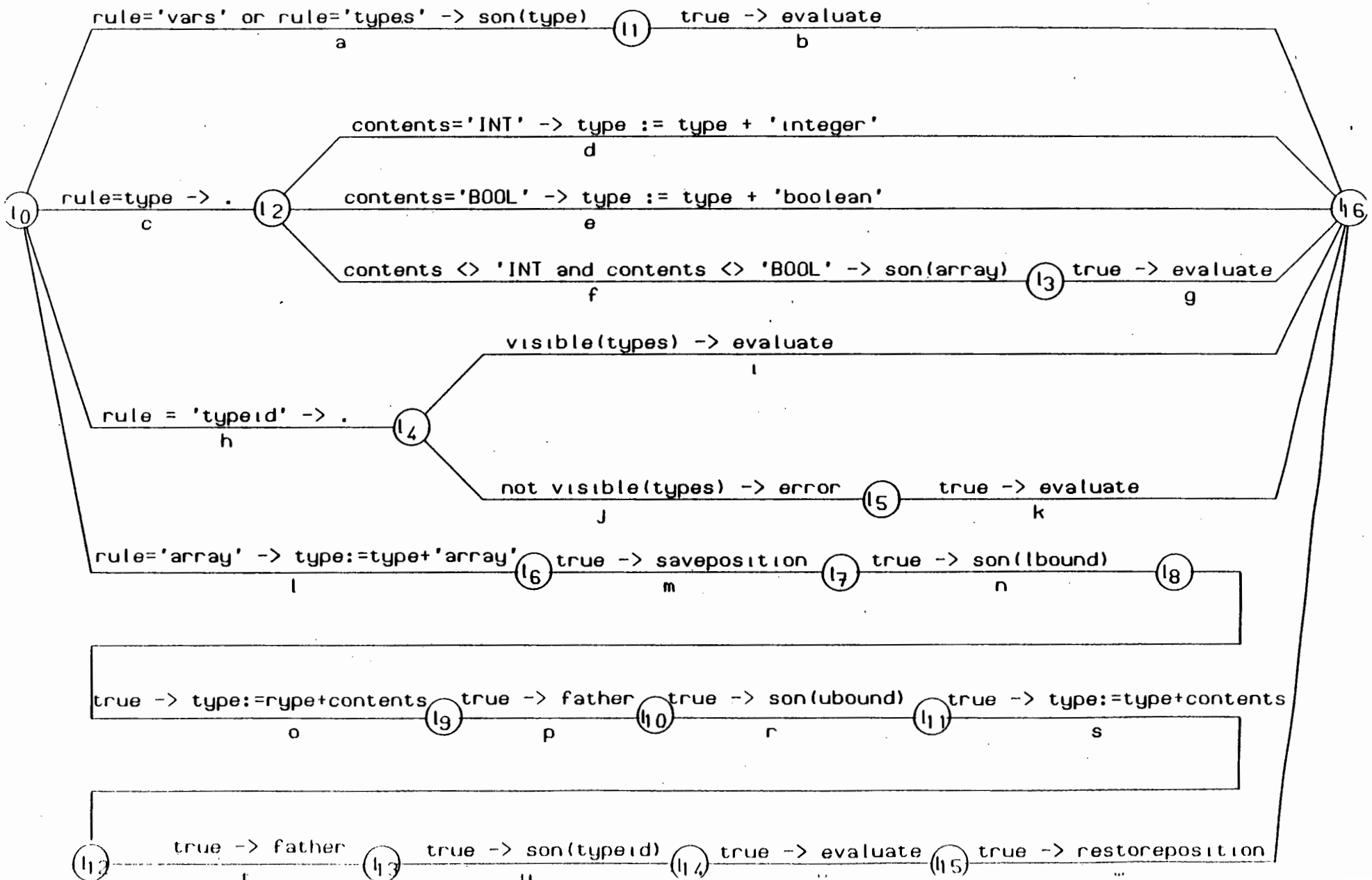
A "type template" for each possible type is defined. The evaluation code is considered to perform correctly if it terminates and generates a valid instantiation of a type template. The templates are referred to by the function TYPE (not to be confused with type, which is an AML string variable). The templates are:

INT: integer
BOOL: boolean
array: array + lbound + ubound + TYPE(typeid)
typeid: visible(types) -> TYPE(identifier), std–error
type: INT ¦ BOOL ¦ TYPE(array)
vars: TYPE(type)
types: TYPE(type)

The tells us, for example, than an ARRAY [1..10] of BOOL would have type 'array110boolean'[1] and that a VAR x : y has the type of y.

[1] This is potentially ambiguous. For example, consider the type 'array1112boolean'. Is this from ARRAY [1..1, 1..2] of BOOL or ARRAY [11..12] of BOOL? A simple way to resolve this ambiguity in practice is to place a distinguished charater between elements of the type (as we did with the

figure 6 - evaluate



For any other type, the TYPE is the null string ''.

The proof for the procedure type–evaluate is trivial and is not considered here; the only function of this procedure is to set the variable type to the null string and call evaluate.

Because the proof of evaluate is recursive it is significantly more complex than the proofs in the first examples above. This proof requires the use of the techniques for proving recursive programs developed in part II of this thesis.

In order that the proof work it is necessary to find a well founded set. The number of elements in the set should reflect the amount of work left to complete the type evaluation which is the number of recursions left in the evaluation process.

The set is constructed as follows:

- Consider the tree defining a type, with all user–defined identifiers replaced by the tree defining that user–defined type. As all identifiers must be defined before use in this language, there cannot be recursive types so this construction process must terminate. If an identifier reference cannot be resolved then replace it by a null pointer.
- Walk the tree in a depth–first, left–to–right order. At each node constructed from a vars, types, typeid, or array rule, or a type rule whose son is an array (as opposed to INT or BOOL) place an element in the set. However, do not place the root of the tree in the set. Call this set T.

In the verification process, we operate with this set, by considering that each time a SON is executed onto a node constructed from a rule with the properties mentioned above, the size of the set decreases by 1.

If the size of this set reflects the number of recursions still to be performed in the type evaluation process, then this set is a well founded set for the purposes of this verification, as when the last recursion is performed the set will be empty.

LEMMA: The size of T reflects the number of recursions still to be performed in the type evaluation process.

PROOF: This is true initially, as the number of recursions in the proof is the same as the number of nodes on the tree whose characteristics would result in an insertion into T when that set was constructed. This property is maintained by the recursions, as each possible recursion is prefaced by a SON onto another node which would have

---

record type evaluation routine in chapter 7).

caused an element to be placed in T, thus reducing T by 1.

LEMMA: T is a well founded set for the purpose of this verification.

PROOF: T always holds a number of elements equal to the number of recursions remaining in the type evaluation process. T therefore decreases as the amount of work needed to complete the type evaluation decreases. The size of T can therefore be used in the rule IND to indicate that each recursion brings the termination of the type evaluation algorithm closer. T is therefore suitable as a well founded set for this verification.

The number of elements in T is represented by |T|.

The insertion points for the name tables are:

$$IP(vars) = \{vars\}$$
$$IP(types) = \{types\}$$

The transition axioms for the transition diagram in figure 6 are:

F_a: [at l_0 ∧ (amlp='vars' ∨ amlp='types')] -> ◇[at l_1 ∧ amlp=type]

F_b: [at l_1] -> ◇[at l_0]

F_c: [at l_0 ∧ amlp='type'] -> ◇[at l_2 ∧ rule = 'type']

F_d: [at l_2 ∧ contents='INT' ∧ type=u] -> ◇[at l_{16} ∧ type =u+'integer']

F_e: [at l_2 ∧ contents='BOOL' ∧ type=u] -> ◇[at l_{16} ∧ type=u+'boolean']

F_f: [at l_2 ∧ contents≠'INT' ∧ contents≠'BOOL'] -> ◇[at l_3 ∧ amlp='array']

F_g: [at l_3] -> ◇[at l_0]

F_h: [at l_0 ∧ amlp='typeid'] -> ◇[at l_4 ∧ amlp='typeid']

F_i: [at l_4 ∧ visible(types)] -> ◇[at l_0 ∧ amlp∈IP(types)]

F_j: [at l_4 ∧ ~visible(types)] -> ◇[at l_5]

F_k: [at l_5 ∧ type=u] -> ◇[at l_{16} ∧ type=std-error]

F_l: [at l_0 ∧ amlp='array' ∧ type=u] -> ◇[at l_6 ∧ type =u+'array']

F_m: [at l_6 ∧ |P|=u ∧ amlp=u'] -> ◇[at l_7 ∧ |P|=u+1 ∧ top(P)=u']

F_n: [at l_7] -> ◇[at l_8 ∧ amlp='lbound']

F_o: [at l_8 ∧ type=u] -> ◇[at l_9 ∧ type=u+contents]

F_p: [at l_9] -> ◇[at l_{10} ∧ amlp='array']

F_r: [at l_{10}] -> ◇[at l_{11} ∧ amlp='ubound']

F_s: [at l_{11} ∧ type=u] -> ◇[at l_{12} ∧ type=u+contents]

F_t: [at l_{12}] -> ◇[at l_{13} ∧ amlp='array']

F_u: [at l_{13} ∧ amlp=u'] -> ◇[at l_{14} ∧ amlp='typeid']

F_v: $[\text{at } l_{14}] \rightarrow \Diamond[\text{at } l_0]$

F_w: $[\text{at } l_{15} \wedge |P|=u \wedge \text{top}(P)=u']$
$\rightarrow \Diamond[\text{at } l_{16} \wedge \text{amlp}=u' \wedge |P|=u-1 \wedge \text{scope}(\text{amlp})=\text{scope}(u')]$

The pre and post conditions of the proof are:

ϕ: $\text{type}_c = '' \wedge \text{switch}=\text{amlp}$

ψ: $\text{type} = \text{TYPE}(\text{amlp}_c)$

we also use the shorthand:

α: $\text{TYPE}(\text{amlp}_c)=\text{type} + \text{TYPE}(\text{amlp})$

The proof proceeds in two parts:

(a) $\phi \rightarrow \exists k Q(k)$
(b) $Q(k) \rightarrow \Diamond\psi$

$\phi \rightarrow \Diamond\psi$ by $\Diamond Q$ rule

where Q(k) is:

$Q(k)$: $[\text{at } l_0 \wedge 0 \leq |T| \leq k \wedge \alpha]$

proof of (a)

1. $\phi \rightarrow [\text{at } l_0 \wedge \text{type}_c='' \wedge \text{amlp}=\text{amlp}_c \wedge \text{type}=\text{type}_c]$
by PR

2. $|T| \geq 0$
initially by definition

3. $0 \leq |T| \leq |T|$
by PR

4. $[amlp=amlp_c] \rightarrow TYPE(amlp)=TYPE(amlp_c)$
by domain

5. $TYPE(amlp) = '' + TYPE(amlp)$
by domain

6. $[at\ l_0 \wedge amlp=amlp_c \wedge type=type_c \wedge type_c='']$
$\rightarrow [at\ l_0 \wedge 0 \leq |T| \leq |T| \wedge \alpha]$
by domain, PR

7. $[at\ l_0 \wedge 0 \leq |T| \leq |T| \wedge \alpha]$
$\rightarrow \exists k.[at\ l_0 \wedge 0 \leq |T| \leq k \wedge \alpha]$
by T24 of [Manna 82]

8. $\phi \rightarrow \exists k.Q(k)$
by 1,6,7,MP,PR

Proof of (b) proceeds in two parts:

(b1) $Q(0) \rightarrow \Diamond\psi$
(b2) $Q(m+1) \rightarrow \Diamond Q(m) \vee \Diamond\psi$

$Q(k) \rightarrow \Diamond\psi$ by IND rule.

proof of (b1)

9. $Q(0) \rightarrow [at\ l_0 \wedge |T|=0 \wedge \alpha]$
by PR

10. $|T|=0$
$\rightarrow$ (amlp='type' $\wedge$ (contents='BOOL' $\vee$ contents='INT')) $\vee$ amlp='typeid' $\wedge$ ~(visible(types)))
by def'n of T

CASE: amlp='type' $\wedge$ (contents='BOOL' $\vee$ contents='INT')

11. $[\text{at } l_0 \wedge |T|=0 \wedge \alpha] \rightarrow [\text{at } l_0 \wedge \text{amlp='type'} \wedge |T|=0 \wedge (\text{contents='INT'} \vee \text{contents='BOOL'}) \wedge \alpha]$
by PR

12. $[\text{at } l_0 \wedge \text{amlp='type'} \wedge |T|=0 \wedge (\text{contents='INT'} \vee \text{contents='BOOL'}) \wedge \alpha] \rightarrow \Diamond[\text{at } l_2 \wedge \text{amlp='type'} \wedge |T|=0 \wedge (\text{contents='INT'} \vee \text{contents='BOOL'}) \wedge \alpha]$
by F_c

13. TYPE('INT') = 'integer' $\wedge$ TYPE('BOOL') = 'boolean'
by def'n of type templates

There are now two subcases.

SUBCASE: contents='INT'

14. $[\text{at } l_2 \wedge \text{amlp='type'} \wedge |T|=0 \wedge \text{contents='INT'} \wedge \text{type}=u \wedge \alpha] \rightarrow \Diamond[\text{at } l_{16} \wedge \text{type}=u+\text{'integer'} \wedge \text{TYPE}(\text{amlp}_c)=\text{type}]$
by F_d, PR

15. $[\text{at } l_{16} \wedge \text{TYPE}(\text{amlp}_c) = \text{type}] \rightarrow \psi$
by PR

SUBCASE: contents='BOOL'

16. $[\text{at } l_2 \wedge \text{amlp='type'} \wedge |T|=0 \wedge \text{contents='BOOL'} \wedge \text{type}=u \wedge \alpha] \rightarrow \Diamond[\text{at } l_{16} \wedge \text{type}=u+\text{'boolean'} \wedge \text{TYPE}(\text{amlp}_c)=\text{type}]$
by F_e, PR

17. $[\text{at } l_{16} \wedge \text{TYPE}(\text{amlp}_c) = \text{type}] \rightarrow \psi$
by PR

CASE: amlp='typeid' $\wedge$ ~(visible(types))

17a. $[\text{at } l_0 \wedge \text{amlp='typeid'} \wedge \sim\text{visible(types)} \wedge |T|=0 \wedge \alpha] \rightarrow \Diamond[\text{at } l_5 \wedge \text{amlp='typeid'} \wedge |T|=0 \wedge \alpha]$

by F_h, F_j, PR

17b. [at $l_5 \wedge |T|=0 \wedge$ amlp='typeid' $\wedge \alpha$]
-> [at $l_{16} \wedge$ type=std-error $\wedge \alpha$]
by F_k, PR

17c. [at $l_{16} \wedge$ type=std-error $\wedge \alpha$] -> ψ
by PR

18. $Q(0) \rightarrow \Diamond\psi$
by 9..17c, PR, $\Diamond Q$

which concludes the proof of (b1)

Proof of (b2):

19. $Q(m+1) \rightarrow$ [at $l_0 \wedge 0 \leq |T| \leq m+1 \wedge \alpha$]
by PR

CASE: $|T|=0$

20. $Q(m+1) \wedge |T|=0 \rightarrow \Diamond\psi$
by 9..17, PR, $\Diamond Q$

CASE: $|T|>0$

There are a number of subcases, one for each branch out of l_0.

SUBCASE 1: amlp='vars' $\vee$ amlp='types'

21. rule='vars' -> amlp='vars'
by def'n of rule.

22. rule='types' -> amlp='types'
by def'n of rule.

23. [at l_0 ∧ (amlp='vars' ∨ amlp='types') ∧ $0 \leq |T| \leq m+1$ ∧ α]
    -> ◇[at l_1 ∧ amlp='type' ∧ $0 \leq |T| \leq m$ ∧ α]
    by F_a, syntax of types and vars rules
    def'n of T,
    def'n of type templates

24. [at l_1 ∧ amlp='type' ∧ $0 \leq |T| \leq m$ ∧ α]
    -> [at l_1 ∧ $0 \leq |T| \leq m$ ∧ α]
    by PR

25. [at l_1 ∧ $0 \leq |T| \leq m$ ∧ α]
    -> ◇[at l_0 ∧ $0 <= |T| \leq m$ ∧ α]
    by F_b

26. [Q(m+1) ∧ $|T| > 0$ ∧ (amlp='vars' ∨ amlp='types')] -> ◇Q(m)
    by 23..25, ◇Q, PR

SUBCASE 2: amlp='type'

27. [Q(m+1) ∧ amlp='type'] -> [at l_0 ∧ $0 \leq |T| \leq m+1$ ∧ amlp='type' ∧ α]
    by PR

28. [at l_0 ∧ $0 \leq |T| \leq m+1$ ∧ amlp='type' ∧ α]
    -> ◇[at l_2 ∧ $0 \leq |T| \leq m+1$ ∧ amlp='type' ∧ α]
    by F_c

There are now 3 cases:

CASE: contents='INT'

29. [at l_2 ∧ amlp='type' ∧ $0 \leq |T| \leq m+1$ ∧ contents='INT' ∧ type=u ∧ α]
    -> ◇[at l_{16} ∧ type=u+'integer' ∧ TYPE($amlp_c$)=type]
    by F_d,
    def'n of type templates

30. [at $l_{16} \wedge$ TYPE(amlp$_c$) = type] -> ψ
by PR

CASE: contents='BOOL'

31. [at $l_2 \wedge$ amlp='type' $\wedge$ $0 \leq |T| \leq m+1 \wedge$ contents='BOOL' $\wedge$ type=u $\wedge \alpha$]
-> $\diamond$[at $l_{16} \wedge$ type=u+'boolean' $\wedge$ TYPE(amlp$_c$)=type]
by F_e,
def'n of type templates

32. [at $l_{16} \wedge$ TYPE(amlp$_c$) = type] -> ψ

CASE: contents $\neq$ 'INT' $\wedge$ contents $\neq$ 'BOOL'

31. [at $l_2 \wedge$ amlp='type' $\wedge$ $0 \leq |T| \leq m+1 \wedge \alpha \wedge$ contents$\neq$'INT' $\wedge$ contents$\neq$'BOOL']
-> $\diamond$[at $l_3 \wedge$ amlp='array' $\wedge$ $0 \leq |T| \leq m \wedge \alpha$]
by F_f, def'n of son syntax of type rule
def'n of T, def'n of type templates

32. [at $l_3 \wedge$ $0 \leq |T| \leq m \wedge \alpha \wedge$ amlp='array']
-> [at $l_3 \wedge$ $0 \leq |T| \leq m \wedge \alpha$]
by PR

33. [at $l_3 \wedge$ $0 \leq |T| \leq m \wedge \alpha$]
-> $\diamond$[at $l_0 \wedge$ $0 \leq |T| \leq m \wedge \alpha$]
by F_g

34. [Q(m+1) $\wedge$ rule='type'] -> $\diamond\psi \vee \diamond$Q(m)
by PR, $\diamond$Q

SUBCASE 3: rule='typeid'

35. [Q(m+1) $\wedge$ amlp='typeid']
-> [at $l_0 \wedge$ $0 \leq |T| \leq m+1 \wedge \alpha \wedge$ amlp='typeid']
by PR

36. $[\text{at } l_0 \wedge 0 \leq |T| \leq m+1 \wedge \alpha \wedge \text{amlp}=\text{'typeid'}]$
$\rightarrow \Diamond[\text{at } l_4 \wedge 0 \leq |T| \leq m+1 \wedge \alpha \wedge \text{amlp}=\text{'typeid'}]$
by F_h

There are now two cases:

CASE: The type identifier is visible in the name table 'types'

37. $[\text{at } l_4 \wedge 0 \leq |T| \leq m+1 \wedge \alpha \wedge \text{amlp}=\text{'typeid'} \wedge \text{visible(types)}]$
$\rightarrow \Diamond[\text{at } l_4 \wedge \text{amlp}=\text{'types'} \wedge 0 \leq |T| \leq m \wedge \alpha]$
by def'n of visible, IP(types),
def'n of type templates, def'n of T

38. $[\text{at } l_4 \wedge \text{amlp}=\text{'types'} \wedge \alpha \wedge 0 \leq |T| \leq m]$
$\rightarrow [\text{at } l_4 \wedge \alpha \wedge 0 \leq |T| \leq m]$
by PR

39. $[\text{at } l_4 \wedge 0 \leq |T| \leq m \wedge \alpha]$
$\rightarrow \Diamond[\text{at } l_0 \wedge 0 \leq |T| \leq m \wedge \alpha]$
by F_i

CASE: The type identifier is not visible in the name table 'types'

40. $[\text{at } l_4 \wedge 0 \leq |T| \leq m+1 \wedge \alpha \wedge \text{amlp}=\text{'typeid'} \wedge \tilde{}\text{visible(types)}]$
$\rightarrow \Diamond[\text{at } l_5 \wedge 0 \leq |T| \leq m+1 \wedge \alpha \wedge \tilde{}\text{visible(types)}]$
by F_j

41. $[\text{at } l_5 \wedge \tilde{}\text{visible(types)} \wedge 0 \leq |T| \leq m+1 \wedge \alpha]$
$\rightarrow \Diamond[\text{at } l_{16} \wedge \text{type}=\text{std-error} \wedge \alpha]$
by F_k,
def'n of type templates

42. $[Q(m+1) \wedge \text{amlp}=\text{'typeid'}] \rightarrow \Diamond\psi \vee \Diamond Q(m)$
by 20..41, PR, $\Diamond Q$

SUBCASE 4: rule = 'array'

48. [Q(m+1) ∧ amlp='array']
    -> [at l_0 ∧ $0 \leq |T| \leq m+1$ ∧ α ∧ amlp='array' ∧ $type_0$=type]
    by PR

49. [at l_0 ∧ $0 \leq |T| \leq m+1$ ∧ α ∧ amlp='array' ∧ $type_0$=type]
    -> [at l_0 ∧ amlp='array' ∧ $0 \leq |T| \leq m+1$ ∧ α ∧ type=$type_0$ ∧ amlp=u2
    ∧ TYPE($amlp_c$)=$type_0$+TYPE(u2)]
    by PR

50. [at l_0 ∧ amlp='array' ∧ $0 \leq |T| \leq m+1$ ∧ α ∧ type=$type_0$ ∧ amlp=u2
    ∧ TYPE($amlp_c$)=$type_0$+TYPE(u2)]
    -> ◇[at l_6 ∧ type=$type_0$+'array' ∧ amlp=u2 ∧ $0 \leq |T| \leq m+1$
    ∧ TYPE($amlp_c$)=$type_0$+TYPE(u2)]
    by F_1, PR

51. [at l_6 ∧ type=$type_0$+'array' ∧ amlp=u2 ∧ |P|=u
    ∧ TYPE($amlp_c$)=$type_0$+TYPE(u2) ∧ $0 \leq |T| \leq m+1$]
    -> ◇[at l_7 ∧ $0 \leq |T| \leq m+1$ ∧ amlp=u2 ∧ amlp='array'
    ∧ type=$type_0$+'array' ∧ TYPE($amlp_c$)=$type_0$+TYPE(u2)
    ∧ |P|=u+1 ∧ top(P)=u2]
    by F_m,
    def'n of saveposition

52. [at l_7 ∧ $0 \leq |T| \leq m+1$ ∧ amlp=u2 ∧ amlp='array'
    ∧ type=$type_0$+'array' ∧ TYPE($amlp_c$)=$type_0$+TYPE(u2)
    ∧ |P|=u+1 ∧ top(P)=u2]
    -> ◇[at l_8 ∧ $0 \leq |T| \leq m+1$ ∧ amlp='lbound'
    ∧ type=$type_0$+'array' ∧ TYPE($amlp_c$)=$type_0$+TYPE(u2)
    ∧ |P|=u+1 ∧ top(P)=u2]
    by F_n

53. [at l_8 ∧ $0 \leq |T| \leq m+1$ ∧ amlp='lbound'
    ∧ type=$type_0$+'array' ∧ TYPE($amlp_c$)=$type_0$+TYPE(u2)

$\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

-> $\diamond$[at l_9 $\wedge$ 0$\leq$|T|$\leq$m+1 $\wedge$ amlp='lbound' $\wedge$ type=$type_0$+'array'+lbound

$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(u2) $\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

by F_o

54. [at l_9 $\wedge$ 0$\leq$|T|$\leq$m+1 $\wedge$ amlp='lbound' $\wedge$ type=$type_0$+'array'+lbound

$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(u2) $\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

-> $\diamond$[at l_{10} $\wedge$ 0$\leq$|T|$\leq$m+1 $\wedge$ amlp='array' $\wedge$ type=$type_0$+'array'+lbound $\wedge$ amlp=u2

$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(u2) $\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

by F_p

55. [at l_{10} $\wedge$ 0$\leq$|T|$\leq$m+1 $\wedge$ amlp=u2 $\wedge$ amlp='array' $\wedge$ type=$type_0$+'array'+lbound

$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(u2) $\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

-> $\diamond$[at l_{11} $\wedge$ 0$\leq$|T|$\leq$m+1 $\wedge$ amlp='ubound' $\wedge$ type=$type_0$+'array'+lbound

$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(u2) $\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

by F_r

56. [at l_{11} $\wedge$ 0$\leq$|T|$\leq$m+1 $\wedge$ amlp='ubound' $\wedge$ type=$type_0$+'array'+lbound

$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(u2) $\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

-> $\diamond$[at l_{12} $\wedge$ 0$\leq$|T|$\leq$m+1 $\wedge$ amlp='ubound' $\wedge$ type=$type_0$+'array'+lbound+ubound

$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(u2) $\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

by F_s

57. [at l_{12} $\wedge$ 0$\leq$|T|$\leq$m+1 $\wedge$ amlp='ubound' $\wedge$ type=$type_0$+'array'+lbound+ubound

$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(u2) $\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

-> $\diamond$[at l_{13} $\wedge$ 0$\leq$|T|$\leq$m+1 $\wedge$ amlp='array' $\wedge$ type=$type_0$+'array'+lbound+ubound

$\wedge$ amlp=u2 $\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(u2) $\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

by F_t

58. [at l_{13} $\wedge$ 0$\leq$|T|$\leq$m+1 $\wedge$ amlp='array' $\wedge$ type=$type_0$+'array'+lbound+ubound $\wedge$ amlp=u2

$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(u2) $\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

-> $\diamond$[at l_{14} $\wedge$ 0$\leq$|T|$\leq$m $\wedge$ amlp='typeid' $\wedge$ type=$type_0$+'array'+lbound+ubound

$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(u2) $\wedge$ |P|=u+1 $\wedge$ top(P)=u2]

by F_u,

def'n of T

59. [TYPE($amlp_c$)=$type_0$+TYPE(u2) ∧ type=$type_0$+'array'+lbound+ubound
∧ u2='array' ∧ amlp='typeid']
-> [TYPE($amlp_c$)=type+TYPE(amlp)]
by def'n of type templates

60. [at l_{14} ∧ 0≤|T|≤m ∧ amlp='typeid' ∧ type=$type_0$+'array'+lbound+ubound
∧ TYPE($amlp_c$)=$type_0$+TYPE(u2) ∧ |P|=u+1 ∧ top(P)=u2]
-> [at l_{14} ∧ 0≤|T|≤m ∧ amlp='typeid' ∧ α]
by PR

61. [at l_{14} ∧ 0≤|T|≤m ∧ α]
-> ◇[at l_0 ∧ 0≤|T|≤m ∧ α]
by F_v

62. [Q(m+1) ∧ rule='array'] -> ◇Q(m)
by 20, 48..61, ◇Q, PR

63. Q(m+1) -> ◇ψ ∨ ◇Q(m) by 20,26,34,42,62, ◇Q, PR

64. Q(k) -> ◇ψ by 18, 63, IND

This does not quite complete the proof because the procedure is not entirely tail recursive. When a return from the procedure is effected the return point may either be l_{16} or l_{15}:

CASE: Return to l_{16}:

65. at l_{16} ∧ type($amlp_c$) = type
by PR, def'n of T

CASE: Return to l_{15}

66. [at l_{15} ∧ type($amlp_c$)=type ∧ |P|=u+1 ∧ top(P)=u2]
-> ◇[at l_{16} ∧ type($amlp_c$)=type ∧ amlp=u2='array']
by F_w,
def'n of restoreposition,

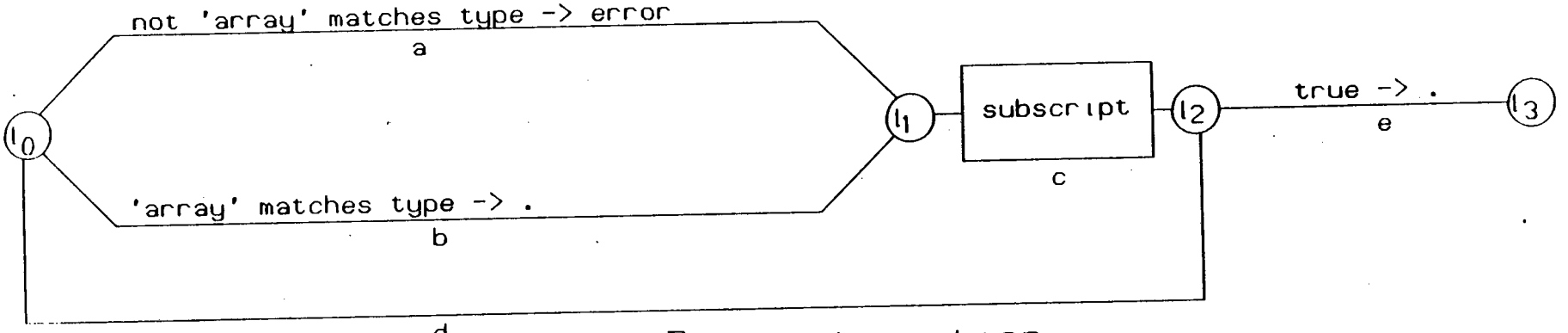


figure 7 - continuation

PR

Also, if the $amlp_c$ is not a types, vars, type, typeid or array, then the algorithm terminates trivially (this path is not shown in the figure), with

$$type=type_c=''$$

which also satisfies ψ.

This completes the proof that the procedure evaluate terminates with the correct result. Note that this is an excellent demonstration of the utility of the techniques developed in part II for proving recursive procedures; using the more traditional approach the procedure would have had to be considered a function and its least fixed point found (an unpleasant prospect). In fact the work in part II was developed entirely to obviate the necessity of doing so.

Consider now the transition diagrams in figures 7 and 8 this example demonstrates that the addition into expression of a factor with subscripts is correct.

The decoration for the rule "continuation" requires only that if a subscription is attempted on a non-array type, then an error is generated.

The postcondition asserts that the stack has been left unaltered and that the flags dyad and monad have been left unchanged.

This is an extremely easy rule to prove correct. The transition axioms for the graph in figure 6 are:

F_a: [at l_0 $\wedge$ ~('array' matches type)] -> $\Diamond$[at l_1]
F_b: [at l_0 $\wedge$ 'array' matches type] -> $\Diamond$[at l_1]
F_c: [at l_1 $\wedge$ |AML|=u $\wedge$ monad=u' $\wedge$ dyad=u'']
-> $\Diamond$[at l_2 $\wedge$ |AML|=u $\wedge$ monad=u' $\wedge$ dyad=u'']
F_d: [at l_2] -> $\Diamond$[at l_0]
F_e: [at l_2] -> $\Diamond$[at l_3]

The post condition is:

ψ: $|AML|=|AML|_0 \wedge dyad=dyad_0 \wedge monad=monad_0$

Once again we introduce some shorthands:

α: |AML|=|AML|$_0$

β: monad=monad$_0$

γ: dyad=dyad$_0$

The proof:

1. [at $l_0 \wedge \alpha \wedge \beta \wedge \gamma$]

   initially

2. [at $l_0 \wedge$ 'array' matches type $\wedge \alpha \wedge \beta \wedge \gamma$]
   -> $\Diamond$[at $l_1 \wedge \alpha \wedge \beta \wedge \gamma$]

   by F_b

3. [at $l_0 \wedge \sim$('array' matches type) $\wedge \alpha \wedge \beta \wedge \gamma$]
   -> $\Diamond$[at $l_1 \wedge \alpha \wedge \beta \wedge \gamma$]

   by F_a

4. [at $l_1 \wedge \alpha \wedge \beta \wedge \gamma$]
   -> $\Diamond$[at $l_2 \wedge \alpha \wedge \beta \wedge \gamma$]

   by F_c,
   conditions of "subscript"

CASE: Another subscript

5. [at $l_2 \wedge \alpha \wedge \beta \wedge \gamma$]
   -> $\Diamond$[at $l_0 \wedge \alpha \wedge \beta \wedge \gamma$]

   by F_d

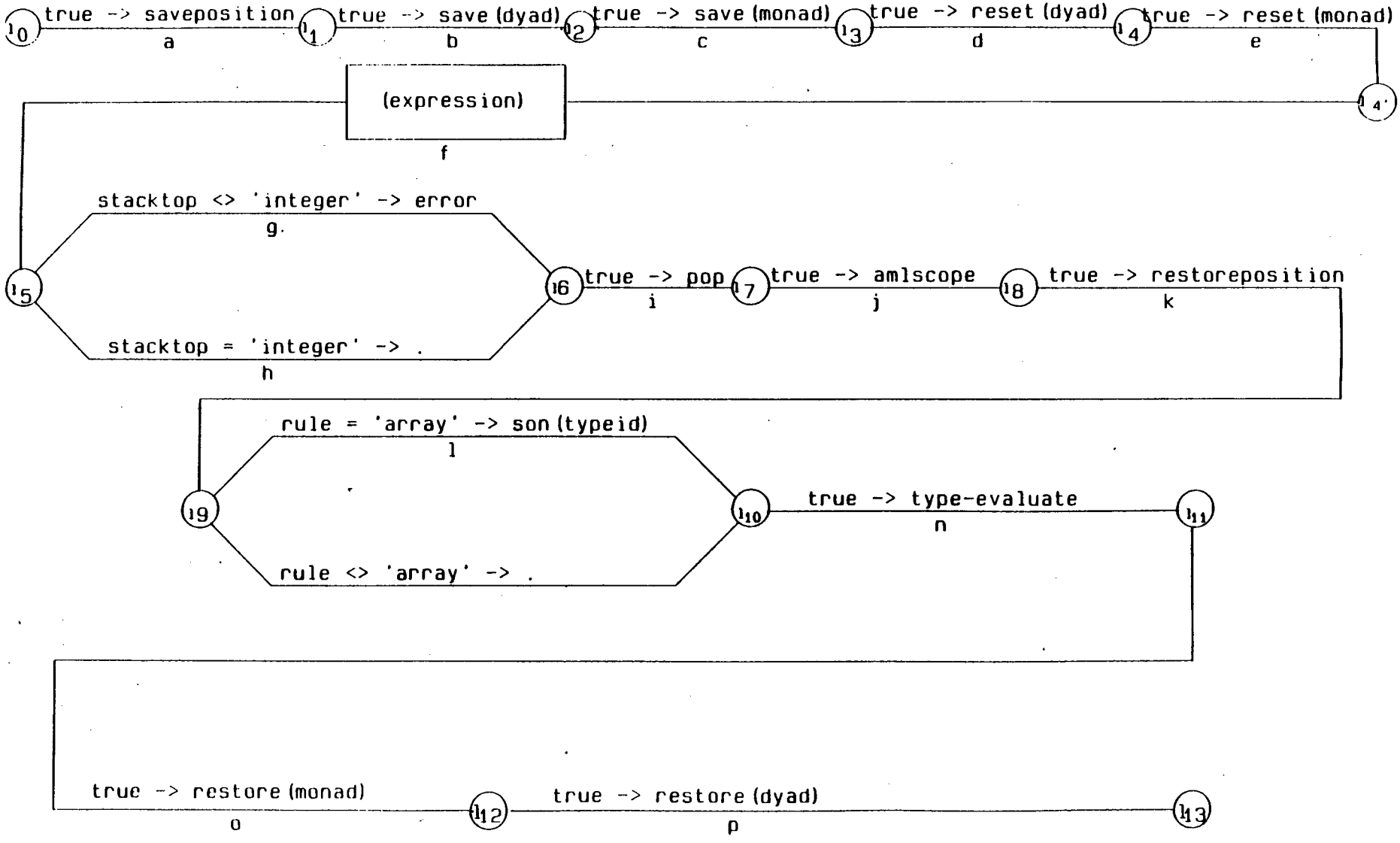
CASE: No more subscripts

6. [at $l_2 \wedge \alpha \wedge \beta \wedge \gamma$]
   -> $\Diamond$[at $l_3 \wedge \psi$]

   by F_e, PR

Note that we did not bother to associate an intermittent assertion with the exit point in the list.

figure 8 - subscript



The rule "subscript" may now be verified. This example illustrates just why the amlscope instruction is needed. It is also the first time that it is necessary to bother explicitly about the pointer switch in a proof.

The transition axioms are:

F_a: [at l_0 $\wedge$ |P|=u $\wedge$ amlp=u']
  -> $\diamond$[at l_1 $\wedge$ |P|=u+1 $\wedge$ top(P)=u']

F_b: [at l_1 $\wedge$ |V-dyad|=u $\wedge$ dyad=u']
  -> $\diamond$[at l_2 $\wedge$ |V-dyad|=u+1 $\wedge$ top(V-dyad)=u']

F_c: [at l_2 $\wedge$ |V-monad|=u $\wedge$ monad=u']
  -> $\diamond$[at l_3 $\wedge$ |V-monad|=u+1 $\wedge$ top(V-monad)=u']

F_d: [at l_3] -> $\diamond$[at l_4 $\wedge$ dyad=false]

F_e: [at l_4] -> $\diamond$[at $l_{4'}$ $\wedge$ monad=false]

F_f: [at $l_{4'}$ $\wedge$ monad=false $\wedge$ dyad=false $\wedge$ |AML|=u]
  -> $\diamond$[at l_5 $\wedge$ |AML|=u+1]

F_g: [at l_5 $\wedge$ stacktop$\neq$'integer'] -> $\diamond$[at l_6]

F_h: [at l_5 $\wedge$ stacktop='integer'] -> $\diamond$[at l_6]

F_i: [at l_6 $\wedge$ |AML|>0] -> $\diamond$[at l_7 $\wedge$ |AML|$\geq$0]

F_j: [at l_7] -> $\diamond$[at l_8 $\wedge$ switch=amlp]

F_k: [at l_8 $\wedge$ |P|=u $\wedge$ top(P)=u']
  -> $\diamond$[at l_9 $\wedge$ |P|=u-1 $\wedge$ amlp=u']

F_l: [at l_9 $\wedge$ amlp='array']
  -> $\diamond$[at l_{10} $\wedge$ amlp='typeid']

F_m: [at l_9] $\wedge$ amlp $\neq$ 'array']
  -> $\diamond$[at l_{10}]

F_n: [at l_{10} $\wedge$ switch=amlp $\wedge$ amlp=u]
  -> $\diamond$[at l_{11} $\wedge$ type=TYPE(u)]

F_o: [at l_{11} $\wedge$ |V-monad|=u $\wedge$ top(V-monad)=u']
  -> $\diamond$[at l_{12} $\wedge$ |V-monad|=u-1 $\wedge$ monad=u']

F_p: [at l_{12} $\wedge$ |V-dyad|=u $\wedge$ top(V-dyad)=u']
  -> $\diamond$[at l_{13} $\wedge$ |V-dyad|=u-1 $\wedge$ dyad=u']

The proof shows that:

ψ: monad=$monad_0$ $\wedge$ dyad=$dyad_0$ $\wedge$ |AML|=$|AML|_0$

$\wedge$ (amlp$_0$ = 'array' -> type = TYPE(typeid), type=type$_0$)

We use the shorthands:

α: monad=monad$_0$
β: dyad=dyad$_0$
γ: |AML|=|AML|$_0$

1. [at $l_0 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep]
initially

2. [at $l_0 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ |P|=u
$\wedge$ amlp=amlp$_0 \wedge$ switch=treep]
-> $\Diamond$[at $l_1 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ |P|=u+1
$\wedge$ top(P)=amlp$_0 \wedge$ switch=treep]
by F_a, def'n of saveposition

3. [at $l_1 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ |P|=u+1
$\wedge$ top(P)=amlp$_0 \wedge$ switch=treep]
-> [|P|=u+1 $\wedge$ top(P)=amlp$_0$] by PR

4. [at $l_1 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ |P|=u+1
$\wedge$ top(P)=amlp$_0 \wedge$ switch=treep]
-> [at $l_1 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep]
by PR

5. [at $l_1 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep
$\wedge$ |V-dyad|=u]
-> $\Diamond$[at $l_2 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep
$\wedge$ |V-dyad|=u+1 $\wedge$ top(V-dyad)=dyad$_0$]
by F_b, def'n of save

6. [at $l_2 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep
$\wedge$ |V-dyad|=u+1 $\wedge$ top(V-dyad)=dyad$_0$]
-> [|V-dyad|=u+1 $\wedge$ top(V-dyad)=dyad$_0$]

by PR

7. [at $l_2 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep
$\wedge$ |V-dyad|=u+1 $\wedge$ top(V-dyad)=$dyad_0$]
-> [at $l_2 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep]
by PR

8. [at $l_2 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep
$\wedge$ |V-monad|=u]
-> $\Diamond$[at $l_3 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep
$\wedge$ |V-monad|=u+1 $\wedge$ top(V-monad)=$monad_0$]
by F_c, def'n of save

9. [at $l_3 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep
$\wedge$ |V-monad|=u+1 $\wedge$ top(V-monad)=$monad_0$]
-> [|V-monad|=u+1 $\wedge$ top(V-monad)=$monad_0$]
by PR

10. [at $l_3 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep
$\wedge$ |V-monad|=u+1 $\wedge$ top(V-monad)=$monad_0$]
-> [at $l_2 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep]
by PR

11. [at $l_3 \wedge \alpha \wedge \beta \wedge \gamma \wedge$ switch=treep]
-> $\Diamond$[at $l_4 \wedge \alpha \wedge$ dyad=false $\wedge \gamma \wedge$ switch=treep]
by F_d,def'n of reset

12. [at $l_4 \wedge \alpha \wedge$ dyad=false $\wedge \gamma \wedge$ switch=treep]
-> $\Diamond$[at l_4, $\wedge$ dyad=false $\wedge$ monad=false
$\wedge$ switch=treep]
by F_e, def'n of reset

12'.[at l_4, $\wedge$ dyad=false $\wedge$ monad=false $\wedge \gamma \wedge$ switch=treep]
-> $\Diamond$[at $l_5 \wedge$ |AML|=$|AML|_0$+1 $\wedge$ switch=treep]
by F_f,

(Satisfies ϕ for expression),
Postcondition for expression,
Entering new AML decoration
after expression sets switch
to treep by definition.

13. $[\text{at } l_5 \wedge |AML| = |AML|_0 + 1 \wedge \text{switch} = \text{treep} \wedge \text{top}(AML) = \text{'integer'}]$
$\rightarrow \Diamond[\text{at } l_6 \wedge |AML| = |AML|_0 + 1 \wedge \text{switch} = \text{treep}]$
by F_h

14. $[\text{at } l_5 \wedge |AML| = |AML|_0 + 1 \wedge \sim(\text{top}(AML) = \text{'integer'})]$
$\rightarrow \Diamond[\text{at } l_6 \wedge |AML| = |AML|_0 + 1 \wedge \text{switch} = \text{treep}]$
by F_g

15. $[\text{at } l_6 \wedge \text{switch} = \text{treep} \wedge |AML| = |AML|_0 + 1]$
$\rightarrow \Diamond[\text{at } l_7 \wedge \gamma \wedge \text{switch} = \text{treep}]$
by F_i, def'n of pop

16. $[\text{at } l_7 \wedge \text{switch} = \text{treep} \wedge \gamma]$
$\rightarrow \Diamond[\text{at } l_8 \wedge \gamma \wedge \text{switch} = \text{amlp}]$
by F_j

17. $[\text{at } l_8 \wedge \gamma \wedge \text{switch} = \text{amlp}]$
$\rightarrow [\text{at } l_8 \wedge \text{switch} = \text{amlp} \wedge |P| = u + 1 \wedge \text{top}(P) = \text{amlp}_0 \wedge \gamma]$
by 3, PR

18. $[\text{at } l_8 \wedge \text{switch} = \text{amlp} \wedge |P| = u + 1 \wedge \text{top}(P) = \text{amlp}_0 \wedge \gamma]$
$\rightarrow \Diamond[\text{at } l_9 \wedge \text{switch} = \text{amlp} \wedge |P| = u \wedge \text{amlp} = \text{amlp}_0 \wedge \gamma]$
by F_k,
def'n of restoreposition

19. $[\text{at } l_9 \wedge \text{switch} = \text{amlp} \wedge |P| = u \wedge \text{amlp} = \text{amlp}_0 \wedge \gamma \wedge \text{amlp} = \text{'array'}]$
$\rightarrow \Diamond[\text{at } l_{10} \wedge \gamma \wedge \text{amlp} = \text{'typeid'} \wedge \text{switch} = \text{amlp}]$
by F_l, PR

20. [at $l_9 \wedge \gamma \wedge$ switch=amlp $\wedge$ |P|=u $\wedge$ amlp=$amlp_0 \wedge$ amlp $\neq$ 'array']
-> $\diamond$[at $l_{10} \wedge \gamma \wedge$ switch=amlp $\wedge$ amlp=$amlp_0$]
by F_m, PR

21. [at $l_{10} \wedge$ switch=amlp $\wedge$ (amlp='typeid' $\vee$ amlp=$amlp_0$) $\wedge$ amlp=u $\wedge \gamma$]
-> $\diamond$[at $l_{11} \wedge \gamma \wedge$ type=TYPE(u)
$\wedge$ ($amlp_0$='array' -> type=TYPE(typeid),type=TYPE($amlp_0$)]
by F_n, PCALL rule,
conditions on evaluation
procedure, PR

22. [at $l_{11} \wedge \gamma \wedge$ type=type($amlp_0$)
$\wedge$ ($amlp_0$='array' -> type=TYPE(typeid),type=TYPE($amlp_0$)]
-> $\diamond$[at $l_{12} \wedge \gamma \wedge \alpha$
$\wedge$ ($amlp_0$='array' -> type=TYPE(typeid),type=TYPE($amlp_0$)]
by F_o, 9, PR

23. [at $l_{12} \wedge \alpha \wedge \gamma$
$\wedge$ ($amlp_0$='array' -> type=TYPE(typeid),type=TYPE($amlp_0$)]
-> $\diamond$[at $l_{12} \wedge \alpha \wedge \beta \wedge \gamma$
$\wedge$ ($amlp_0$='array' -> type=TYPE(typeid),type=TYPE($amlp_0$)]
by F_p, 6, PR

24. [at $l_{12} \wedge \alpha \wedge \beta$
$\wedge$ ($amlp_0$='array' -> type=TYPE(typeid),type=TYPE($amlp_0$)]
-> ψ by PR

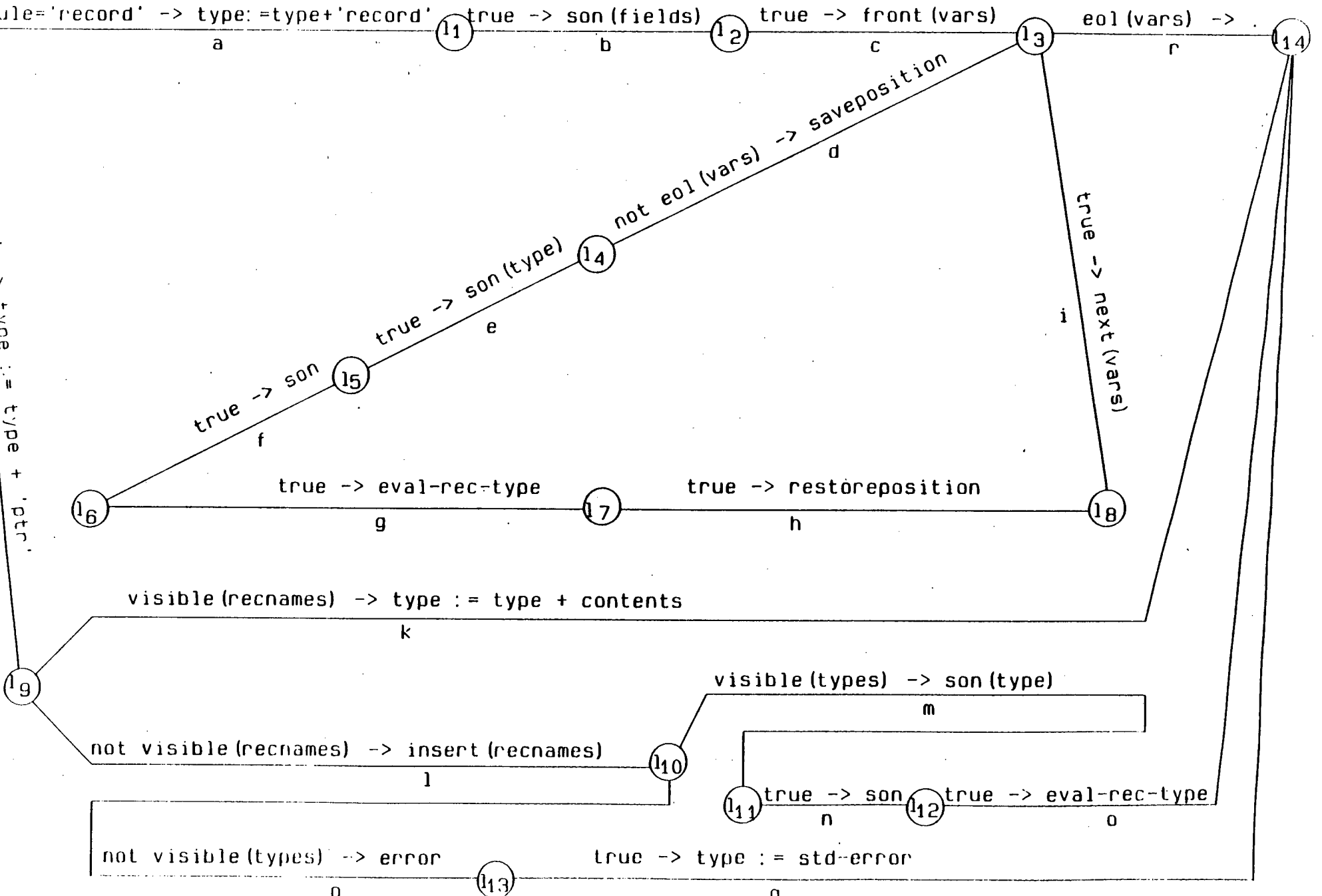
25. [at $l_0 \wedge \alpha \wedge \beta \wedge \gamma$] -> $\diamond\psi$
by $\diamond$Q

which completes the proof.

Note that this proof makes it clear that the decorations to save and restore monad and dyad, and set them to false, might more profitably be placed on the decoration for expression.

Once again, the proof was started on a slightly different version of the example and this version evolved as errors were found in the AML decorations during the verification process. With specifications reaching a reasonable degree of complexity it becomes virtually impossible to test them or to keep all the different possibilities in one's head during the design process. The verification is therefore vital as it allows one to catch the more subtle inconsistencies in a language specification.

figure 9 - record evaluate



Chapter 18: VERIFYING RECURSIVE TYPE EVALUATIONS

In chapter 7 above, a recursive record type and a type evaluation algorithm for it were introduced. This chapter proves that the type evaluation evaluation algorithm given in chapter 7 is correct. The reader may wish to re-read chapter 7 to recall the example fully. The graphical representation of the type evaluation procedure can be found in figure 9.

The type of a record consists of the string 'RECORD' concatenated with the types of the components of the record:

TYPE(record) = 'record'+TYPE(fields)

TYPE(fields) = ∀x∈vars:type+TYPE(x)

TYPE(type) = TYPE(record) ¦ TYPE(pointer)

```
TYPE(pointer) = if identifier∈recnames then 'ptr'+identifier
          if ~(identifier∈recnames)
           then
            begin
             insert(recnames)
             if identifier∈types then 'ptr'+TYPE(identifier)
                                 else std-error
            end
```

This is a particularly interesting procedure to prove because of its structure (see figure 9). The procedure has a recursive structure with a loop layered on as well, which makes the proof quite tricky.

The proof proceeds in a manner similar to that for the proof of the evaluation algorithm with arrays in the second example (figure 6) above. The first part of the proof involves the development of a well-founded set to use in in order to show that the procedure terminates.

We would like to use a similar set to that for the proof of the previous evaluation algorithm. However, in the previous examples the set depended on the fact that there could be no recursive data types, and that there could therefore be no infinite regression when constructing the set, so the construction of this set will be slightly different.

Once again consider the tree defining a type. The set is constructed by walking the tree in prefix order and placing each record or pointer node into the set. When a type identifier is encountered, if it is the first time that that identifier has been encountered then it is substituted by the subtree defining the type of the identifier and that subtree is

walked before continuing; otherwise the identifier is placed in the set and the walk continues.

As before the set is denoted by T and the size of the set by |T|. Walking a node of the tree corresponding to an element of the set in the type evaluation process removes the element from the set and decreases the size of the set by 1.

The pre and postconditions are:

ϕ: switch=amlp $\wedge$ $type_c$=''

γ: $TYPE(amlp_c)$ = type

We use the shorthand notation:

α: $TYPE(amlp_c)$ = type + TYPE(amlp)

Further, the IP sets for types and vars are assumed to be:

IP(vars) = {fields}

IP(types) = {types}

and also SC = {fields}

The CFF nonterminal "types" is assumed to have a similar syntax to that used in the previous examples.

The transition axioms for the proof are:

F_a: [at l_0 $\wedge$ amlp='record' $\wedge$ type=u]
    -> $\diamond$[at l_1 $\wedge$ type =u+'record']

F_b: [at l_1 $\wedge$ amlp='record']
    -> $\diamond$[at l_2 $\wedge$ amlp='fields' $\wedge$ scope(amlp)=NEWSCOPE(record)]

F_c: [at l_2] -> $\diamond$[at l_3 $\wedge$ vars'=vars $\wedge$ amlp='fields']

F_d: [at l_3 $\wedge$ ~eol(vars) $\wedge$ |P|=u1 $\wedge$ amlp=u2]
    -> $\diamond$[at l_4 $\wedge$ |P|=u1+1 $\wedge$ top(P)=u2]

F_e: [at l_4 $\wedge$ amlp=u1 $\wedge$ type$\in$RHS(u1)]
    -> $\diamond$[at l_5 $\wedge$ amlp='type']

F_f: [at l_5 $\wedge$ amlp=u1]

$\rightarrow \Diamond[\text{at } l_6 \wedge \text{amlp} \in \{\text{record,pointer}\}]$

F_g: $[\text{at } l_6] \rightarrow \Diamond[\text{at } l_0]$

F_h: $[\text{at } l_7 \wedge |P| = u1 \wedge \text{top}(P) = u2]$

$\rightarrow \Diamond[\text{at } l_8 \wedge |P| = u1-1 \wedge \text{amlp} = u2 \wedge \text{scope}(\text{amlp}) = \text{scope}(u2)]$

F_i: $[\text{at } l_8 \wedge \text{vars}' = u']$

$\rightarrow \Diamond[\text{at } l_3 \wedge \text{vars}' = \text{tail}(u') \wedge \text{amlp} = \text{'fields'}]$

F_j: $[\text{at } l_0 \wedge \text{amlp} = \text{'pointer'} \wedge \text{type} = u]$

$\rightarrow \Diamond[\text{at } l_9 \wedge \text{type} = u + \text{'ptr'}]$

F_k: $[\text{at } l_9 \wedge \text{visible(recnames)} \wedge \text{type} = u1 \wedge \text{contents} = u2]$

$\rightarrow \Diamond[\text{at } l_{14} \wedge \text{type} = u1 + u2]$

F_l: $[\text{at } l_9 \wedge \tilde{}\text{visible(recnames)}]$

$\rightarrow \Diamond[\text{at } l_{10} \wedge \text{visible(recnames)}]$

F_m: $[\text{at } l_{10} \wedge \text{visible(types)}]$

$\rightarrow \Diamond[\text{at } l_{11} \wedge \text{amlp} = \text{'type'}]$

F_n: $[\text{at } l_{11} \wedge \text{amlp} = u]$

$\rightarrow \Diamond[\text{at } l_{12} \wedge \text{amlp} \in \text{RHS}(u)]$

F_o: $[\text{at } l_{12}] \rightarrow \Diamond[\text{at } l_0]$

F_p: $[\text{at } l_{10} \wedge \tilde{}\text{visible(types)}] \rightarrow \Diamond[\text{at } l_{13}]$

F_q: $[\text{at } l_{13} \wedge \text{type} = u]$

$\rightarrow \Diamond[\text{at } l_{14} \wedge \text{type} = \text{std-error}]$

F_r: $[\text{at } l_3 \wedge \text{eol(vars)}] \rightarrow \Diamond[\text{at } l_{14}]$

The recursion is considered as the major problem in the proof, and the following assertion is used:

$$R(k) = [\text{at } l_0 \wedge 0 \leq |T| \leq k \wedge \text{TYPE}(\text{amlp}_c) = \text{type} + \text{TYPE}(\text{amlp})]$$

The loop on the path {a,b,c,d,e,f,g,h,i,r} is considered in the context of one possible path for the recursion. The assertion

$$L(l) = [\text{at } l_3 \wedge 0 \leq |\text{vars}'| \leq l \wedge \text{amlp} = \text{'fields'} \wedge \text{TYPE}(\text{amlp}_c) = \text{type} + \text{TYPE}(\text{amlp})]$$

is used to show termination of the loop.

The proof proceeds by showing:

(a) $\phi \rightarrow \exists k.R(k)$

(b) $R(k) \rightarrow \Diamond\gamma$

$\phi \rightarrow \Diamond\gamma$ by $\exists\Diamond Q$ rule

where $\psi = \gamma$.

Proof of (a):

1. ϕ -> [at $l_0 \wedge \alpha$]
   by PR

2. [at $l_0 \wedge \alpha$] -> [$|T| \geq 0$]
   by def'n of T

3. $|T| \geq 0$ -> $0 \leq |T| \leq |T|$ domain

4. [at $l_0 \wedge \alpha$] -> [at $l_0 \wedge \alpha \wedge 0 \leq |T| \leq |T|$]
   by PR

5. [at $l_0 \wedge \alpha \wedge 0 \leq |T| \leq |T|$]
   -> $\exists k$.[at $l_0 \wedge \alpha \wedge 0 \leq |T| \leq k$]
   by T24 of [Manna 82]

6. [at $l_0 \wedge \alpha \wedge 0 \leq |T| \leq k$] -> R(k)
   by PR

7. ϕ -> $\exists k$.R(k) by PR

Proof of (b) proceeds in two parts:

(b1) R(0) -> $\diamond\psi$
(b2) R(m+1) -> $\diamond$R(m) $\vee$ $\diamond\psi$

R(k) -> $\diamond\psi$) by IND rule.

Proof of (b1):

8. R(0) -> [at $l_0 \wedge \alpha \wedge |T| = 0$]

by PR

There are now three cases:

CASE: amlp='pointer' $\wedge$ ~(visible(recnames)) $\wedge$ ~(visible(types))

9. [at $l_0 \wedge \alpha \wedge |T|=0 \wedge$ amlp='pointer' $\wedge$ ~(visible(recnames)) $\wedge$ ~(visible(types))]
   -> $\Diamond$[at $l_9 \wedge$ type=$type_0$+'ptr' $\wedge$ TYPE($amlp_c$)=$type_0$+type(amlp)
   $\wedge$ ~(visible(recnames)) $\wedge$ ~(visible(types))]
   by F_j, PR

10. [at $l_9 \wedge$ type=$type_0$+'ptr' $\wedge$ TYPE($amlp_c$)=$type_0$+type(amlp)
   $\wedge$ ~(visible(recnames)) $\wedge$ ~(visible(types))]
   -> $\Diamond$[at $l_{10} \wedge$ type=$type_0$+'ptr' $\wedge$ TYPE($amlp_c$)=$type_0$+type(amlp)
   $\wedge$ ~(visible(types))]
   by F_l

11. [at $l_{10} \wedge$ type=$type_0$+'ptr' $\wedge$ TYPE($amlp_c$)=$type_0$+type(amlp)
   $\wedge$ ~(visible(types))]
   -> $\Diamond$[at $l_{13} \wedge$ type=$type_0$+'ptr' $\wedge$ TYPE($amlp_c$)=$type_0$+type(amlp)
   $\wedge$ ~(visible(types))]
   by F_p,
   def'n of error

12. [at $l_{13} \wedge$ type=$type_0$+'ptr' $\wedge$ TYPE($amlp_c$)=$type_0$+type(amlp)
   $\wedge$ ~(visible(types))]
   -> $\Diamond$[at $l_{14} \wedge$ type=std-error $\wedge \alpha$]
   by F_q,
   def'n of TYPE

13. [at $l_{14} \wedge \alpha$] -> ψ by PR

CASE: amlp='pointer' $\wedge$ visible(recnames)

14. [at $l_0 \wedge |T|=0 \wedge$ amlp='pointer' $\wedge \alpha$

$\wedge$ visible(recnames)]
$->$ $\diamond$[at l_9 $\wedge$ type=$type_0$+'ptr' $\wedge$ amlp='pointer' $\wedge$ visible(recnames)
$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE($amlp_0$)]
by F_j, PR,
def'n of TYPE

15. [at l_9 $\wedge$ type=$type_0$+'ptr' $\wedge$ amlp='pointer' $\wedge$ visible(recnames)
$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE($amlp_0$)]
$->$ $\diamond$[at l_{14} $\wedge$ type=$type_0$+'ptr'+contents $\wedge$ α]
by F_k, PR

CASE: amlp $\neq$ 'record' $\wedge$ amlp $\neq$ 'pointer'

This is the null case not shown on figure 9. In this case it is trivial to see that:

16. [R(0) $\wedge$ amlp $\neq$ 'record' $\wedge$ amlp $\neq$ 'pointer']
$->$ $\diamond$[at l_{14} $\wedge$ α]

from which we can deduce

17. R(0) $\wedge$ amlp $\neq$ 'record' $\wedge$ amlp $\neq$ 'pointer' $->$ ψ

which allows us to conclude

18. R(0) $->$ $\diamond\psi$ by 9..15, $\diamond$Q, PR

This technically does not complete this part of the proof because the path {a,b,c,r} through the graph has not been considered. Proof of this edge sequence will be subsumed into the proof of the case when $|T| > 0$ and vars is an empty list, below.

Proof of (b2):

19. R(m+1) $->$ [at l_0 $\wedge$ α $\wedge$ $0 \leq |T| \leq m+1$]
by PR

CASE: $|T|=0$

20. $[\text{at } l_0 \wedge |T|=0 \wedge \alpha] \rightarrow \Diamond\psi$
by 8..18

CASE: $|T| > 0$

21. $[\text{at } l_0 \wedge 0 \leq |T| \leq m+1 \wedge \alpha]$
$\rightarrow [\text{at } l_0 \wedge (\text{amlp}=\text{'record'} \vee \text{amlp}=\text{'pointer'}) \wedge 0 \leq |T| \leq m+1 \wedge \alpha]$
by def'n of T

There are now two cases:

CASE: amlp='pointer' (Considering this case first simplifies the presentation of the proof)

22. $[\text{at } l_0 \wedge \text{amlp}=\text{'pointer'} \wedge 0 \leq |T| \leq m+1 \wedge \alpha]$
$\rightarrow \Diamond[\text{at } l_9 \wedge 0 \leq |T| \leq m+1 \wedge \text{amlp}=\text{'pointer'}$
$\wedge \text{type}=\text{type}_0+\text{'ptr'}$
$\wedge \text{TYPE}(\text{amlp}_c)=\text{type}_0+\text{TYPE}(\text{amlp}_0)]$
by F_j,
def'n of TYPE

There are now two subcases

SUBCASE: The identifier is visible in the name table recnames

23. $[\text{at } l_9 \wedge 0 \leq |T| \leq m+1 \wedge \text{amlp}=\text{'pointer'}$
$\wedge \text{type}=\text{type}_0+\text{'ptr'} \wedge \text{visible(recnames)}$
$\wedge \text{TYPE}(\text{amlp}_c)=\text{type}_0+\text{TYPE}(\text{amlp}_0)]$
$\rightarrow \Diamond[\text{at } l_{14} \wedge \alpha$
$\wedge \text{type}=\text{type}_0+\text{'ptr'}+\text{contents}]$
by F_k, PR,
def'n of TYPE

24. $[\text{at } l_{14} \wedge \alpha] \rightarrow \psi$ by PR

SUBCASE: The identifier is not visible in the name table recnames

25. [at l_9 ∧ $0 \leq |T| \leq m+1$ ∧ amlp='pointer'
∧ ~(visible(recnames)) ∧ type=$type_0$+'ptr'
∧ TYPE($amlp_c$)=$type_0$+TYPE($amlp_0$)]
-> ◇[at l_{10} ∧ $0 \leq |T| \leq m+1$ ∧ amlp='pointer'
∧ visible(recnames) ∧ type=$type_0$+'ptr'
∧ TYPE($amlp_c$)=$type_0$+TYPE($amlp_0$)]
by F_l,
def'n of insert

There are now two sub-subcases:

SUB-SUBCASE: The identifier is visible in the name table types

26. [at l_{10} ∧ visible(types) ∧ $0 \leq |T| \leq m+1$
∧ type=$type_0$+'ptr' ∧ amlp='pointer'
∧ TYPE($amlp_c$)=$type_0$+TYPE($amlp_0$)]
-> [at l_{10} ∧ amlp='types' ∧ $0 \leq |T| \leq m+1$ ∧ α
∧ type=$type_0$+'ptr']
by def'n of visible,
def'n of TYPE,
IP(types)

27. [at l_{10} ∧ $0 \leq |T| \leq m+1$ ∧ amlp='types'
∧ type=$type_0$+'ptr' ∧ α]
-> ◇[at l_{11} ∧ $0 \leq |T| \leq m+1$ ∧ amlp='type'
∧ type=$type_0$+'ptr' ∧ α]
by F_m,
def'n of son

28. [at l_{11} ∧ $0 \leq |T| \leq m+1$ ∧ amlp='types'
∧ type=$type_0$+'ptr' ∧ α]
-> ◇[at l_{12} ∧ (amlp='pointer' ∨ amlp='record')
∧ α ∧ $0 \leq |T| \leq m$] by F_n,
PR,
def'n of T

29. $[\text{at } l_{11} \wedge 0 \leq |T| \leq m \wedge \alpha$
$\wedge$ (amlp='pointer' $\vee$ amlp='record')]
$\rightarrow \Diamond[\text{at } l_0 \wedge 0 \leq |T| \leq m \wedge \alpha$
$\wedge$ (amlp='record' $\vee$ amlp='pointer')]
by F_o

30. $[\text{at } l_0 \wedge 0 \leq |T| \leq m \wedge \alpha$
$\wedge$ (amlp='record' $\vee$ amlp='pointer')]
$\rightarrow$ R(m) by PR

SUB–SUBCASE: The identifier is not visible in the name table types

31. $[\text{at } l_{10} \wedge 0 \leq |T| \leq m+1 \wedge$ amlp='pointer'
$\wedge$ type=$type_0$+'ptr'
$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE($amlp_0$)]
$\rightarrow \Diamond[\text{at } l_{13} \wedge$ type=$type_0$+'ptr']
by F_p,
def'n of error,
PR

32. $[\text{at } l_{13} \wedge$ type=$type_0$+'ptr']
$\rightarrow \Diamond[\text{at } l_{14} \wedge$ type=std–error $\wedge \alpha]$
by F_q,
def'n of TYPE

33. R(m+1) $\wedge$ amlp='pointer' $\rightarrow \Diamond$R(m) $\vee \Diamond\psi$
by 22,24,30,32,PR,$\Diamond$Q

CASE: amlp='record'

34. $[\text{at } l_0 \wedge 0 \leq |T| \leq m+1 \wedge \alpha \wedge$ amlp='record']
$\rightarrow \Diamond[\text{at } l_1 \wedge 0 \leq |T| \leq m+1 \wedge$ amlp='record'
$\wedge$ type=$type_0$+'record'
$\wedge$ TYPE($amlp_c$)=$type_0$+TYPE(amlp)]
by F_a

35. [at $l_1 \wedge 0 \leq |T| \leq m+1 \wedge$ amlp='record'
$\wedge \alpha \wedge$ type=$type_0$+'record']
-> $\Diamond$[at $l_2 \wedge 0 \leq |T| \leq m+1 \wedge$ amlp='fields'
$\wedge$ scope(amlp)=newscope(record) $\wedge \alpha$]
by F_b,
def'n of son,
fields $\in$ SC,
def'n of TYPE

36. [at $l_2 \wedge 0 \leq |T| \leq m+1 \wedge$ amlp='fields' $\wedge \alpha$]
-> $\Diamond$[at $l_3 \wedge 0 \leq |T| \leq m+1 \wedge$ amlp='fields'
$\wedge \alpha \wedge$ vars'=vars $\wedge |vars'| \geq 0$]
by F_c,
PR,
def'n of front

At this point the loop in the evaluation algorithm for records must be considered. (Edges d..i in the graph in figure 9). This loop is proven using another application of the IND rule and the assertion L(l).

37. [at $l_3 \wedge 0 \leq |T| \leq m+1 \wedge$ amlp='fields' $\wedge \alpha$
$\wedge 0 \leq |vars'|$]
-> [at $l_3 \wedge 0 \leq |T| \leq m+1 \wedge$ amlp='fields' $\wedge \alpha$
$\wedge 0 \leq |vars'| \leq |vars'|$]
by domain, PR

38. [at $l_3 \wedge 0 \leq |vars'| \leq |vars'| \wedge 0 \leq |T| \leq m+1$
$\wedge$ amlp='fields' $\wedge \alpha$]
-> $\exists$l.[at $l_3 \wedge 0 \leq |vars'| \leq l \wedge 0 \leq |T| \leq m+1$
$\wedge$ amlp='fields' $\wedge \alpha$]
by T24 of [Ma82]

39. [at $l_3 \wedge 0 \leq |vars'| \leq l \wedge 0 \leq |T| \leq m+1$
$\wedge$ amlp='fields' $\wedge \alpha$]
-> L(l) $\wedge 0 \leq |T| \leq m+1$ by PR

Termination of the loop may now be shown in two parts. As was pointed out earlier, this

proof is significantly more complex than those of the previous examples. This complication will become clear in the next few lines of the proof. A brief discussion of the remainer of the proof of the loop is therefore in order.

Loop termination is shown using the IND rule. In the case that l=0, the termination is obvious. In the case that l>0, the loop body is entered. edges d,e and f of figure 9 are traversed as usual, before calling the procedure again recursively. It is at this point that the logic of the proof becomes more advanced. Before the call we demonstrate that $|T|$ shrinks by one, thus making the upper bound m as opposed to m+1. The call returns us to l_0. By definition of IND therefore, the procedure terminates. We can then go a step further than was necessary in the previous proofs and consider the possible cases of return from a recursive call of the procedure. There are two possible cases, namely a return to l_{14} as a result of the recursive call on edge o, and a return to l_7, as a result of the recursive call on edge g. The proof of the procedure and the loop can then simultaneously be completed.

The proof once again has two parts:

(c1) $L(0) \wedge 0 \leq |T| \leq m+1 \rightarrow \Diamond\gamma$

(c2) $L(n+1) \wedge 0 \leq |T| \leq m+1 \rightarrow \Diamond\gamma \vee \Diamond L(n)$

$L(l) \rightarrow \Diamond\gamma$ by the IND rule.

Proof of (c1):

40. $[L(0) \wedge 0 \leq |T| \leq m+1]$
$\rightarrow [at\ l_3 \wedge |vars'|=0 \wedge 0 \leq |T| \leq m+1$
$\wedge$ amlp='fields' $\wedge\ \alpha]$
by PR

41. $|vars'|=0 \rightarrow eol(vars)$ by def'n of eol

42. $[at\ l_3 \wedge |vars'|=0 \wedge 0 \leq |T| \leq m+1$
$\wedge$ amlp='fields' $\wedge\ \alpha]$
$\rightarrow [at\ l_3 \wedge eol(vars) \wedge 0 \leq |T| \leq m+1$
$\wedge$ ampl='fields' $\wedge\ \alpha]$ by PR

43. [at l_3 ∧ eol(vars) ∧ $0 \leq |T| \leq m+1$
∧ amlp='fields' ∧ α]
-> $\diamond$[at l_{14} ∧ α] by F_r, PR

44. L(0) ∧ $0 \leq |T| \leq m+1$ -> $\diamond\gamma$ by $\diamond Q$

Proof of (c2)

45. [L(n+1) ∧ $0 \leq |T| \leq m+1$]
-> [at l_3 ∧ $0 \leq |vars'| \leq n+1$ ∧ $0 \leq |T| \leq m+1$
∧ amlp='fields' ∧ α] by PR

CASE: |vars'|=0

46. [at l_3 ∧ |vars'|=0 ∧ $0 \leq |T| \leq m+1$
∧ amlp='fields' ∧ α]
-> $\diamond\gamma$ by 40..44

CASE: |vars'|>0

47. |vars'|>0 -> ~eol(vars)

48. [at l_3 ∧ $0 \leq |vars'| \leq n+1$ ∧ |vars'|>0
∧ $0 \leq |T| \leq m+1$ ∧ amlp='fields' ∧ α]
-> [at l_3 ∧ $0 \leq |vars'| \leq n+1$ ∧ $0 \leq |T| \leq m+1$
∧ ~eol(vars) ∧ amlp='fields' ∧ α]
by PR

49. [at l_3 ∧ $0 \leq |vars'| \leq n+1$ ∧ $0 \leq |T| \leq m+1$
∧ ~eol(vars) ∧ amlp='fields' ∧ α
∧ |P|=u1]
-> $\diamond$[at l_4 ∧ $0 \leq |vars'| \leq n+1$ ∧ $0 \leq |T| \leq m+1$
∧ amlp='fields' ∧ α ∧ |P|=u1+1
∧ top(P) = 'fields']
by F_d,

def'n of saveposition

50. [at l_4 $\wedge$ $0 \leq |vars'| \leq n+1$ $\wedge$ $0 \leq |T| \leq m+1$
$\wedge$ top(P)='fields' $\wedge$ amlp='fields' $\wedge$ α]
-> $\diamond$[at l_5 $\wedge$ $0 \leq |T| \leq m+1$
$\wedge$ top(P)='fields' $\wedge$ amlp='type' $\wedge$ α]
by PR,
F_e,
def'n of son
Note that we can drop
the $0 \leq |vars'| \leq n+1$
at this time as it is
reset for us by the
restoreposition on
edge h.

51. [at l_5 $\wedge$ $0 \leq |T| \leq m+1$
$\wedge$ top(P)='fields' $\wedge$ amlp='type' $\wedge$ α]
-> $\diamond$[at l_6 $\wedge$ $0 \leq |T| \leq m$ $\wedge$ α
$\wedge$ top(P)='fields' $\wedge$ (amlp='record $\vee$ amlp='pointer)]
by F_f,
def'n of son,
def'n of T

52. [at l_6 $\wedge$ $0 \leq |T| \leq m$
$\wedge$ top(P)='fields' $\wedge$ α
($\wedge$ amlp='record' $\vee$ amlp='pointer')]
-> $\diamond$[at l_0 $\wedge$ $0 \leq |T| \leq m$ $\wedge$ α]
by F_g, PR

53. [at l_0 $\wedge$ $0 \leq |T| \leq m$ $\wedge$ α] -> R(m)
by PR

54. R(m+1) -> $\diamond\psi$ $\vee$ $\diamond$R(m) 18,20,33,53,$\diamond$Q,PR

which completes the proof of (b) and allows us to conclude:

55. ϕ -> $\Diamond\psi$ by IND

The proof of (c) can be completed by considering the possible places where the procedure can return to when the recursion terminates (which we have just shown it must do). The simplest possibility is that the return is at the sink vertex of edge o, ie l_{14}. This is a simple tail recursion and needs no further exposition here. The second possibility is that the return is at the sink vertex of edge g, ie l_7. We consider the possibility that arises if the recursion returns to this edge.

56. $\Diamond$[at l_7 $\wedge$ top(P)='fields']
by def'n of recursive
procedure call.
Note that we can make no
assumption about |vars'|
until the restoreposition
has restored the state to
that which held before the
recursive call.

57. [at l_7 $\wedge$ top(P)='fields' $\wedge$ scope(top(P))=u2]
-> $\Diamond$[at l_8 $\wedge$ α $\wedge$ $0 \leq |vars'| \leq n+1$
$\wedge$ amlp='fields' $\wedge$ scope(amlp)=u2]
by F_h,
def'n of TYPE,
def'n of restoreposition

58. [at l_8 $\wedge$ $0 \leq |vars'| \leq n+1$ $\wedge$ α $\wedge$ amlp='fields']
-> $\Diamond$[at l_3 $\wedge$ $0 \leq |vars'| \leq n$ $\wedge$ α $\wedge$ amlp='fields']
by F_i,
def'n of next,
PR

59. [at l_3 $\wedge$ $0 \leq |vars'| \leq n$ $\wedge$ α $\wedge$ amlp='fields']
-> L(n) by PR

60. L(l) -> $\diamond\gamma$ by IND

which completes the proof.

REMARKS

We have now seen that context specification and checking problems of real-life complexity can be specified in CFF/AML, and then shown to be formally correct, and to terminate. This goes further than earlier work on attribute grammars, where attribute propagations could be shown to terminate, but without any guarantee that correct information was propagated.

The above proofs, particularly the final one, are sometimes a little subtle and arcane. However, the exercise of proving a specification correct is most definitely a worthwhile one. The benefits of such a proof are that the specifications are then known to correct with respect to the verification assertions and that the language specifier gets to grips with the subtleties of the specification and can therefore remove any deep errors that may not become apparent if the specification were debugged in a fashion similar to that of debugging a program.

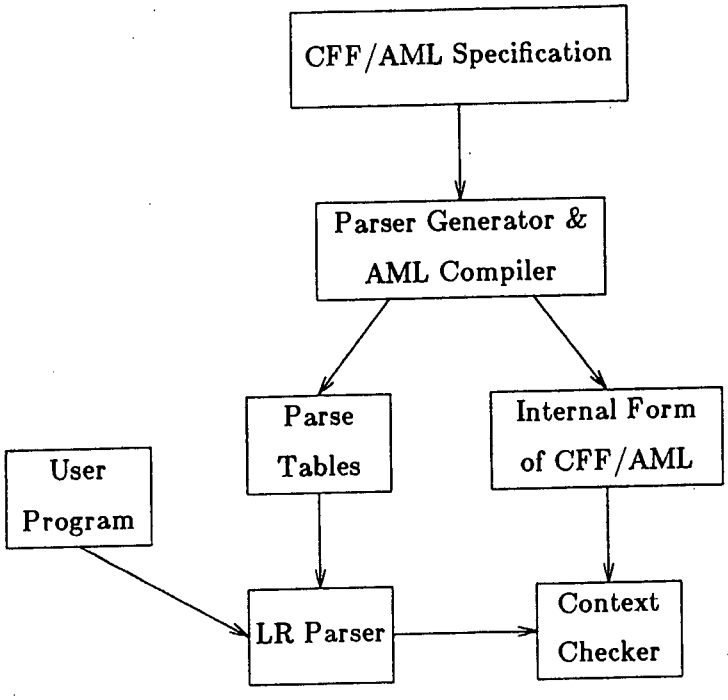
Another advantage of introducing specification verifications is that once the language specifier becomes used to verifying specifications and developing assertions, a more rigorous approach, namely that of evolving the specifications from the assertions, becomes possible, thus making the whole specification process less error prone.

PART IV - CFF/AML IMPLEMENTATION

This part of the thesis considers the current implementation of a compiler front-end driven by CFF/AML.

CFF/AML has been implemented in Pascal on a SPERRY 1100/81 at the University of Cape Town, and currently consists of a set of background programs to 'compile' a CFF/AML specification, an LR parser driven by CFF, and a tree walker/interpreter which performs context condition checking.

The system can be diagrammatically viewed as:



Conceptually, the system can be viewed as operating as follows:

- The language specifier builds a CFF/AML specification of the syntax and context conditions of a programming language.
- This is fed into a set of background programs which translate the CFF into LR parse tables, and both CFF and AML into an internal form for use in the context condition checker.
- A user program can then be fed into the language-independent LR parser, which will parse a program by reference to the LR parse tables supplied by the generator program, and produce as output a stripped parse tree representation of the program, where all tokens not needed for source code reconstruction are removed.

- The parse tree is then used by the context checker, which walks the tree and performs the checks specified by the AML to determine whether or not the program is legal with respect to context condition specification.

The various entities of this diagram are considered in more detail in the following two chapters.

Chapter 19: THE GENERATOR PROGRAMS

This chapter discusses the generator programs. There are two of these, the first being the CFF/AML compiler, and the second the parse table generator.

The CFF/AML specification is checked and changed to internal representations for ease of use. The CFF part of a specification is just 'Godel numbered' [Kfoury et al 82], and the AML is translated to a form of P–code developed especially for this purpose.

When the translation process is complete then the internal forms of the specification are written to text files for later consumption by other programs.

The parse table generator then picks up the internal form of the CFF, modifies this to a more rigid formalism, and then uses a standard LR parse table generation algorithm to create the parse tables [Barrett & Couch 79].

Each of these procedures is now examined in more detail. It is simpler to perform this descriptive process 'bottom up', describing the AML compiler first, although this is called from the CFF translator.

THE AML COMPILER

The CFF translator translates the CFF rules into internal form, as will be seen below, until a '[' (start AML decoration metasymbol) is encountered. Control then passes to the AML compiler, which compiles the decoration and then returns control to the CFF translator.

AML is an LL(1) language, so the AML compiler is a standard recursive descent compiler, except that because each AML decoration is compiled as encountered, the set of start symbols for the compilation is the set of start symbols for statements.

The AML source code is translated into a form of P–code especially devised for this purpose. This P–code is a stack machine code.

It has the usual set of instructions for handling the AML integer type and control statements.

It has a special set of instructions for handling AML string variable manipulation, namely CONCAT to concatenate two strings, and EQUATES, which compares two strings and returns an indication of whether or not they are equal.

The special commands introduced into AML for scope, stack and tree walk control, as well as the additional 'conventional' statements, are all embedded in the AML interpreter. Their corresponding P–codes can therefore also be high–level. Thus, for example

exit, error, insert, son, etc are all atomic AML instructions, and are mapped directly onto a P–code without further breakdown (except for some additional instructions for parameter passing where applicable).

The AML compiler holds all the AML translated code in a code table. Each decoration ends with an AML return command, which indicates to the AML interpreter that execution of the context checks is finished and control should pass back to the tree walker. (See the chapter on the Tree Walker/AML interpreter below).

In order that an AML decoration can be both distinguished from the rest of a CFF rule internal form, and located when its execution is desired, an AML decoration is indicated in a CFF rule internal form by holding in the rule, in place of the decoration, the complement of the location in the code table where the start of the P–code version of the AML decoration is found.

As well as a code table, the AML compiler constructs a map of the simple AML variables, which is interpreted as a simple variables main memory when the AML interpreter runs. Constant strings are also held in this simple variable memory.

The AML name tables are by nature dynamic structures. A reference to an AML name table can therefore be checked only to ensure that such a table is declared in some scope; no assumptions regarding its type can be made. Thus, if a FRONT(x) command is encountered, the compiler can check that a table x is declared; it cannot check that x is indeed a list type.

Each AML name table is assigned an internal number and each AML simple variable is also assigned an internal number, namely the 'location' in the 'simple variable memory'. As the command sets using AML simple variables and AML name tables are disjoint, the numbers of name tables and 'locations' in the 'simple variable memory' can overlap.

When the parse of the AML decoration is complete, control is passed back to the CFF translator.

TRANSLATING CFF TO INTERNAL FORM

The CFF translator translates the CFF part of a CFF/AML specification into an internal form.

The CFF metasymbols and standard CFF rules (integer, number, identifier and string) are mapped onto predefined constants.

If the highest number assigned to the standard CFF rules is m, then the reserved words and special tokens of the language being defined are mapped onto consecutive integers such that the first is mapped onto the first integer greater than m. Recall that these reserved words and special tokens are easily identifiable in a CFF specification; the reserved words are in uppercase and the tokens are any special characters not in the metacharacter set, or any quoted characters.

If the last reserved word or special token has number n, then the rules are numbered from the first integer multiple of 10 greater than n. There is no upper limit on the number of reserved words and special tokens.

When looking at a CFF rule it is then possible to determine the type of any component of the rule from its internal number.

When an AML decoration is encountered, then the AML compiler is called. This returns the complement of the location in the code table of the first P–code instruction corresponding to this decoration. This number is then inserted in the internal form of the rule, and the translation process continues.

AN EXAMPLE

The CFF/AML rule

```
"types"
TYPE <identifier[ if unique(types) then
                    insert(types)
                  else
                    error('Redeclared typeid')
                ] = type_;> %
```

is a degenerate form of the types rule in Pascal (we only check that the identifier is not in the type table, as opposed to the full check against all name tables). The internal form of this rule would be:

```
118 types
58 4 22 -317 31 119 5 28 6 2
```

118 is the internal number of the CFF rule types (the name is included for human debugging information, and to make error messages in the parser and tree walker/interpreter more comprehensible).

58 is the internal number of the reserved word TYPE.

4 4 is the internal number of the start–of–list metasymbol <.

22 is the internal number of the standard CFE rule identifier.

–317 The absolute value of this number is the location in the code table where the P–code representation of the AML decoration is to be found.

31 is the internal number of the = sign.

119 is the internal number of the rule type.

5 is the internal number of the list exit metasymbol _.

28 is the internal number of the ;.

6 is the internal number of the end–of–list metasymbol >.

2 is the internal number of the end–of–rule metasymbol %.

The code table might contain (using a mnemonic representation of the opcodes) the codes

LOCATION	OPCODE	OPERAND	comment
317	FPUSH	1	push name table with internal number 1 (types) onto stack
318	PARMS	1	the unique instruction can have multiple parameters, so indicate that in this case we have one parameter
319	SCALL	6	call the standard AML function unique to determine whether or not the identifier is in types
320	JEQ	323	unique returns 0 to indicate false
321	LFINS	1	insert using the leaf (identifier) as key, into table 1 (types)
322	JMP	324	skip else part
323	ERROR	5	print error message. Recall that error messages are held in 'main memory for simple AML variables. The error message is in location 5
324	AMLEND		end of AML decoration

represented in the table. Appropriate numbers have been chosen to represent rules, tables, etc.

MAKING INTERNAL FORMS AVAILABLE TO UTILITIES

The internal forms must be made available to the utilities in the system, such as the parser or context condition checker. The various internal forms are therefore written out to a text file, constructed as follows.

- The number of metasymbols, followed by the metasymbols
- The number of reserved words/special tokens, followed by the reserved words and special tokens.
- The number of CFF rules, followed by the rules in the form presented above. The number of symbols in each rule is also indicated.
- The size of the code table, followed by the code table.
- The size of the 'simple variables main memory', followed by, for each location in this memory, the name of a variable (or blank if a literal), the type (integer, flag, string), and the initial value of the location (mainly for literals).
- The number of name tables declared, followed by the internal number and name of each table. This information is to enable user–friendly error messages to be generated when bugs in the AML code cause the interpreter to crash.

THE PARSE TABLES GENERATOR

This program reads in the CFF rules internal form, and manipulates this to get a grammar amenable to being used in a standard LR parse tables generation algorithm.

The internal form file is read. The AML information is stripped from the rule descriptions. The rules are then transformed according to the following rules:

A new goal rule is introduced, just in case the language specifier has inadvertently used the goal rule (the first rule) of the CFF grammar recursively.

Any list rule of the form

"rule"
a <b_c> d %

is translated into a new set of rules

"rule"
a listrule d %

"listrule"
b c listrule

¦b %

Any rule with an option, such as

"rule"
a ?b c %

is translated into a rule of the form

"rule"
a b c
¦a c %

Any new rules introduced during the translation process are given unique names and placed in the internal forms list of the CFF rules, and the CFF rule internal forms modified to reflect the changes.

When the transformations are complete, then a standard parse table generation algorithm [Barrett & Couch 79] is applied, and the parse tables written away in text file form for future use.

Chapter 20 : THE UTILITY PROGRAMS

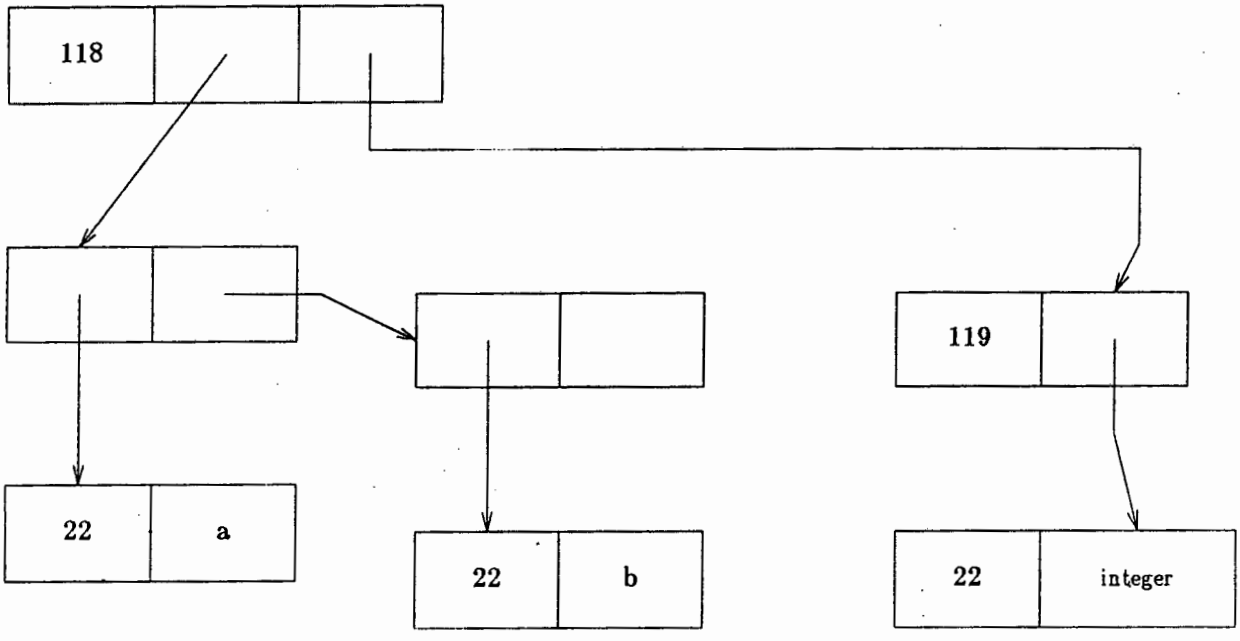
This chapter discusses the utility programs currently extant in the system. There are two of these, namely an LR parser and a context condition checker.

The LR parser is an uninteresting utility. It takes the LR parse tables generated by the LR parse table generator, and a user program, and generates a parse tree representation of the program, except that wherever possible, reserved words and tokens are omitted unless needed for the process of source code reconstruction. All reserved words not held can be extracted from the internal form of the CFF rules.

A sample tree (from the CFF rule introduced in the previous chapter) for the syntax:

TYPE a,b : integer ;

would have the form:



The root of this subtree contains a 118 (the name of the rule from which it was constructed), and two pointers, the lefthand one to the list of identifiers, and the righthand one to a node constructed from the rule type (rule 119).

This lefthand pointer points to a node constructed from the listrule rule, but which is made anonymous as listrule is not known to the CFF grammar. Conceptually flat lists in CFF therefore become deep in the parse tree, and a good deal of effort had to be expended on the processes for mapping deep lists onto CFF lists in the context checker.

The anonymous listrule node in turn points to a leaf (the identifier a), and another listrule, which points only to the leaf holding the identifier b, as there are no more iterations

of the list.

Returning now to the types rule node (118), the second pointer points to a node constructed from rule 119 (types), which in turn points to the leaf holding the identifier integer.

Note that the : is not held, as it can be extracted from rule 118 (types).

THE TREE WALKER

The context checker has buried in it a tree walker, which walks the parse tree in pre-order. Simultaneous to walking the parse tree, the CFF rules from which the nodes are constructed are walked as well. In fact the walk is driven from the rules, not the tree shape, which enables us to interpret an open list metasymbol as 'go down a level in the tree', anticipate optional elements of a rule which might not be present, and so forth.

Each time an AML decoration in encountered in the rule, the tree walker halts and calls the AML interpreter, passing it the start address of the AML P-code to be executed. When the AML interpreter halts, the tree walker then continues.

THE AML INTERPRETER

The AML interpreter is a stack machine driven by the AML P-codes. It operates on a number of data structures, namely:

- The parse tree version of a program.
- A scoping framework structure which forms a skeleton for the name tables.
- The name tables.
- The 'AML stack', ie the stack affected by the AML push, pop, acomp, mcomp and dcomp commands.
- An internal stack, used for expression evaluation and parameter passing.
- A return address stack, used for procedure return addresses.
- A positional stack, used for the saveposition and restoreposition commands.

All of these except the scoping framework and name tables are the obvious structures; the scoping framework and name tables are considered below.

A new scope is created by the NEWSCOPE command, which triggers the creation of a new scope header block and links this block into the scope structure. A scope header block can be diagrammatically represented as in figure 1.

Lexical Level Number
^ Father SHB
^ Point on AST pointed to by Tree Walk Pointers when NEWSCOPE was executed
^ list of set table headers
^ list of stack table headers
^ list of list table headers
^ local assignment compatibility table
^ local dyadic compatibility table
^ local monadic compatibility table

Figure 1

All the scope headers are linked together using the pointer to the FATHER SHB to build a tree structure. Once a scope header block has been identified then all the context information for that scope can be extracted from the structures rooted in the scope header block.

The tree walk pointers and AML pointers each have structure as represented in figure 2.

^ current internal form of CFF rule
^ current node on parse tree
^ current scope header block

Figure 2

It is therefore always possible to identify the current scope header block from the information carried in the pointers.

A list of nodes on which scope changes, similar to the set SC constructed in the formal definition of the language in part III, together with the addresses of the scope header blocks corresponding to each scope change, must be maintained for handling scope

changes. When a SON onto a node in the list succeeds, or the tree walk pointers reach such a node during the navigation process, then the pointer to the current scope header block must be altered to reflect the change in scope. A similar strategy must be adopted for identifying when to change scope for FATHER commands.

An invisible scope level is NOT discarded. There are two reasons for this:

- o Although the editor does not yet exist, we must anticipate the context checker being used from the editor, and in an editor the user may jump around the tree in an arbitrary fashion.
- o The information in the scope may be needed for type evaluations, for example in languages which have modules, it must be possible to get back to the declarations contained in the EXPORT list of a module.

The set name table header structure can be represented as in Figure 3.

the type of the table - SET
the name of the table
^ scope header block in which name table is rooted.
^ set table element structure
^ next set table header

Figure 3

The set name table element structure can be chosen to maximise access time efficiency, such as a binary tree or hash table. Each element of the structure has the form indicated in figure 4.

key part – the string which was inserted into the table
^ place on parse tree from which the insert was made
CFF rule on which the insert was executed
housekeeping pointer information, eg left and right links in a binary tree

Figure 4

The stack table header is identical to that for set tables, except that a different type indication is held. The element structure must be arranged in LIFO order. This could be a linear stack, or a tree or hash table for fast lookup and a thread running through the structure giving order for stack operations.

The list element structure can be arranged as for a stack, except that the order is FIFO. The header block is different and can be represented as in figure 5.

the type of the table – LIST
the name of the table
^ scope header block in which name table is rooted.
^ list table element structure
^ next list table header
^ next element in the list

Figure 5

The new field is a pointer to the next element in the list, and is used in the FRONT, NEXT and EOL operations. Front sets this field to the front of the list, next modifies it to point to the next entry, and EOL returns true or false depending on its value.

Implicit in the above diagrams is the understanding that the various table type header blocks are searched sequentially by type to identify if a name table exists at a particular scope. This means that identifying if a particular table exists at a particular scope will take on average n/2 different comparisons, as the lists of header block are not necessarily ordered. This problem can be overcome by maintaining for each scope an additional structure, built as a hash table or tree for efficiency, each element of which looks like figure 6.

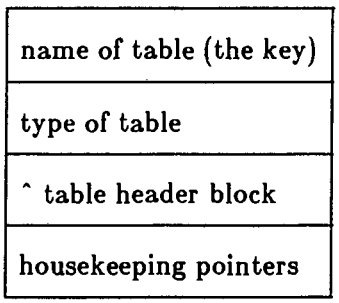


Figure 6

Alternately, this structure can be rooted in the scope header block in place of the three pointers to table lists currently there.

Commands such as visible use the pointer to the current scope header block in the AML pointer or tree walk pointer (depending on the switch) to identify the local scope, then search this scope for an element with a matching key. If no match is found then the pointer to the father scope header block in the current scope header block is used to locate the next scope and the process is repeated, until a match is found or we run out of scopes.

AN EXAMPLE

Let us consider the execution of the AML decoration on the types rule which was introduced above. The tree walker walks the tree in preorder, and walks through the definition of the CFF rule for types simultaneously. When the AML decoration is encountered, then the start address of the P–code is passed to the AML interpreter, which then executes the P–code until an AMLEND instruction is encountered, which causes the interpreter to halt and pass control back to the tree walker.

LOCATION	OPCODE	OPERAND	comment
317	FPUSH	1	push name table with internal number 1 (types) onto stack
318	PARMS	1	the unique instruction can have multiple parameters, so indicate that in this case we have one parameter
319	SCALL	6	call the standard AML function unique to determine whether or not the identifier is in types
320	JEQ	323	unique returns 0 to indicate false
321	LFINS	1	insert using the leaf (identifier) as key, into table 1 (types)
322	JMP	324	skip else part
323	ERROR	5	print error message. Recall that error messages are held in 'main memory for simple AML variables. The error message is in location 5
324	AMLEND		end of AML decoration

Figure 7

The P-code for the decoration is contained in figure 7. The execution of the AML P-code

starts at location 317.

317 Push a 1 onto the implicit stack. This is the internal number of the types name table.

318 Indicate that there is only one parameter. This is used by the standard functions which can have multiple parameters to discover how many elements on the implicit stack are to be interpreted as the internal numbers of name table arguments.

319 Call standard function 6 (unique with an implicit string argument – the contents of the leaf rule identifier). It will remove the top n elements of the implicit stack, where n corresponds to the number given in the parms P–code, use the tree walk pointers' pointer to the local scope header block to find the local scope header block, and search the list of name tables to find a table called types. If there is no local table types, then the father pointer to the next scope block is used and the lookup is tried again, until either an entry is found in the most local occurrence of the table types, or we run out of scopes, in which case unique returns true. An indication of the result of this search is placed on the implicit stack and the function terminates.

320 If a 0 is returned, then the identifier is not unique. Jump to the else part of the AML IF statement.

321 Insert using the leaf as a key. An entry record for the identifier is constructed, using the pointers or the top of the MARK stack if $|MARK|>0$ to indicate where to point to on the parse tree. This entry is then linked into the name table entries structure for the table types.

322 Jump over the else part.

323 Generate an error, using the string in location 5 of the simple variables memory as the error message.

324 The AMLEND causes the AML interpreter to stop execution and pass control back to the tree walker.

SOME REMARKS ON THE IMPLEMENTATION

The system described above has been implemented with a primary view to demonstrating that CFF/AML is successfully implementable; this goal has been attained.

Efficiency was a secondary consideration. Lists, stacks and set name tables were implemented using a linear structure, which impacts negatively on run time; techniques to speed up table lookups are well known and could easily be implemented [Aho et al 83].

The SPERRY computer on which the implementation was undertaken introduces three major limitations:

- o No program may exceed 250K in size.
- o All programs larger than 20K must be run in batch mode.
- o All I/O is batch oriented, ie syntax directed editors cannot be written simply because it is impossible to perform successful screen control.

These restrictions meant that the parser and context checker had to be split into two programs. Further, all large data structures, such as the GOTO and ACTION tables in the parse table construction and the representation of the parse tree have to be held in secondary memory using FORTRAN relative access files. These are extremely slow, so that speeding up the running of the interpreter would have a negligible effect on total runtime in this implementation.

Some figures for timings are contained in figure 8.

	Processor Time	I/O Time
CFF/AML COMPILER	3.183 secs	4.222 secs
LR PARSER	1.943 secs	15.327 secs
CONTEXT CHECKER	1.238 secs	12.732 secs

Figure 8

The time for the generator program is for the specification of Pascal. The sample program parsed by the LR parser and walked in the context checker had 85 nodes in the parse tree and required execution of 874 AML P-codes. The I/O characteristic of the context checker and parser reflect the inherent slowness of FORTRAN I/O on the SPERRY. If only processor time is considered, the context checker is quite fast without any data structure optimization. These times are expected to improve with the next implementation. LR parse table construction times are not given here; these are known to be large, and for Pascal the construction of the parse tables takes of the order of 20 minutes of processor time and 1.75 hours of I/O time.

The AML interpreter itself is an extremely simple machine, as it executes only the conventional P–codes, such as jumps or pushes and pops on the implicit stack, and arithmetic directly. All other P–codes are interpreted as calls to procedures buried inside the context checker, such as visible, newscope, insert, etc. This means that most of the operations of AML are run as native code and are not interpreted, which speeds up the context checker.

PART V – CONCLUSIONS

This part contains some concluding remarks concerning the thesis. A discussion of related work, so that this project can be placed into the correct perspective and discussion of the contributions of this thesis are also included, as are some possible directions for future research.

Chapter 21: RELATED WORK

Related work should really be discussed in sections corresponding approximately to the parts of this thesis. CFF/AML is therefore considered against the various context condition specification methodologies which were available before the thesis work began, and work which has been proceeding contemporaneously with this project. Then our work on verification of recursive programs is compared with other known attempts, and finally proving specifications is compared to other work in this field.

The question of automating the generation of context condition checks of a programming language in an automatically generated environment has been receiving a significant degree of research interest in recent years, either in compilers or in editors. A number of techniques have been used for the specification of these conditions.

The first could be called the 'ad hoc' method, where no attempt has been made to specify the context conditions at all; instead, provision is made for 'semantic mechanisms' to be linked in to the code generated by the compiler generator. These mechanisms must be written in the target language of the generator. Projects in this vein include the venerable YACC [Johnson 75], which makes only a bow at context conditions, and the S/SL project [Holt et al 82]. This is a more ambitious effort, where the syntax is described as a sort of high–level automaton, and semantic mechanisms may be called when needed. As CFF/AML is entirely self contained (no new code in conventional programming languages need be written when generating a new environment) there is no real analogy between these methods and the approach of this thesis.

The second method is to use a formalism of the W–grammar variety, for example the CDL compiler compiler [Koster 76a, Koster 76b]. This approach is not favoured, as the exact specification of syntax which CFF gives is preferred over the 'generative' syntax of a W–grammar.

The third method (and the one most beloved of those who have taken the formalism approach) is to use some variant of attribute grammars [Bochman & Ward 78, Farrow 82, Kastens et al 82, Kennedy & Warren 76, Raiha et al 78]. One of the most recent compiler compiler in this vein is the GAG system [Kastens et al 82], which uses ordered attribute grammars [Kastens 80] as the specification formalism. However, the GAG system uses the grammars merely to give attribute flow, and relies on functions written in an external language to perform the processing. Attribute grammars have also been used as a specification formalism for editors, such as POE [Johnson 83] or the synthesizer generator [Reps 84, Reps et al 83, Reps & Teitelbaum 84]. POE introduces graph–like attribute

flow where the attributes flow to the required point directly as opposed to by explicit synthesis or inheritance, which is closer to our point of view.

Our reasons for not using attribute grammars were made clear in the introduction of this thesis. Primarily they are unsuitable because, despite their theoretical advantages, and with no slight intended to the excellent and ingenious work devoted to making them implementable, they remain an extremely unpleasant formalism to use. One of the objectives of this thesis was to simplify the task of the language designer and specifier by creating a formalism which would be easy to use and therefore simplify the design and specification tasks. Nevertheless, CFF/AML owes a great debt to attribute grammars, especially their concept of decorating the syntax of a programming language with the context conditions associated with that syntax.

Another approach, which has been adopted by the Mentor Group [Donzeau–Gouge et al 80, Kahn et al 83] and the GANDALF project [Ambriola et al 84, Habermann & Notkin 82, Kaiser 84a, Kaiser 84b, Kaiser & Feiler 84, Medina–Mora et al 83], is to have a special tree manipulation language in which the context conditions are specified. This is closer to the AML approach. However, the languages tend to be of a far lower level than AML, being more oriented towards tree manipulation in editors than context condition checking.

The denotational approach has been investigated by a number of researchers contemporaneously with this thesis [Rosselet 84, Pleban 84]. There is no analogy between this work and the CFF/AML approach.

Context condition checking by inference rules has been investigated by Despeyroux [Despeyroux 84]. The inference rules are mapped onto Prolog and then interpreted. Once again, there is no analogy between this approach and CFF/AML, although the increased abstractness and conciseness of this and the denotational approach make them very attractive.

Finally CFF/AML should be contrasted with operational definitions of programming languages. Conceptually there is no doubt that CFF/AML is a hybrid of the attribute and operational concepts, with programming language type code (the operational influenced part) augmenting context free syntax (the attribute influenced part). However, AML, with its powerful and flexible built in name table facility and high–level commands is not an operational definition in the Vienna Definition Language [Wegner 72] style at all. The VDL style is significantly more low level than that of AML.

We turn now to the problem of verifying recursive programs. Despite the strong theoretic relationship between recursion and iteration, this does not appear to have been utilised in proving recursive programs correct. A search of the literature turned up Hoare's paper on an axiomatic approach to procedures and functions [Hoare 71], in line with his earlier work on verification, and a number of theses and papers on verification systems, all of which follow a Hoare–style approach or a fixpoint theory approach to recursion verification [Ariely 75, Igarashi et al 75, Suzuki 76, Vullemin 74]. Manna and Pnueli's work has not dealt with this issue at all, being more interested in verification of concurrent systems, so we are therefore forced to conclude that this work has no analogy other than the work on verification systems which appears to have tailed off by the mid 1970's. The approach to proving calls, ie the PCALL and FCALL rules, has been adopted in a different formalism by Gries [Gries 81], particularly a similar notation for parameter handling.

No reference in the literature to using temporal logic as the specification formalism of the semantics of a programming language could be found.

The work on verifying context conditions of programming languages appeared to be an entirely new field when this project began. The nearest approach to ours is taken by Rosselet [Rosselet 84], who uses the denotational definition to prove that the implementation of a specification is correct.

Chapter 22: THE CONTRIBUTIONS OF THIS THESIS

This thesis has made contributions to Computer Science in the following fields:

- In the field of language specification. CFF/AML is a metalanguage for the specification of compiler front ends. We believe that writing CFF/AML specifications requires significantly less effort on the part of a language specifier than other specification methodologies.
- In the field of programming language design. This is an indirect contribution, as CFF/AML simplifies the task of the designer. In general, it is possible to build the CFF/AML specification of a programming language as the language is designed. If the AML specification should become overly unwieldy, this should indicate to the language designer that the context conditions are getting out of control, and that a redesign is in order.
- In the field of theory of computation. The approach which presented here to the verification of recursive programs is entirely original, and should greatly simplify verification of subprograms. This should make a relatively arcane field far more accessible, especially to the average programmer who might wish to take advantage of verification techniques.
- In the field of formal specification of context conditions. The ability to verify that CFF/AML specifications satisfy some associated assertions allows the language specifier far greater confidence in his specification. This is an improvement on previous specification techniques. Even the theory of attribute grammars allows one to infer only about the attribute propagation; there is no guarantee that the attribute grammar functions perform correctly. The verification of CFF/AML specifications allows the language specifier to infer far more rigorous conclusions about the correctness of his specification.
- In the field of formal specification of programming language semantics. To the best of our knowledge, this thesis represents the first attempt to use temporal logic to specify the semantics of a programming language.

Chapter 23: CONCLUDING REMARKS & FUTURE PROSPECTS

This thesis reports on the results of a project which was intended to investigate methodologies for specification of context conditions of programming languages.

We looked at how a language specifier would operate in translating from an informal language specification to a formal one, and have oriented CFF/AML to reflect our opinion of this approach, which we believe simplifies the task of the language specifier.

We wished to be able to reason formally about CFF/AML, and CFF/AML specifications. We also needed to be able to prove correct recursive specifications which are not amenable to the traditional fixpoint approach. Because of the close links between CFF syntax, syntax diagrams and graphs, and also because it could support verification of recursive subprograms, we wished to use a temporal proof approach, which would exploit these links.

In order to do this, the existing temporal proof systems had to be extended to cope with subprograms and recursion. As a result of this detour, a new approach to the verification of recursive programs, using the same approach that is used for loops, was discovered, thus strengthening the theoretical link between iteration and recursion. A major advantage of this approach is that, rather than needing to find the least fixpoint of a recursive program and use that for the verification, the verifier can prove correct a recursive program directly, using an approach based on that used for verifying iterative programs which shows convergence of recursive programs.

Having developed a temporal proof approach to verification of recursive subprograms, it was natural to use a temporal proof approach to the verification of AML specifications. This meant the construction of a TPS for AML, which in turn resulted in a formal semantics for AML. Temporal logic was the natural way to provide such a formal semantics because the TPS was also used verifications. Thus, any other specification methodology would have been redundant.

In verifying AML specifications, we have shown that it is possible to reason formally about AML specifications of programming languages. It is also possible to go further than other work in this field because specifications can be shown to be correct as well as that they terminate.

An implementation of a CFF/AML–driven front end compiler has been built. This consists of four programs. The first, the CFF/AML compiler, takes a CFF/AML specification and translates this into a number of internal forms, including an internal

form of CFF, a P-code like translation of AML, and various other auxiliary tables. The second is an LR parse tables generator which translates the internal form of CFF into LR parse tables. The third is an LR parser which takes as input a set of LR parse tables for a language and a program written in that language, and produces as output a parse tree representation of the program. The final program is a context condition checker, which uses the parse tree and the CFF/AML code internal forms to check that the program represented in the parse tree does not violate any context conditions.

As yet this implementation is a prototype only; however it has been used for languages up to the complexity of Pascal, and is currently in use as an assistant for programming language design work.

The appendices contain four CFF/AML specifications. Appendix I is a CFF/AML description in CFF/AML, and demonstrates that CFF/AML is sufficiently powerful to describe itself.

Appendix II is a CFF/AML description of Pascal, and demonstrates, along with the specification of CLU in appendix IV, CFF/AML specifications of full-size languages.

Appendix III is a CFF/AML description of Dijkstra's Language. A subset of this language was used for demonstration purposes in the body of the thesis. This appendix is a CFF/AML description of the language at a stage in its evolution before it had acquired arrays, and demonstrates that CFF/AML can handle the convoluted scoping rules of that language.

Several avenues of possible future research radiate out from this thesis. Amongst these are the following:

- We are currently rebuilding the CFF/AML implementation on an NCR Tower under UNIX. Many of the problems which beset the SPERRY implementation will not be a factor on the Tower, namely memory size limitations and the inability to do screen oriented I/O on the SPERRY. This implementation will include a syntax-directed editor driven by CFF and interfacing with the context checker. Like [Despeyroux 84], we do not view context condition checking as being best performed incrementally but rather at intervals, possibly at the request of the user. The implementation will be significantly more robust than the SPERRY implementation, and we foresee its being used as a teaching tool in a language design course.
- AML has given us a number of insights into the context condition aspect of programming languages. We are currently working on the design of a modular

language with features for high level data abstraction; we would hope to design a language which gives such features whilst remaining context sensitively 'easy' to specify and work with.

- The work on recursion using temporal logic, and the procedure call rules and semantics, is adequate for this thesis but by no means complete (although this does not mean that the formal definition of AML is incomplete). An investigation into other aspects of 'procedure call' such as modules, iterators and generics would be most interesting.

- Temporal logic has now been used for verification of concurrent programs [Manna & Pnueli 83b], program synthesis [Manna & Wolper 84], and specification of hardware [Moszkowski 82], in addition to the work using it in this thesis. We feel this formalism would have utility in other areas and plan to investigate its applicability to specification in other areas, including abstract data types, context conditions directly, and distributed processing, as well as some further investigations into temporal logic as a semantics specification formalism.

- Because CFF/AML is formally defined using a generic temporal proof system, it should be possible to use a temporal logic–based verification system to automate the proofs of correctness of AML decorations.

- CFF/AML, despite its utility, is not terribly abstract. We would like to investigate the possibilities of developing methodologies for the extraction of CFF/AML specifications from more abstract specification formalisms. We have looked at denotational semantics for this, and notice that if an approach such as that of the denotational semantics in Polak's work [Polack 81] is taken then there is a correspondence between the auxiliary semantic functions and AML procedures, and the denotational conditions and AML decorations.

- Finally, there is one drawback to the CFF/AML approach which we would seek to rectify, namely that the domain of application of the language does not include dynamic semantics. A definite avenue for research would be to investigate ways of specifying the dynamic semantics of a programming language. In keeping with the philosophy of this project, we would not wish to build a denotational semantics interpreter, but rather look at a more accessible specification technique, possibly tying up with the work on temporal specification of programming language semantics in this thesis.

REFERENCES

[Ada 83] *Ada Programming Language*, ANSI/MIL–STD–1815A, United States Department of Defense, 1983.

[Addyman 80] Addyman, A. M., *A Draft Proposal for Pascal*, Pascal News 18, 1980.

[Aho & Ullmann 72] Aho, R. V. and J. D. Ullman, *Theory of Parsing, Translation and Compiling*, Volumes 1 and 2, Prentice–Hall, 1972.

[Ambriola et al 1984] Ambriola, V., G. E. Kaiser and R. J. Ellison, *An Action Routine Model for ALOE*, Computer Science Department Tech. Rep. CMU–CS–84–156, Carnegie–Mellon University, 1984.

[Ariely 75] Ariely, G., *Verification of Programmed Systems*, Ph.D Thesis, Carnegie–Mellon University, 1975.

[Barrett & Couch 79] Barret, W. A. and J. D. Couch, *Compiler Construction – Theory and Practice*, Science Research Associates, 1979.

[Bauer & Eickel 76] Bauer, F. L. and J. Eickel (eds), *Compiler Construction – an Advanced Course*, Springer–Verlag, 1976.

[Bjorner & Oest 80] Bjorner, D. and O. Oest, *Towards a Formal Description of Ada*, Lecture Notes in Computer Science 98, Springer–Verlag, 1980.

[Broy & Schmidt 82] Broy, M. and G. Schmidt (eds), *Theoretical Foundations of Programming Methodology*, D. Reidel Publishing Company, 1982.

[Blikle & Tarlecki 83] Blikle, A. and A. Tarlecki, *Naive Denotational Semantics*, in: [Mason 83], pp 345–346.

[Bochman & Ward 78] Bochman, G. V. and P. Ward, *Compiler Writing System for Attribute Grammars*, The Computer Journal, 21, 2, 1978, pp144–148.

[Campbell & Kirslis 84] Campbell, R. H. and P. A. Kirslis, *The SAGA Project – A System for Software Development*, in: P. Henderson (ed), *Proceedings of the ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments*, pp73–80, 1984.

[Cattell 78] Cattel, R., *Formalisation and Automatic Generation of Code Generators*, Ph.D Thesis, Carnegie–Mellon University, 1978.

[Chusho et al 83] Chusho, T., Wanatabe, T. and Hayashi, T., *A Language–Adaptive Programming Environment*, in: [Mason 83], pp 621–626.

[DeMillo et al 79] DeMillo, R. A., R. J. Lipton and A. J. Perlis, *Social Processes and Proofs of Theorems and Programs*, Communications of the ACM, 22, 5, 1979.

[Despeyroux 84] Despeyroux, T., *Executable Specification of Static Semantics*, in: [Kahn et al 84], pp215–234, 1984.

[Dijkstra 76] Dijkstra, E. W., *A Discipline of Programming*, Prentice–Hall, 1976.

[Dijkstra 82] Dijkstra, E. W., *Repaying Our Debts*, in: [Broy & Schmidt 82], pp135–141.

[Donahue 76] Donahue, J. E., *Complementary Definitions of Programming Languages*, Lecture Notes in Computer Science 42, Springer–Verlag, 1976.

[Donzeau–Gouge et al 80] Donzeau–Gouge, V., G. Huet, G. Kahn, and B. Lang, *Programming Environments based on Structure Editors: The Mentor Experience*, INRIA Rapport de Recherche 26, 1980.

[Donzeau–Gouge et al 83] Donzeau–Gouge, V., G. Kahn, B. Lang, B. Melese and E. Marcos, *Outline of a Tool for Document Manipulation*, in: [Mason 83], pp 615–620.

[Druker 83] Druker, J. S., *A Language Independent LR(1) Parser*, B.SC(Hons) Thesis, Computer Science Department, University of Cape Town, 1983.

[Evans et al 82] Evans, A., K. J. Butler, G. Goos and W. A. Wulf, *Draft Revised DIANA Reference Manual*, Tartan Laboratories, 1982.

[Farrow 82] Farrow, R. W., *Yet Another Translator–Writing System Based on Attribute Grammars*, in: Proceedings of the SIGPLAN 82 Symposium on Compiler Construction, ACM SIGPLAN Notices, 17,6, 1982, pp95–107.

[Friedman & Olivier 84] Friedman, I. M. and S. P. Olivier, *Implementation of a Semantic Evaluation for a Language Independent Programming Environment*, B.SC(Hons) Thesis, Computer Science Department, University of Cape Town, 1984.

[Gill 76] Gill, A., *Applied Algebra for the Computer Sciences*, Prentice–Hall, 1976.

[Graham 80] Graham, S., *Table–Driven Code Generation*, Computer, 13, 8, 1980.

[Guttag et al 78] Guttag, J. V., E. Horowitz and D. R. Musser, *The Design of Data Type Specifications*, in: [Yeh 78], pp 60–79, 1978.

[Goldblatt 82] Goldblatt, R., *Axiomatising the Logic of Programming Language*, Lecture Notes in Computer Science 130, Springer–Verlag, 1982.

[Gougen et al 78] Gougen, J. A., J. W. Thatcher and E. W. Wagner, *An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types*, in: [Yeh 78], pp 80–149, 1978.

[Gries 81] Gries, D., *The Science of Programming*, Springer–Verlag, 1981.

[Griffiths 76] Griffiths, D., *Introduction to Compiler–Compilers*, in [Bauer & Eickel 76], pp 356–365, 1976.

[Habermann & Notkin 82] Habermann, A. N. and D. S. Notkin, *The Gandalf Software Development Environment*, Tech. Rep., Computer Science Department, Carnegie–Mellon University, 1982.

[Hoare 71] Hoare, C. A. R., *Procedures and Parameters - An Axiomatic Approach*, in: *Symposium on Semantics of Algorithmic Languages*, Lecture Notes in Mathematics 188, Springer–Verlag, 1971.

[Hoare & Wirth 73] Hoare, C. A. R. and N. Wirth, *An Axiomatic Definition of the Programming Language Pascal*, Acta Informatica, 2, pp 335–355, 1973.

[Holt et al 82] Holt, R. C., R. C. Cordy and D. B. Wortman, *An Introduction to S/SL: Syntax/Semantics Language*, ACM Transactions on Programming Languages and Systems, 4, 2, pp 149–178, April 1982.

[Horowitz 84] Horowitz, E., *Fundamentals of Programming Languages (2nd Edition)*, Springer–Verlag, 1984.

[Igarashi et al 75] Igasashi, S., R. L. London and D. C. Luckham, *Automatic Program Verification I: A Logical Basis and its Implementation*, Acta Informatica, 4, pp148–182, 1975.

[Jensen & Wirth 74] Jensen, K. and N. Wirth, *Pascal User Manual and Report*, Springer–Verlag, 1974.

[Johnson 75] Johnson, S. C., *YACC - Yet Another Compiler Compiler*, Computing Science Tech. Rep. 32, Bell Laboratories, 1975.

[Johnson 83] Johnson, G. F., *An Approach to Incremental Semantics*, Ph.D Thesis, University of Wisconsin–Madison, 1983.

[Kaplan 83a] Kaplan, S. M., *CFF/AML Users Reference Manual*, Computer Science Tech. Rep., University of Cape Town, 1983.

[Kaplan 83b] Kaplan, S. M., *CFF/AML Report*, Computer Science Tech. Rep., University of Cape Town, 1983.

[Kaiser 84a] Kaiser, G. E., *Tree Manipulation Language Users Manual*, Technical Report RTL–84–TM–106, Siemens Research and Technology Laboratories, 1984.

[Kaiser 84b] Kaiser, G. E., *DOSE Tree Manipulation Language Reference Manual*, Technical Report RTL–84–TR–030, Siemens Research and Technology Laboratories, 1984.

[Kaiser & Feiler 84] Kaiser, G. E. and P. Feiler, *Generation of Language–Oriented Editors*, Computer Science Tech. Rep., Carnegie–Mellon University, 1984.

[Kastens 80] Kastens, U., *Ordered Attribute Grammars*, Acta Informatica, 13, pp 229–256, 1980.

[Kastens et al 82] Kastens, U., B. Hutt and E. Zimmermann, *GAG – A Practical Compiler Compiler*, Lecture Notes in Computer Science 141, Springer–Verlag, 1982.

[Kahn et al 83] Kahn, G., B. Lang, B. Melese and E. Marcos, *METAL: A Formalism to Specify Formalisms*, Science of Computer Programming, 3, pp151–187, August 1983.

[Kahn et al 84] Kahn, G., D. B. MacQueen and G. Plotkin (eds), *Semantics of Data Types*, Lecture Notes in Computer Science 173, Springer–Verlag, 1984.

[Kennedy & Warren 76] Kennedy, K. and S. K. Warren, *Automatic Generation of Efficient Evaluators for Attribute Grammars*, in: Conference Record of the Third ACM Symposium on Principles of Programming Languages, 1976, pp32–49.

[Kfoury et al 82] Kfoury, A. J., R. N. Moll and M. A. Arbib, *A Programming Approach to Computability*, Springer–Verlag, 1982.

[Knuth 68] Knuth, D., *Semantics of Context–Free Languages*, Mathematical Systems Theory, 2, 2, pp127–1145, 1968.

[Knuth 71] Knuth, D., *Semantics of Context–Free Languages: Correction*, Mathematical Systems Theory, 5, 1, pp95–96, 1971.

[Koskimies 84] Koskimies, K., *A Specification Language for One–Pass Semantic Analysis*, in: Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, pp 179–189, 1984.

[Koster 76a] Koster, C. H. A., *Using the CDL Compiler–Compiler*, in: [Bauer & Eickel 76], pp 366–426, 1976.

[Koster 76b] Koster, C. H. A., *Two–Level Grammars*, in: [Bauer & Eickel 76], pp146–169.

[Lamport 83] Lamport, L., *What Good is Temporal Logic*, in: [Mason 83], pp657–668, 1983.

[Liskov et al 81] Liskov, B., R. Atkinson, T. Bloom, E. Moss, J. C. Schaeffert, R. Scheifler and A. Snyder, *CLU Reference Manual*, Lecture Notes in Computer Science 114, Springer–Verlag, 1981.

[Manna 74] Manna, Z., *Mathematical Theory of Computation*, McGraw–Hill, 1974.

[Manna 80] Manna, Z., *Logics of Programs*, in: S. H. Lavington (ed), *Information Processing 80 – Proceedings of the IFIP Congress 1980*, North–Holland, 1980.

[Manna 82] Manna, Z., *Verification of Sequential Programs: Temporal Axiomatization*, in: [Broy & Schmidt 82], pp 53–103, 1982. Also Tech. Rep. STAN–CS–81–877, Computer Science Department, Stanford University, 1981.

[Manna & Pnueli 81] Manna, Z. and A. Pnueli, *Verification of Concurrent Programs: Temporal Proof Principles*, in: D. Kozen (ed), *Proceeding of the Workshop on Logics of Programs*, Lecture Notes in Computer Science 131, pp 200–252, Springer–Verlag, 1981. Also Tech. Rep. STAN–CS–81–843, Computer Science Department, Stanford University, 1981.

[Manna & Pnueli 82] Manna, Z. and A. Pnueli, *Verification of Concurrent Programs: The Temporal Framework*, in: Boyer, C. and J. S. Moore (eds), *The Correctness Problem in Computer Science*, International Lecture Series in Computer Science, Academic Press, 1982. Also Tech. Rep. STAN–CS–81–836, Computer Science Department, Stanford University, 1981.

[Manna & Pnueli 83a] Manna, Z. and A. Pnueli, *How to Cook a Temporal Proof System for your Pet Programming Language*, in: *Proceedings of the Symposium on Principles of Programming Languages*, Austin, Texas, 1983. Also Tech. Rep. STAN–CS–82–954, Computer Science Department, Stanford University, 1982.

[Manna & Pnueli 83b] Manna Z. and A. Pnueli, *Verification of Concurrent Programs: A Temporal Proof System*, Tech. Rep. STAN–CS–83–967, Computer Science Department, Stanford University.

[Manna & Wolper 84] Manna, Z. and P. Wolper, *Synthesis of Communicating Processes from Temporal Logic Specifications*, ACM Transactions on Programming Languages and Systems, 6,1,1984.

[Marcotty et al 76] Marcotty, M., H. F. Ledgard and G. V. Bochmann, *A Sampler of Formal Definitions*, ACM Computing Surveys, 8, June 1976.

[Mason 83] Mason, R. E. A. (ed), *Information Processing 83 – Proceedings of the IFIP Congress 1983*, North–Holland, 1983.

[McCarthy & Levin 65] McCarthy, J. and S. Levin, *LISP 1.5 Programmers Manual (2nd ed), M.I.T. Press, 1965.*

[McDermott 84] McDermott, T. S., *Developing an Intelligent Editor*, Questionaes Informatica, 3, 1984.

[McGettrick 82] McGettrick, A. D., *Program Verification Using Ada*, Cambridge Computer Science Texts 13, Cambridge University Press, 1982.

[McKeeman et al 70] McKeeman W. M., J. J. Horning and D. B. Wortmann, *A Compiler Generator*, Prentice–Hall, 1970.

[Medina–Mora 82] Medina–Mora, R., Syntax–Directed Editing: Towards Integrated Programming Environments, Ph.D Thesis, Carnegie–Mellon University, 1982.

[Medina–Mora et al 83] Medina–Mora, R., R. J. Ellison, D. B. Garlan, G. E. Kaiser and D. S. Notkin, *ALOE Users and Implementors Guide*, Tech. Rep., Carnegie–Mellon University, 1983.

[Mosses 82] Mosses, P.D., *SIS – Semantics Implementation System (User manual and Reference Guide)*, DAIMI MD–30, Computer Science Department, Aarhus University, 1982.

[Moszkowski 82] Moszkowski, B., *A Temporal Logic for Multi–Level Reasoning about Hardware*, Tech. Rep. STAN–CS–82–952, Department of Computer Science, Stanford University, 1982.

[Pagan 81] Pagan, F. G., *Formal Specification of Programming Languages*, Prentice–Hall, 1981.

[Pleban 84] Pleban, U. F., *Compiler Prototyping using Formal Semantics*, Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, pp 94–105, 1984.

[Polack 81] Polack, W., *Compiler Specification and Verification*, Lecture Notes in Computer Science 124, Springer–Verlag, 1981.

[Pratt 84] Pratt, T. W., *Programming Languages – Design and Implementation (2nd Edition)*, Prentice–Hall, 1984.

[Raiha et al 78] Raiha, K.–J., M. Saarinen, E. Soisalon–Soininen and M. Tienari, *The Compiler Writing System HLP (Helsinki Language Processor)*, Report A–1878–2, Department of Computer Science, University of Helsinki, 1978.

[Reps 84] Reps, T., *Generating Language–Based Environments*, M. I. T. Press, 1984.

[Reps et al 83] Reps, T., T. Teitelbaum and A. Demers, *Incremental Context–Dependent Analysis for Language–Based Editors*, ACM Transactions on Programming Languages and Systems, 5, 3, pp 449–477, 1983.

[Reps & Teitelbaum 84] Reps, T. and T. Teitelbaum, *The Synthesizer Generator*, Proceedings of the ACM SIGPLAN/SIGSOFT Symposium on Software Development Environments, ACM SIGPLAN Notices 9, 3, pp 42–48, 1984.

[Rose & Roper 83] Rose, G. and T. Roper, *Generation of Program Preparation Systems for Formatted Languages*, in: [Mason 83], pp931–936, 1983.

[Rosselet 84] Rosselet, A., *Definition and Implementation of Context Conditions for Programming Languages*, Ph.D Thesis, University of Toronto, 1984.

[Scott 82] Scott, D., *Lectures on a Mathematical Theory of Computation*, in: [Broy & Schmidt 82], pp145–292, 1982.

[Standish 78] Standish, T. A., *Data Structures - An Axiomatic Approach*, in: [Yeh 78], pp 30–59, 1978.

[Stoy 77] Stoy, J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, M. I. T. Press, 1977.

[Stoy 82] Stoy, J. E., *Semantic Models*, in: [Broy & Schmidt 82], pp 293–325, 1982.

[Suzuki 76] Suzuki, N., *Automatic Verification of programs with Complex Data Structures*, Ph.D Thesis, Stanford University, 1976.

[Teitelbaum & Reps 81] Teitelbaum, T. and T. Reps, *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment*, Communications of the ACM, 24, 9, 1981.

[Tennant 76] Tennant, R. D., *The Denotational Semantics of Programming Languages*, Communications of the ACM, 19, 8, pp437–453, 1976.

[Vuillemin 74] Vuillemin, J., *Proof Techniques for Recursive Programs*, Ph.D Thesis, Stanford University, 1974.

[Waite 76] Waite, W. G., *Semantic Analysis*, in: [Bauer & Eickel 76], pp 157–168, 1976.

[Waite & Goos 84] Waite, W. G. and G. Goos, *Compiler Construction*, Springer–Verlag, 1984.

[Watt 79] Watt, D. A., *An Extended Attribute Grammar for Pascal*, SIGPLAN Notices, 14, 2, 1979.

[Wegner 72] Wegner, P., *The Vienna Definition Language*, ACM Computing Surveys, 4,1,1972.

[Wetherell 81] Wetherell, C. S., *Problems with the Ada Reference Grammar*, SIGPLAN Notices, 16, 9, 1981.

[Wirth 71] Wirth, N., *The Design of a Pascal Compiler*, Software Practice and Experience, 1, pp309–333, 1971.

[Wirth 76] Wirth, N., *Algorithms + Data Structures = Programs*, Prentice–Hall, 1976.

[Wirth 83] Wirth, N., *Programming in Modula–2 (2nd Ed.)*, Springer–Verlag, 1983.

[Wulf 80] Wulf, W. A., *PQCC: A Machine–Relative Compiler Technology*, Tech. Rep., Department of Computer Science, Carnegie–Mellon University, 1980.

[Wulf et al 81] Wulf, W. A., M. Shaw, P. N. Hilfinger and L. Flon, *Fundamental Structures of Computer Science*, Addison–Wesley, 1981.

APPENDIX I – CFF/AML IN CFF/AML

This appendix contains a description of the syntax and context conditions of CFF/AML in CFF/AML.

There were two possible ways to go about doing this specification; the first would have been to have the specification as a two–pass object, where all the rulenames, name tables and simple variable names are placed in name tables in pass 1, and then the required checks carried out in pass 2. The alternative approach would be to use auxiliary tables to hold forward references, and remove these when they are resolved. As the latter approach has been used in examples in the body of the thesis, the former is adopted here. Each decoration is prefixed by a passname. All those decorations with the passname 'setup' are executed in pass one, and all those with the passname 'check' in pass two.

Note that because of the dynamic nature of the name table scoping, questions of whether or not a name table has the correct attributes for certain operations, or whether or not a table is local in a given scope are runtime questions, and are therefore defined in the formal semantics of part III of the thesis, and not handled here.

The full specification is:

```
[ PASS setup, check ;
  DECLARE
    FLAG dyad, monad, isfunc, needsparams, haspasses,issubtree ;
]

"cff/aml-spec"
[ setup:
  NEWSCOPE
    SET    ints, strings, flags, symbols, rules,
           passes, procs, funcs ;
    STACK  funcstack ;
]
?aml-variables-section
<cff-rules_>
?aml-subprograms-section [ setup: ENDSCOPE ] %

"cff-rules"
'"' identifier[ setup:
                  if unique(rules)
                    then insert(rules)
                    else error('rule already declared') ] '"'
<cff-symbol_> '%' %

"cff-symbol"
```

```
 identifier [check: if unique(rules)
                          then error('Illegal rule name')]
¦string
¦metasymbol
¦list
¦aml-decoration %

"metasymbol"
 '#'
¦'|'
¦'.'
¦'?'
¦'!' %

"list"
'<' ?<cff-symbols_> '_' ?<cff-symbols_> '>' %

"aml-decoration"
 '[' [ check:
       if haspasses and not subtree(pass-indicator)
          then
             error('There must be a pass indicator')
     ]?pass-indicator
<aml-statement_;> ']' %

"pass-indicator"
identifier[ check:
              if unique(passes)
                 then error('invalid pass indicator')
           ] ':' %

"aml-variables-section"
 '[' DECLARE
?passname-declaration
<declaration;_> ']' %

"passname-declaration"
[setup:
 set(haspasses)
]PASS <identifier[ setup:
                      if unique(passes)
                         then insert(passes)
                         else error('repeated passname')]_> ; %

"declaration"
 string-declaration
¦flag-declaration
```

```
|integer-declaration %

"string-declaration"
STRING <identifier[ setup:
                        if unique(strings+ints+flags)
                          then insert(strings)
                          else error('Identifier already declared')
                      ]→> %

"flag-declaration"
FLAG <identifier[ setup:
                      if unique(strings+ints+flags)
                        then insert(flags)
                        else error('Identifier already declared')
                    ]→> %

"integer-declaration"
INTEGER <identifier[ setup:
                       if unique(strings+ints+flags)
                         then insert(flags)
                         else error('Identifier already declared)
                     ]→> %

"aml-statement"
  conventional-statement
|scope-statement
|tree-statement
|stack-statement %

"conventional-statement"
  assign
|block
|error
|exit
|if
|procedure-call
|null
|repeat
|reset
|restore
|save
|set
|warning
|while %

"assign"
identifier[ check:
```

```
                if visible(ints+strings+flags) or top(funcstack)
                   then
                      push(type)
                   else
                      error('Undeclared identifier')
             ] ':=' expression[acomp] %

"block"
BEGIN
  <aml-statement_;>
END %

"if"
IF expression[ check: checkbool
               ] THEN aml-statement ?else %

"else"
ELSE aml-statement %

"while"
WHILE expression[ check: checkbool
                  ] DO aml-statement %

"repeat"
REPEAT <aml-statement_;> UNTIL expression[ check: checkbool
                                           ] %

"procedure-call"
identifier[ check: if unique(procs) then
                        error('Not a procedure name')] %

"exit"
EXIT %

"null"
%

"save"
SAVE ( identifier[ check:
                     if unique(ints+flags+strings)
                        then error('Illegal identifier')] ) %

"restore"
RESTORE ( identifier[ check:
                        if unique(ints+flags+strings)
                           then error('Illegal identifier')] ) %
```

```
"set"
SET (identifier[ check: if not visible(flags) and
                                    not contents=top(funcstack)
                                  then
                                    error('Not a flag identifier')] ) %

"reset"
RESET (identifier[ check: if not visible(flags) and
                                     not contents=top(funcstack)
                                   then
                                     error('Not a flag identifier')]

"error"
ERROR ( string ) %

"warning"
WARNING ( string ) %

"scope-statement"
  start-scope
|finish-scope
|dump
|acomp-table-load
|mcomp-table-load
|dcomp-table-load
|insert
|delete
|copy
|clear
|unstack
|front
|next
|aml-scope
|tree-scope %

"start-scope"
NEWSCOPE
? <symbol-table-declaration;_> %

"symbol-table-declaration"
  set-declaration
|list-declaration
|stack-declaration %

"symbol-declaration"
SET <identifier[ setup:
                    if unique(symbols)
```

```
                        then insert(symbols)] ¬> %

"list-declaration'
LIST <identifier[ setup:
                    if unique(symbols)
                      then insert(symbols)]¬> %

"stack-declaration"
STACK <identifier[ setup:
                    if unique(symbols)
                      then insert(symbols)]¬> %

"finish-scope"
ENDSCOPE %

"dump"
DUMP ( identifier[ check: checksymbol] ) %

"acomp-table-load"
LOAD-ACOMP-TABLE(expression[check: checkstring
                                ], expression[check: checkstring]) %

"mcomp-table-load"
LOAD-MCOMP-TABLE(expression[check: checkstring
                                ], expression[check: checkstring
                                                  ], expression[check:
                                                                  checkstring
                                                               ]) %


"dcomp-table-load"
LOAD-DCOMP-TABLE(expression[check: checkstring
                                ], expression[check: checkstring],
                    expression[check: checkstring
                                ], expression[check: checkstring]) %

"insert"
INSERT ( ?stringpart identifier[check: checksymbol] ) %

"delete"
DELETE ( ?stringpart identifier[check: checksymbol] ) %

"stringpart"
expression[check: checkstring], %

"copy"
COPY ( identifier[check: checksymbol] ) %
```

```
"clear"
CLEAR ( identifier[check: checksymbol] ) %

"unstack"
UNSTACK ( identifier[check: checksymbol] ) %

"front"
FRONT ( identifier[check: checksymbol] ) %

"next"
NEXT (identifier[check: checksymbol] ) %

"aml-scope"
AMLSCOPE %

"tree-scope"
TREESCOPE %

"tree-statement"
  SAVEPOSITION
¦RESTOREPOSITION
¦son
¦father
¦INTOLIST
¦NEXT-ITERATION
¦MARK
¦RELEASE %

"son"
SON ?rule-place %

"rule-place"
( identifier[check:
                  if unique(rules)
                     then error('not a CFF rulename] ) %

"father"
FATHER ?rule-place %

"stack-statements"
` push
¦pop
¦ACOMP
¦MCOMP
¦DCOMP %

"push"
```

```
PUSH ( expression[check: checkstring] ) %

"pop"
POP ?pop-place %

"pop-place"
( identifier[check:
                if unique(strings+ints+flags)
                then
                   error('Not an AML simple identifier] ) %

"expression"
[ check:
   save(monad) ;
   save(dyad) ;
   reset(monad) ;
   reset(dyad)
] ?sign <<factor[ check:
                     if monad then
                        begin
                           mcomp ;
                           reset(monad)
                        end ;
                     if dyad then
                        begin
                           dcomp ;
                           reset(dyad)
                        end
                  ]_multop[ check: set(dyad)
                           ]>_addop[ check: set(dyad)
                                    ]> ?predicate [ check:
                                                      restore(monad) ;
                                                      restore(dyad)] %

"predicate"
relop ?sign <<factor[ check:
                        if monad then
                           begin
                              mcomp ;
                              reset(monad)
                           end ;
                        if dyad then
                           begin
                              dcomp ;
                              reset(dyad)
                           end
```

```
                                ]>_addop[ check: set(dyad)
                                        ]>[check: dcomp] %

"factor"
 identifier[check:
                if visible(ints+flags+strings) or std-function
                   then
                      push(type)
                   else
                      error('not an allowable factor')
                if not subtree(arguments) and needsparams
                   then error('missing function arguments')
                ] ?argument %
¦( expression )
¦string [check: push('string')]
¦integer[check: push('integer')]
¦NOT factor[check:
                if stacktop <> 'flag'
                   then
                      error('must be a boolean type')]    %

"argument"
 simple-argument
¦argument-list %

"simple-argument"
(identifier[ check: if not(needsparams) then
                              error('illegal parameters')
                           else begin
                              if issubtree then
                                 begin
                                    if unique(rules) then
                                       error('illegal identifier') ;
                                 end
                              else
                                  checksymbol
                           end] ) %

"argument-list"
[if not needsparams
        then error('Illegal parameters')
]( ?stringpart <identifier[check:
                                    checksymbol]_+> ) %

"sign"
 + [check: push('+'); set(monad)]
¦ [check: push('-'); set(monad)]%
```

```
"addop"
  + [check: push('+')]
|- [check: push('-')]
|OR[check: push('OR')] %

"multop"
AND[check: push('AND')] %

"relop"
  =     [check: push('=')]
|'<>'[check: push('<>')]
|'<' [check: push('<')]
|'>' [check: push('>')]
|'<= [check: push('<=')]'
|'>= [check: push('>=')]'
|MATCHES[check: push('MATCHES')] %

"aml-subprograms-section"
  '[' <subprogram-declaration;_> ']' %

"subprogram-declaration"
  procedure
|function %

"procedure"
PROCEDURE identifier[setup:
                        if unique(procs+funcs+strings+ints+flags)
                          then
                            insert(procs)
                          else
                            error('Identifier already declared')
                       ] ; block %

"function"
FUNCTION identifier[setup:
                        if unique(procs+funcs+strings+ints+flags)
                          then
                            insert(procs)
                          else
                            error('Identifier already declared')
                      ][check: insert(funcstack)
                       ] RETURN type ; block [check:
                                              unstack(funcstack)] %

"type"
  STRING
```

```
¦FLAG
¦INTEGER %

[ procedure checkbool ;

    begin
      if stacktop <> 'flag'
        then
          error('Expression type not boolean') ;
      pop ;
    end ;

  procedure checksymbol ;

    begin
      if unique(symbols)
        then
          error('not a symbol table name')
    end ;

  procedure checkstring ;

    begin
      if stacktop <> 'string'
        then
          error('Expression type not string') ;
      pop ;
    end ;

function type return string ;

    begin
      if isfunc then begin
        if (contents='visible') or (contents='local')
        or (contents='unique') or (contents='subtree')
        or (contents='empty) or (contents='eol')
        or (contents='another-elt'
          then type := 'flag' ;
        if (contents='contents') or (contents='stacktop')
        or (contents='rule')
          then type := 'string' ;
        if (contents='value') or (contents='length')
          then
            type := 'integer'
      end else
      begin
        if rule='function-declaration'
```

```
            then
               begin
                  son(type) ;
                  type := contents ;
               end ;
         if rule = 'integer-declaration' then type := 'integer' ;
         if rule = 'string-declaration'  then type := 'string' ;
         if rule = 'flag-declaration'    then type := 'flag' ;
      end ;
   end ;

function std-function return flag ;

   begin
      reset(isfunc) ;
      reset(needsparams) ;
      reset(issubtree) ;
      if (contents='visible') or (contents='local')
      or (contents='unique') or (contents='empty')
      or (contents='top') or (contents='subtree')
      or (contents='eol') then
         begin
            set(isfunc) ;
            set(needsparams)
            if contents='subtree' then set(issubtree)
         end ;

      if (contents='contents') or (contents='stacktop')
      or (contents='length') or (contents='rule')
      or (contents='another-elt')
         then set(isfunc) ;

      if isfunc then set(std-function) else reset(std-function)
   end ;
]
```

DYADIC COMPATIBILITY TABLE FOR AML			
OPERATOR	OPERAND	OPERATOR	RESULT
integer	+	integer	integer
integer	-	integer	integer
string	+	string	string
*	=	*	*
*	<>	*	*
string	matches	string	flag
integer	<	integer	flag
integer	>	integer	flag
integer	<=	integer	flag
integer	>=	integer	flag
flag	AND	flag	flag
flag	OR	flag	flag

MONADIC COMPATIBILITY TABLE FOR AML		
OPERATOR	OPERAND	RESULT
+	integer	integer
-	integer	integer

ASSIGNMENT COMPATIBILITY TABLE FOR AML	
LEFTHAND SIDE	RIGHTHAND SIDE
*	*

APPENDIX II – PASCAL

This appendix presents a CFF/AML description of Pascal.

A 'profile' approach to checking subprogram parameters has been adopted in this appendix as opposed to the parameter–for–parameter check which would normally be used. The usual approach is illustrated in the example in chapter 8. The profiling approach used here makes it possible to overload procedures, and is an extension of the example in chapter 9 for operator overloading. Array parameters (subscripts) are still checked on a one–for–one basis.

```
[ DECLARE
    FLAG   monad, mult, add, fwd, anotherelt, noparams, eflag, nametype,
           std-types, std-consts ;
    STRING subname, type, casetype, profbld, x
]
"program"
[newscope ;
 LIST  vars ;
 SET   labels, constants, types,  procs, funcs, unusedlabels,
       fwdsubs, fwdtypes, scalars, procprofile, funcprofile ;
 STACK  funcnames ;
]PROGRAM identifier;
?labels ?constants ?types ?variables ?<  #subprocedure;

_>BEGIN
<?label  #statement_;
>
END.[dump-leftovers]%

"labels"
LABEL <integer[ if unique(labels)
                  then
                    begin insert(unusedlabels) ; insert(labels) end
                  else
                    error('label already defined');
                if (value < 1) or (value > 9999)
                  then
                    error('value must be in interval 1..9999');
              ]_,> ;
 %

"constants"
CONST <identifier[ if unique(constants + types + vars + procs + funcs)
                     then
                       insert(constants)
```

```
                        error('identifier already declared') ;
                    ] = constant;_
        >%

"types"
TYPE <identifier[ if unique(constants + types + vars + procs + funcs)
                        then
                            begin
                                insert(types) ;
                                if local(fwdtypes)
                                    then
                                        delete(fwdtypes)
                                end
                        else
                            error('identifier already declared') ;
                    ] = type ;_
      >
 %

"variables"
VAR <<identifier[ if unique(constants + types + vars + procs + funcs)
                        then
                            insert(vars)
                        else
                            error('identifier already declared')
                    ]_> ':' type; _>
%

"subprocedure"
proc|func%

"label"
integer[ if local(labels)
            then
                if local(unusedlabels)
                    then
                        delete(unusedlabels)
                    else
                        error('label already used on another statement')
            else
                error('label not declared')] ':' %

"statement"
 assignment|subroutine|block|if|case|while|repeat|for|with|goto|null
 |read|readln|write|writeln %

"read"
```

```
READ( <lhs␣,> ) %

"write"
WRITE( <expression␣,> ) %

"writeln"
WRITELN( <expression␣,> ) %

"readln"
READLN( <lhs␣,> ) %


"constant"
 str[ type := 'char' ]
 ¦id:¦numeral:¦signed-integer:%

"type"
 typeid
¦scalar
¦subrange
¦pointer
¦array
¦file
¦set
¦#record%

"proc"
PROCEDURE identifier[ subname := contents ;
                      profbld := contents ;
                      mark ;
                      reset(noparams) ;
                      if local(fwdsubs)
                         then
                            begin
                               set(noparams) ;
                               delete(fwdsubs)
                            end
                         else
                            if unique(constants + types + vars +
                                      procs + funcs)
                               then
                                  insert(procs)
                               else
                                  error('identifier already declared');
                      newscope ;
                      LIST    params, vars ;
                      SET     labels, constants, types,  procs, funcs,
```

```
                                    unusedlabels, fwdsubs, fwdtypes, scalars,
                                    procprofile, funcprofile;]?parameters ;
[ amlscope ; father(proc) ; insert(profbld, procprofile) ]
body[ dump-leftovers;
      release ;
      endscope ]%

"func"
FUNCTION identifier[ save(subname) ;
                     subname := contents ;
                     profbld := contents ;
                     mark ;
                     reset(noparams) ;
                     if local(fwdsubs)
                        then
                           begin
                              set(noparams) ;
                              delete(fwdsubs)
                           end
                        else
                           if unique(constants + types + vars +
                                     procs + funcs)
                              then
                                begin
                                 insert(funcs) ;
                                 insert(funcnames) ;
                                end
                              else
                                 error('identifier already declared') ;
                     newscope ;
                     LIST     params, vars ;
                     SET      labels, constants, types, procs, funcs,
                              unusedlabels, fwdsubs, fwdtypes, scalars,
                              procprofile, funcprofile ;
                   ] ?parameters ':' typeid ;
[ amlscope ; father(func) ; insert(profbld, funcprofile) ]
body[ dump-leftovers ;
      release ;
      delete(subname,funcnames) ;
      restore(subname) ;
      endscope ]%

"assignment"
lhs [ push(type) ] ':=' expression [acomp] %

"lhs"
identifier[ if not (visible(vars) or top(funcnames))
```

```
                    then
                      begin
                        error('invalid identifier reference');
                        type := std-error ;
                      end
                    else
                      type-evaluate
                ]?reference[ if 'file' matches type
                                then
                                  error('cannot assign to a file type') ;
                             ] %

"subroutine"
identifier[ if visible(procs)
                    then
                      profbld := contents
                    else
                      error('not a procedure identifier')
                ]?arguments [ if not visible(profbld,  procprofile)
                                then
                                  error('illegal parameters') ] %

"block"
BEGIN
<?label   #statement_;
>
END%

"if"
IF expression[ if stacktop <> 'boolean'
                      then
                        error('expression must be of boolean type') ;
                    pop ]
  THEN
?label   #statement ?else %

"case"
[save(casetype)
]CASE expression[ pop(casetype) ;
                      if ('real' matches casetype) or
                          ('file' matches casetype)
                          then
                            error('expression type must be scalar')
                    ] OF <
    <caselabel_> ':' ?label #statement_;>
END[ restore(casetype)]%
```

```
"caselabel"
 str[ if not ('char' matches casetype)
              then
                 error('incompatible caselabel type') ]
|?sign int[ if not ('integer' matches casetype)
                        then
                           error('incompatible casetype') ]
|id [ if type <> casetype
    then
       error('incompatible caselabel type')] %

"while"
WHILE expression[ if stacktop <> 'boolean'
                          then
                             error('expression must be of boolean type') ;
                       pop] DO
?label   #statement%

"repeat"
REPEAT ?<
?label   #statement_;>
UNTIL expression [ if stacktop <> 'boolean'
                            then
                               error('expression must be of boolean type') ;
                         pop ]%

"for"
FOR identifier[ if visible(vars)
                          then
                            begin
                               type-evaluate ;
                               push(type) ;
                               push(type) ;
                               if ('array' matches type) or
                                   ('record' matches type)   or
                                   ('file' matches type) or
                                   ('real' matches type)
                                  then
                                     error('invalid for loop control variable') ;
                            end
                          else
                            error('identifier not declared')
                    ] ':=' expression[ acomp ] thru expression[ acomp ] DO
?label   #statement%

"with"
WITH |newscope;
```

```
        LIST vars; ]<identifier[ if visible(vars)
                                    then
                                       type-evaluate
                                    else
                                       error('identifier not declared')
                                 ]?reference[ if 'record' matches type
                                                 then
                                                    begin
                                                       amlscope ;
                                                       son(fields) ;
                                                       copy(vars) ;
                                                    end
                                                 else
                                                 error('not a record variable')
                                              ]→> DO
?label   #statement[ endscope ]%

"goto"
GOTO integer[ if not visible(labels)
                 then
                    error('undeclared label') ]%

"null"
%

"id"
?sign!ident%

"ident"
identifier [ if visible(constants)
                        then
                           begin
                              type-evaluate ;
                              if monad
                                 then
                                    begin
                                       push(type);
                                       mcomp;
                                       reset(monad) ;
                                       pop(type) ;
                                    end ;
                           end
                        else
                           error('constant identifier not declared')] %

"numeral"
?sign!num[ if monad
```

```
                    then
                      begin
                        reset(monad);
                        pop
                      end ]%

"signed-integer"
?sign!int[ type := 'integer' ; if monad
                    then
                      begin
                        reset(monad) ;
                        pop
                      end ]%

"scalar"
(<identifier[ if unique(constants + types + vars + procs + funcs)
                    then
                      begin
                        insert(constants);
                        insert(scalars);
                      end
                    else
                      error('identifier already declared');
              ]→>)%

"subrange"
lowerbound[ push(type) ]'..'upperbound[ push(type);
                                                acomp ] %

"lowerbound"
constant%

"upperbound"
constant%

"pointer"
^ptrid %

"ptrid"
identifier [ if unique(constants + types + vars + funcs + procs)
                    then
                      insert(fwdtypes)
                    else
                      error('identifier already declared') ] %

"array"
?packed ARRAY '[' index ']' OF type%
```

```
"index"
<rangetype_,> %

"file"
?packed FILE OF type%

"set"
SET OF type%

"record"
?packed RECORD   [ newscope ;
                     LIST vars ;]
   #fields .
END[ endscope ]%

"parameters"
[ if noparams
     then
        error('cannot have parameters on second declaration')
](<?var <identifier[ if unique(vars)
                          then
                             begin
                                insert(vars) ;
                                insert(params) ;
                             end
                          else
                             error('identifier already declared')
                       ]_,> ':' typeid[ profbld := profbld + x ;
                                        ]_;>)%

"body"
 FORWARD[ amlscope ; father ; insert(subname,fwdsubs) ]
¦EXTERNAL
¦definition%

"reference"
<selector_>%

"expression"
[ reset(monad) ;
   reset(add)    ;
   reset(mult)
]?sign<<factor[ if monad
                    then
                       begin
                          mcomp          ;
```

```
                        reset(monad)
                     end ;
                  if mult
                     then
                        begin
                           dcomp          ;
                           reset(mult)
                        end ;
                  if add
                     then
                        begin
                           dcomp ;
                           reset(add)
                        end
            ]_multop[ set(mult) ]>_addop[ set(add) ]> ?predicate%

"arguments"
( <expression [ profbld := profbld + stacktop ;
                  pop ]_,> ) %

"else"
   ELSE
?label   #statement%

"thru"
 TO¦DOWNTO%

"sign"
 +[ push('+') ; set(monad) ]
¦-[ push('-') ; set(monad) ] %

"packed"
PACKED %

"rangetype"
 identifier[ if visible(types)
                  then
                     begin
                        type-evaluate ;
                        if not (('scalar' matches type) or
                                 ('subrange' matches type))
                           then
                              error('not a subrange or scalar type')
                     end
               else
                  error('not a type identifier')]
¦subrange¦scalar%
```

```
"fields"
 fixed|variant%

"var"
VAR %

"definition"
?labels?constants?types?variables?<    #subprocedure;
_> BEGIN
<?label   #statement_;
>
END%

"selector"
[ if not ('array' matches type)
      then
        begin
          error('not an array type');
          skip(subscript)
        end ]subscript
|[ if not ('record' matches type)
       then
         begin
           error('not a record type')  ;
           skip(field)
         end
       else
         son(fields) ] field
|[ if not ('pointer' matches type)
       then
         error('not a pointer type') ]^%

"factor"
 name
|num[ push('real') ]
|int[ push('integer') ]
|str[ push('char') ]
|subexpression
|negation
|setdescription%

"multop"
   *[ push ('*') ; set(mult) ]
| /[ push ('/') ; set(mult) ]
| DIV[ push ('div') ; set(mult) ]
| MOD[ push ('mod') ; set(mult) ]
```

```
¦AND[ push ('and') ; set(mult) ]%
"addop"
  +[ push('+') ; set(add) ]
¦-[ push('-') ; set(add) ]
¦OR[ push('or') ; set(add) ]%

"predicate"
[ reset(monad);
   reset(mult) ;
   reset(add)
]relop ?sign<<factor[ if monad
                        then
                          begin
                            mcomp ;
                            reset(monad)
                          end ;
                      if mult
                        then
                          begin
                            dcomp ;
                            reset(mult) ;
                          end ;
                      if add
                        then
                          begin
                            dcomp ;

                            reset(add)
                          end
                    ]_multop[ set(mult) ]>_addop[ set(add) ]>[ dcomp ]%

"fixed"
<<identifier[ if unique(vars)
                then
                  insert(vars)
                else
                  error('identifier already declared')
            ]_,> ':' type_;
>%

"variant"
?fixed [ save(casetype) ]
CASE ?casevariable¦identifier[ if not visible(types)
                                 then
                                   error('not a type identifier')
                               else
                                   begin
```

```
                                        type-evaluate;
                                        if ('real' matches type) or
                                            ('file' matches type)
                                          then error('invalid type');
                                        casetype := type
                                    end ] OF


<   <constant[ if type <> casetype
                  then
                    error('invalid caselabel type')
             ]_,> ':' (#fields)_;
 >[ restore(casetype) ]%

"num"
number %

"int"
integer %

"str"
string %


"typeid"
identifier[ if not (visible(types) or std-type)
               then
                  error('not a type identifier' );
            x := contents ]%

"subscript"
 '[' [ amlscope ; son(index) ; intolist
     ]<[ amlscope ; type-evaluate ;
          push(type) ; saveposition
       ]expression[ acomp ; restoreposition
                  ]_,[ amlscope ;
                         if another-iteration
                            then
                               next-iteration
                            else
                               error('too many subscripts')
                       ]> ']'[ amlscope ;
                                 if another-iteration
                                    then
                                       error('too few subscripts') ;
                               father(array) ;
                               son(type) ;
```

```
                                        type-evaluate ] %

"field"
 .identifier[ x := contents ; amlscope ;
                if local(x,vars)
                   then
                      type-evaluate
                   else
                      error(' identifier not declared in record') ]%

"name"
identifier[ reset(nametype) ;
               if visible(vars + constants) or std-const
                  then
                     type-evaluate
                  else
                     if visible(funcs)
                        then
                           begin
                              set(nametype) ;
                              profbld := contents ;
                           end
                        else
                           error('invalid identifier')
           ]?continuation[ push(type) ]%

"subexpression"
[ save(monad);
   save(mult) ;
   save(add)
](expression)[ restore(add) ;
                  restore(mult) ;
                  restore(monad) ]%

"negation"
NOT factor[ if type <> 'boolean'
                  then
                    error('expression type must be boolean') ]%

"setdescription"
'[' elements ']'[ type := 'set' + stacktop ;
                        pop ;
                        push(type) ]%
"relop"
  '<'[ push('<') ]
|'>'[ push('>') ]
|=[ push('=') ]
```

```
¦'>='[ push('>=') ]
¦'<='[ push('<=') ]
¦'<>'[ push('<>') ]
¦'IN'[ push('in') ]%

"casevariable"
identifier[ if unique(vars)
                then
                  insert(vars)
                else error('identifier already declared') ;
          ] ':'%

"continuation"
 reference
¦[ if type <> 'function'
       then
         error('not a subprogram identifier')
 ]arguments[ if visible(profbld, funcprofile)
                then
                  begin
                    son(typeid) ;
                    type-evaluate ;
                  end
                else
                  begin
                    error('illegal parameters') ;
                    type := std-error ;
                  end ] %

"elements"
 elementlist¦null[ push('null') ]%

"elementlist"
[reset(eflag)
]<expression ?range[ if eflag
                          then
                            begin
                              pop(type) ;
                              if type <> stacktop
                                 then
                                   error('incompatible set rangetypes') ;
                              reset(eflag)
                            end ]→[ set(eflag) ]>%

"range"
 '..' expression[ pop(type);
                  if type <> stacktop
```

```
                then
                  error('incompatible set rangetypes') ]%

[ procedure dump-leftovers ;
    begin
      if not empty(unusedlabels)
        then
          begin
            error('labels declared but not used') ;
            dump(unusedlabels) ;
          end ;
     if not empty(fwdtypes)
        then
          begin
            error('pointers without type bodies') ;
            dump(fwdtypes) ;
          end ;
     if not empty(fwdsubs)
       then
          begin
            error('forward subprograms without bodies') ;
            dump(fwdsubs) ;
          end ;
   end ;

procedure get-identifier-value ;
  begin
    saveposition ;
    if visible(scalars)
      then
        type := type + 'scalar' + contents
      else
        if visible(constants)
          then
            evaluate-constant;
    restoreposition ;
  end ;

procedure get-subrange-limits ;
  begin
    son(lowerbound) ;
    son(constant) ;
    evaluate-constant ;
    father(subrange) ;
    son(upperbound) ;
    son(constant) ;
    evaluate-constant ;
```

```
  end ;

procedure add-sign ;
  begin
    if subtree(sign)
      then
        begin
          son(sign) ;
          if contents = '-'
            then
              type := type + '-' ;
          father ;
        end ;
  end  ; .

procedure evaluate-constant ;
  begin
    if rule = 'numeral'
      then
        begin
          add-sign ;
          son(num) ;
          type := type + contents ;
        end ;
    if rule = 'str'
      then
        begin
          type := type + 'char' + contents ;
        end ;
    if rule = 'id'
      then
        begin
          add-sign ;
          son(ident) ;
          get-identifier-value ;
        end ;
    if rule = 'signed-integer'
      then
        begin
          add-sign ;
          son(int) ;
          type := type + contents ;
        end ;
  end ;

procedure string-together-scalars ;
  begin
```

```
      type := type + 'scalar';
      if rule <> 'scalar'
         then
            father(scalar) ;
      intolist ;
      save(anotherelt) ;
      while anotherelt do
         begin
            saveposition ;
            type := type + contents ;
            restoreposition ;
            next-iteration ;
         end ;
      restore(anotherelt);
   end ;

function std-type return flag ;
   begin
      if (contents = 'integer') or (contents = 'boolean') or
          (contents = 'real')      or (contents = 'char')
         then set(std-types)
         else reset(std-types) ;
       std-type := std-types ;
   end ;

function std-const return flag ;
   begin
      if (contents = 'nil') or (contents = 'true') or (contents = 'false')
      or (contents = 'maxint')
         then set(std-consts)
         else reset(std-consts) ;
      std-const := std-consts ;
   end ;

procedure type-evaluate ;
   begin
      type := ' ' ;
      evaluate ;
   end ;
procedure evaluate ;
 begin

    if rule = 'types'
       then
          begin
             son(type) ;
             evaluate ;
```

```
        exit ;
      end ;

  if rule = 'typeid'
     then
       begin
         if visible(types)
            then
              begin
                evaluate ;
                exit ;
              end
            else
             if (contents = 'integer') or (contents = 'boolean') or
                 (contents = 'real')      or (contents = 'char')
               then
                 begin
                   type := type + contents ;
                   exit ;
                 end
               else
                 error(' invalid identifier') ;
         exit ;
       end ;

if (rule = 'name') and (std-const)
  then
     begin
       if contents = 'maxint' then type := type + 'integer' ;
       if contents = 'nil' then type := type + 'ptrnil' ;
       if (contents = 'true') or (contents = 'false') then
          type := type + 'boolean' ;
       exit ;
     end ;

  if (rule = 'variables') or ( rule = 'fixed')
     then
       begin
         son(type) ;
         evaluate ;
         exit ;
       end ;

  if rule = 'type'
     then
       begin
         son ;
```

```
            evaluate ;
            exit ;
         end ;

   if rule = 'pointer'
      then
         begin
            type := type + 'pointer' ;
            son(ptrid) ;
            evaluate ;
            exit ;
         end ;

if rule = 'index'
   then
      begin
         son(rangetype) ;
         son ;
         if rule = 'subrange'
            then
               begin
                  son(lowerbound) ;
                  son(constant) ;
                  evaluate ;
                  exit ;
               end ;
         if rule = 'scalar'
            then
               begin
                  evaluate ;
                  exit ;
               end ;
         if rule = 'rangetype'
            then
               begin
                  if visible(constants) or std-const
                     then
                        begin
                           evaluate ;
                           exit ;
                        end
                     else
                        error('Not a Constant Identifier')
                end
      end ;

if rule = 'file'
```

```
then
     begin
        type := type + 'file';
        son(type) ;
        evaluate ;
        exit ;
     end ;

 if rule = 'set'
    then
       begin
          type := type + 'set' ;
          son(type) ;
          evaluate ;
          exit ;
       end ;

 if rule = 'parameters'
    then
       begin
          son(typeid) ;
          evaluate ;
          exit ;
       end ;

 if rule = 'constant'
    then
       begin
          son ;
          evaluate ;
          exit ;
       end ;

 if rule = 'constants'
    then
       begin
          son(constant) ;
          evaluate ;
          exit ;
       end ;


 if rule = 'numeral'
    then
       begin
          son(num) ;
          evaluate ;
```

```
if rule = 'num'
   then
      begin
         type := type + 'real' ;
         exit ;
      end ;

if (rule = 'integer') or (rule = 'int')
   then
      begin
         type := type + 'integer' ;
         exit;
      end ;

if rule = 'scalar'
   then
      begin
         string-together-scalars ;
         exit ;
      end ;

if rule = 'subrange'
   then
      begin
         son(lowerbound) ;
         son(constant) ;
         evaluate ;
         exit ;
      end ;

if rule = 'array'
   then
      begin
         saveposition ;
         if subtree(packed)
            then
               type := type + 'packed' ;
         type := type + 'array' ;
         son(index) ;
         front(subs) ;
         save(anotherelt) ;
         while not eol(subs) do
            begin
               saveposition ;
              son(rangetype) ;
               son ;
```

```
            if rule = 'subrange'
               then
                  get-subrange-limits ;
            if rule = 'scalar'
               then
                  string-together-scalars;
            if rule = 'rangetype'
               then
                  get-identifier-value ;
            restoreposition ;
            next(subs) ;
         end ;
      restore(anotherelt) ;
      father(array) ;
      son(type) ;
      evaluate ;
      restoreposition ;
      exit ;
   end;

if rule = 'record'
   then
      begin
         saveposition ;
         if subtree(packed)
            then
               type := type + 'packed' ;
         type := type + 'record' ;
         son(fields) ;
         front(vars) ;
         while not eol(vars) do
            begin
               saveposition ;
               son(type) ;
               evaluate ;
               restoreposition ;
               next(vars) ;
            end ;
         restoreposition ;
         exit ;
      end ;

if rule = 'proc'
   then
      begin
         type := 'procedure' ;
         exit ;
```

```
        end ;

  if rule = 'func'
     then
        begin
           saveposition ;
           son(typeid) ;
           evaluate ;
           restoreposition ;
           exit
        end ;

end ]
```

MCOMP-TABLE		
int	+	int
int	-	int
real	+	real
real	-	real

DCOMP-TABLE			
*	=	*	bool
*	<>	*	bool
int	+	int	int
int	-	int	int
int	*	int	int
int	DIV	int	int
int	MOD	int	int
int	<	int	bool
int	>	int	bool
int	<=	int	bool
int	>=	int	bool
int	<>	int	bool
real	+	real	real
real	-	real	real
real	*	real	real
real	/	real	real
real	>	real	bool
real	>=	real	bool
real	<	real	bool
real	<=	real	bool
SET*	+	SET*	SET*
SET*	-	SET*	SET*
SET*	*	SET*	SET*
bool	AND	bool	bool
bool	OR	bool	bool
*	IN	SET*	bool
char	>	char	bool
char	>=	char	bool
char	<	char	bool
char	<=	char	bool

ACOMP-TABLE	
*	*
real	int
ptr*	nil
set*	setnull

APPENDIX III – DIJKSTRA'S LANGUAGE

In this appendix a description of Dijkstra's Language in CFF/AML is presented. The language is presented at a stage before it has gained arrays but after the unusual scoping rules of the language have been introduced.

In this language, a block is similar to ALGOL–60 in that it may have a number of decorations at the start of the block. However, unlike other languages in the ALGOL paradigm, variables in outer scopes are not visible in inner scopes unless explicitly imported into it. Variables must further be imported one scope level at a time only, ie only variables from the most recently enclosing scope may be imported. A variable, besides its type, has one of three characteristics:

GLO A GLO variable is a 'normal' variable, inherited from the outer scope.

PRI A PRI variable is a 'local' variable. It may only be used once it has been initialised, and cannot be inherited by a nested scope as a GLO until it has been intitialised.

VIR A VIR variable is a virginal variable in the sense that it has not been initialised. It cannot be used in the block until it has been initialised, and the block must intitialise the variable. VIR variables can be passed as VIR to nested blocks before the initialisation. Thereafter, they are 'normal' variables and can be passed as GLO variables.

The CFF/AML description of this language is given below. One of the most interesting aspects of the presentation is that the techniques used here can be applied to any language with module–like structures.

```
[DECLARE
 flag monad, dyad ;
 string status ;
]
"program"
[ newscope
  SET     glo, vir, pri, active, vars,
          saveglo, savevir, savepri, saveactive,
          localglo, localvir, localpri, localactive ;
  list newactive ;
]block%

"block"
[ startscope
]BEGIN ?glos ?virs ?pris <statement_;> END [ closescope ] %

"glos"
```

```
GLO <identifier[ if unique(localglos) and visible(active)
                    then
                      begin
                        insert(localactive) ;
                        insert(localglo)
                      end
                    else
                      error('identifier not unique') ]→> ; %

"virs"
VIR <identifier[ if unique(localglos) and unique(active) and
                     (visible(vir) or visible(pri)) then
                    insert(localvir)
                  else
                    error('identifier not unique') ]→> ; %

"pris"
PRI <<identifier[ if unique(localglo+localvir+localpri) then
                      begin
                        insert(localpri) ;
                        insert(vars)
                      end
                    else
                      error('identifier not unique') ]→> : type ;→> %

"type"
INT¦BOOL%

"statement"
  assignment
¦guarded-if
¦guarded-do
¦block
¦skip %

"assignment"
identifier[ status := 'wrong'
            if local(localactive) then status = 'ok'
            if local(localvir)      then
              begin
                status := 'ok'
                insert(newactive) ;
                insert(localactive) ;
                delete(localvir)
              end ;
            if local(localpri) then
              begin
```

```
                    status := 'ok' ;
                    delete(localpri) ;
                    insert(localactive) ;
                  end ;
                if (status = 'ok') and (visible(vars)) then
                  push(type)
                else
                    begin
                      error('cannot resolve identifier reference') ;
                      push(std-error) ;
                    end
              ] ':=' expression [ acomp ] %

"guarded-if"
IF <expression[ if stacktop <> 'boolean'
                      then
                        error('Expression type not Boolean') ;
                    pop
                  ] -> <statement_;>_|> FI %

"guarded-do"
DO <expression[ if stacktop <> 'boolean'
                      then
                        error("Expression Type not Boolean') ;
                    pop
                  ] -> <statement_;>_|> OD %

"skip"
SKIP %

"expression"
[ reset(mult) ;
  reset(dyad) ;
]?sign<<factor[ if monad
                      then
                        begin
                          mcomp ;
                          reset(monad)
                        end ;
                    if dyad
                      then
                        begin
                          dcomp ;
                          reset(dyad)
                        end ;
                  ]_multop[ set(dyad)
                          ]>_addop[ set(dyad)
```

```
                                        ]> ?predicate %

"predicate"
relop ?sign<<factor[ if monad
                                    then
                                      begin
                                        mcomp ;
                                        reset(monad)
                                      end ;
                                    if dyad
                                      then
                                        begin
                                          dcomp ;
                                          reset(dyad)
                                        end ;
                                ]_multop[ set(dyad)
                                        ]>_addop[ set(dyad)
                                                ]> [ dcomp
                                                   ]%

"factor"
identifier[ if local(localactive) and visible(vars)
                        then
                          begin
                            push(type)
                          end
                        else
                          begin
                            push(std-error);
                            error("Identifier not declared') ;
                          end ]
|integer [ push('integer') ]
|TRUE     [ push('boolean') ]
|FALSE    [ push('boolean') ]
|([ save(monad) ;
     save(dyad) ; ]expression[ restore(dyad) ;
                                        restore(monad) ])%

"sign"
  + [ push('+') ; set(monad) ]
|- [ push('-') ; set(monad) ] %

"addop"
  + [ push('+') ]
|- [ push('+') ]
|OR[ push('OR') ]%
```

```
"multop"
  * [ push('*') ]
|/ [ push('/') ]
|AND[ push('AND') ]%

"relop"
  '<'  [ push('<')   ]
|'>'  [ push('>')   ]
|'<=' [ push('<=')  ]
|'>=' [ push('>=')  ]
|'='  [ push('=')   ]
|'<>' [ push('<>')  ]

[
function type return string ;

   begin
      son(type) ;
      if contents = 'INT'
         then
            type := 'int'
         else
            type := 'bool'
   end ;

procedure startscope ;

   begin
      clear(saveglo) ;
      clear(savevir) ;
      clear(savepri) ;
      copy(glo, saveglo) ;
      copy(vir, savevir) ;
      copy(pri, savepri) ;
      clear(glo) ;
      clear(vir) ;
      clear(pri) ;
      clear(active) ;
      copy(localglo, glo) ;
      copy(localvir, vir) ;
      copy(localpri, pri) ;
      copy(localactive, active) ;

      newscope ;

      SET      localglo, localvir, localpri, localactive ,
               saveglo, savevir, savepri, saveactive ;
```

```
   end ;

procedure closescope ;

begin
   if not empty(localpri) then
      begin
         error('The following pri'''s were not initialised') ;
         dump(localpri)
      end ;

   if not empty(localvir) then
      begin
         error('The following vir'''s were not initialised') ;
         dump(localvir)
      end ;

   endscope ;

   front(newactive) ;
   while  not eol(newactive) do
      begin
         if local(localvir) then
            begin
               delete(localvir) ;
               insert(localactive) ;
            end
         else if local(localpri) then
            begin
               delete(localpri) ;
               delete(newact) ;
               insert(localactive) ;
            end ;
         next(newactive) ;
      end ;

   clear(glo) ;
   clear(vir) ;
   clear(pri) ;
   copy(saveglo, glo) ;
   copy(savepri, pri) ;
   copy(savevir, vir) ;

end ;
]
```

MCOMP-TABLE		
int	+	int
int	-	int

DCOMP-TABLE			
int	+	int	int
int	-	int	int
int	*	int	int
int	/	int	int
int	<	int	bool
int	>	int	bool
int	<=	int	bool
int	>=	int	bool
int	=	int	bool
int	<>	int	bool
bool	AND	bool	bool
bool	OR	bool	bool

ACOMP-TABLE	
int	int
bool	bool

APPENDIX IV – CLU

This appendix contains a CFF/AML specification of the syntax and context conditions of the programming language CLU.

CLU is included here as another example of a 'real–life' language specification, with context conditions more complex than those for Pascal. For simplicity, parameterised (or generic) clusters and multiple assignments have been omitted from the language given here. Some of the most trivial AML procedures and some of the obvious decorations are omitted and replaced by comments. The specification should be sufficiently complete that the reader can get the feel of the approach taken.

The expression and primary rules are once again similar to the many which have been included in the text of the thesis and the appendix showing Pascal. The decorations on these rules are therefore omitted.

The CFF/AML specification for CLU is:

```
[ DECLARE
    string stackname, type ;
    flag   exponent, monad, dyad, infunc, initer, cansignal,
           iscluster ;
]
"program"
[beginscope] module [endscope] %

"module"
 ?<equate_> procedure
|?<equate_> iterator
|?<equate_> cluster %

"procedure"
[save(infunc) ; save(cansignal) ;
 reset(infunc); reset(cansignal) ;
]identifier[ savename ;
             if isunique
               then insert(procs)
           ] = PROC [ beginscope
                    ] args ?returns ?signals ;
              routine-body
              END identifier[checkname ];
                             endscope
                             restore(infunc) ;
                             restore(cansignal) ] ; %

"iterator"
```

```
[save(initer) ; save(cansignal) ;
 reset(initer); reset(cansignal)
]identifier[ savename ;
                if isunique
                   then insert(iters)
             ] = ITER [ beginscope
                      ] args ?yields ?signals ;
                routine-body
                END identifier[checkname ;
                                  endscope
                                  restore(initer) ;] ; %

"cluster"
[ set(iscluster)
]identifier[ savename ;
                if isunique
                   then insert(clusters)
             ] = CLUSTER [ beginscope
                         ] IS <identifier␣,> ;
                cluster-body
                END identifier[ checkname ;
                                  endscope ] ; %

"args"
( <<identifier[ if isunique then
                      begin
                         insert(args) ;
                         insert(vars)
                ]␣,> : type-spec␣,> ) %

"decl"
<identifier[ if isunique
                   then insert(vars)
              ]␣,> : type-spec %

"returns"
[set(infunc)
]RETURNS ( <type-spec␣,> ) %

"yields"
[set(initer)
]YIELDS ( <type-spec␣,> ) %

"signals"
[set(cansignal)
]SIGNALS ( <exception␣,> ) %
```

```
"exception"
identifier[if unique(signals)
                then
                   insert(signals)
                else
                   error('Not a unique exception')
           ] ?type-part %

"type-part"
( <type-spec_,> ) %

"constant"
 expression
|type-spec %

"routine-body"
?<equate_>
?<own-var_>
?<statement_> %

"cluster-body"
?<equate_> REP = type-spec ; ?<equate_>
?<own-var_>
<routine_> %

"routine"
 procedure
|iterator %

"equate"
identifier[ if isunique then
                insert(consts) ] = constant %

"own-var"
 OWN decl ;
|OWN identifier[ if isunique
                      then insert(vars)
                 ] : type-spec := expression %
|OWN <decl_,> := invocation %

"type-spec"
 NULL
|BOOL
|INT
|REAL
|CHAR
|STRING
```

```
|ANY
|REP
|CVT
|array
|sequence
|record
|struct
|oneof
|variant
|proctype
|itertype
|typeid %

"typeid"
identifier [ if not visible(clusters) then
                error('not a type abstraction') ] %

"array"
ARRAY '[' type-spec ']'

"sequence"
SEQUENCE '[' type-spec ']'

"record"
RECORD '[' [ localscope
          ]<field-spec,> [endscope] ']'

"struct"
STRUCT '[' [ localscope
          ]<field-spec,> [endscope] ']'

"oneof"
ONEOF '[' [ localscope
          ]<field-spec,> [endscope] ']'

"variant"
VARIANT '[' [ localscope
            ]<field-spec,> [ endscope] ']'

"proctype"
PROCTYPE ( ?<type-spec,> ) ?returns ?signals %

"itertype"
ITERTYPE ( ?<type-spec,> ) ?yields ?signals %

"field-spec"
<identifier[ if unique(vars)
```

```
                then
                   insert(vars)
                else
                   error('Non-unique tag')
             ]→> : type-spec %

"statement"
  decl
¦init-decl
¦invoc-decl
¦assign
¦invoc-assign
¦name-assign
¦array-assign
¦procedure-invocation
¦while
¦for-id
¦for-decl
¦if
¦tagcase
¦return
¦yield
¦signal
¦exit
¦BREAK
¦CONTINUE
¦block
¦RESIGNAL
¦except %

"procedure-invocation"
primary [if not 'proctype' matches type then
            error('not a procedure invocation')
         else
            begin
               son(args) ;
               front(args) ;
               saveposition ;
               type-evaluate ;
               push(type) ;
            end] ( ?<expression[ acomp;
                                    nextiter
                                ]→> [handle-iter-overflow
                                     ] ) %

"init-decl"
identifier[if isunique
```

```
                then
                   insert(vars)
             ] : type-spec[ son(type-spec) ;
                                type-evaluate ;
                                push(type) ;
                             ] := expression [acomp] %

"assign"
identifier[ if visible(vars)
                then
                   begin
                      type-evaluate ;
                      push(type) ;
                   end
          ' ] := expression [acomp] %

"name-assign"
primary[ if not (('record' matches type) or
                      ('struct' matches type) or
                      ('oneof'  matches type) or
                      ('variant' matches type))
            then error('must be a record type) ;
         son(field-spec)
       ].identifier[ if not local(vars)
                        then
                           error('not a record component'
                        else  begin
                           type-evaluate ;
                           push(type) ;
                        end
                     ] := expression [acomp] %

"array-assign"
primary[ if not ('array' matches type)
            then error('must be an array type)
         get-final-array-type ;
         push(type) ;
         get-subscript-type;
         push(type) ;
       ]
 '[' expression[acomp] ']' := expression[acomp]  %

"while"
WHILE expression[ checkbool
                ] DO [beginscope
                      ] body END [endscope]%
```

```
"for-decl"
[checkiter
]FOR [ beginscope
     ]?<<identifier[ if isunique
                        then insert(vars)
                   ]→> : type-spec[ son(type-spec) ;
                                    type-evaluate ;
                                    push(type) ;
                                    acomp ;
                                    nextiter
                                ]→>[check-iter-underflow
                                    ] IN id DO body END [endscope
                                                        ]%

"for-id"
[checkiter
]FOR [ beginscope
     ]?<identifier[ if not visible(vars)
                      then error('illegal identifier')
                      else begin
                        type-evaluate ;
                        push(type) ;
                        acomp ;
                        nextiter
                   ]→>[check-iter-underflow
                      ] IN id DO body END [endscope] %

"id"
identifier %

"if"
IF expression[ checkbool
             ] THEN [beginscope
                    ] body [endscope]
  ?<ELSEIF expression[ checkbool
                      ] THEN [ beginscope
                             ] body[ endscope ]_>
  ?elsepart
END %

"tagcase"
TAGCASE expression[if not (('oneof' matches stacktop)
                           and ('variant' matches stacktop)
                     then error('invalid type for case') ;
                   pop
                   amlscope ;
                   son(field-spec)   ]
```

```
   <tag-arm_>
   ?otherspart
END %

"return"
RETURN ?returnlist %

"returnlist"
[ if not isfunc then
      error('cannot return values')
   else begin
      father(procedure) ;
      son(returns) ;
      intolist ;
      saveposition ;
      son(type-spec) ;
      type-evaluate ;
      push(type) ;
   end
]( <expression[ acomp ;
                     restoreposition ;
                     if another-iteration then
                        begin
                           next-iteration ;
                           saveposition ;
                           son(type-spec) ;
                           type-evaluate ;
                           push(type) ;
                        end
                     else
                        error('too many return values')
                  ]->[ restoreposition ;
                        if another-iteration
                           then
                              error('too few return values')] ) %

"yield"
YIELD ?yieldlist %

"yieldlist"
[ if not isiter then
      error('cannot yield values')
   else begin
      father(procedure) ;
      son(yields) ;
      intolist ;
      saveposition ;
```

```
      son(type-spec) ;
      type-evaluate ;
      push(type) ;
   end
]( <expression[ acomp ;
                    restoreposition ;
                    if another-iteration then
                       begin
                          next-iteration ;
                          saveposition ;
                          son(type-spec) ;
                          type-evaluate ;
                          push(type) ;
                       end
                    else
                       error('too many yield values')
                 ]→>[ restoreposition ;
                       if another-iteration
                          then
                             error('too few yield values')] ) %


"signal"
SIGNAL identifier [ if not cansignal then
                       error('illegal signal clause)
                    else begin
                       if visible(signals)
                          then begin
                             son(exception) ;
                             intolist ;
                             saveposition ;
                             son(type-spec) ;
                             type-evaluate ;
                             push(type) ;
                           end
                          else
                             error('invalid exception name')
                      end ] ?signallist %

"signallist"
( <expression[ acomp ;
                    restoreposition ;
                    if another-iteration then
                       begin
                          next-iteration ;
                          saveposition ;
                          son(type-spec) ;
```

```
                    type-evaluate ;
                    push(type) ;
                  end
               else
                  error('too many signal values')
            ]→>[ restoreposition ;
                  if another-iteration
                    then
                      error('too few signal values')] ) %

"exit"
EXIT identifier [ if not cansignal then
                    error('illegal signal clause)
                  else begin
                    if visible(signals)
                      then begin
                        son(exception) ;
                        intolist ;
                        saveposition ;
                        son(type-spec) ;
                        type-evaluate ;
                        push(type) ;
                       end
                      else
                        error('invalid exception name')
                  end ] ?signallist %

"block"
[ beginscope
]BEGIN body END [ endscope ]%

"resignal"
statement [ if not cansignal then
              error('cannot perform a resignal')
          ] RESIGNAL <identifier[ if unique(signals) then
                                    error('unknown exception')
                                  ]-> %

"except"
statement [ if not cansignal) then
              error('no exceptions to handle')
          ] EXCEPT ?<when-handler_>
                ?others-handler
                END %

"tag-arm"
TAG [ beginscope
```

```
    ] <identifier[ amlscope ;
                        if not local(vars) then
                           error('not a component of the type')
                     ]_> ?declpart : body [endscope]%

"declpart"
( identifier[ if isunique then
                     insert(vars) ] : type-spec ) %

"when-handler"
 WHEN <identifier[ if unique(signals) then
                           error('illegal exception')
                     ]_> [beginscope
                           ] ?declslist : body [endscope]%
|WHEN <identifier[ if unique(signals) then
                           error('illegal exception')
                     ]_> (*) : [ beginscope
                           ] body [endscope]%

"declslist"
( <decl_> ) %

"others-handler"
OTHERS ?declpart : body %

"body"
?<equate_> ?<statement_> %

{ The code for expression decoration is similar to that for Pascal,
  with the addition of a third layer of priority.  We therefore
  do not include it here }

"expression"
?sign <<<<primary_expop>_multop>_addop> ?predicate %

"predicate"
relop ?sign <<<<primary_expop>_multop>_addop> %

"primary"
 NIL
|TRUE
|FALSE
|integer
|number
|string
|identifier
|identifier '[' <constant_> ']'
```

```
¦primary.name
¦primary '[' expression ']'
¦invocation
¦FORCE   '[' type-spec ']'
¦UP (expression)
¦DOWN (expression)
¦(expression)
¦ ~ primary %

"elsepart"
ELSE [beginscope] body [endscope]%

"otherspart"
OTHERS : [beginscope] body [endscope] %

"expop"
** %

"multop"
  //
¦/
¦*
¦
¦&
¦CAND %

"addop"
   '||'
   ||
¦+
¦-
¦'|'
¦ |
¦COR %

"relop"
  '<'
¦'>'
¦'<='
¦=
¦'>='
¦'~<'
¦'~<='
¦~='
¦'>='
¦'~>'%

[
procedure beginscope ;
   begin
```

```
    newscope ;
    set      types, vars, consts, procs, clusters,
             iters ;
    list     args, signals, returns, ;
  end ;

function isunique return flag ;
  begin
    if visible(types+vars+consts+procs+clusters+iters)
       then
          begin
             error('Redeclared identifier');
             reset(isunique) ;
          end
       else
           set(isunique)
  end ;

procedure savename ;
  begin
    stackname := contents ;
    save(stackname) ;
  end ;

procedure checkname ;
  begin
    restore(stackname) ;
    if stackname <> contents then
       error ('Identifiers do not match') ;
  end

procedure checkbool
  begin
    if stacktop <> 'boolean'
       then
          error('not a boolean expression');
    pop
  end

procedure checkiter ;
  begin
    son(id) ;
    if visible(iters) then
       begin
          son(args) ;
          front(args) ;
          saveposition ;
```

```
        end
      else error('not a valid iterator')
   end

procedure nextiter ;
    begin
       restoreposition ;
       if not eol(args) then
          begin
             next(args) ;
             saveposition ;
             type-evaluate ;
             push(type) ;
          end
       else
          error('iterator parameter overflow') ;
   end ;

procedure check-iter-underflow ;
   begin
      restoreposition ;
      if not eol(args) then
         error('Too few iterator parameters')
   end ;

procedure type-evaluate ;
   begin
      type := '' ;
      evaluate ;
   end

procedure evaluate ;
   begin
      if rule = 'type-spec'
         then
            begin
               IF contents = 'NULL'   then type := 'null'   else
               IF contents = 'BOOL'   then type := type+'bool'   else
               IF contents = 'REAL'   then type := type+'real'   else
               IF contents = 'INT'    then type := type+'int'    else
               IF contents = 'CHAR'   then type := type+'char'   else
               IF contents = 'STRING' then type := type+'string'else
               IF contents = 'ANY'    then type := type+'any'    else
               IF contents = 'REP'    then
                  begin
                     if iscluster
                        then
```

```
            begin
               father(cluster-body) ;
               son(type-spec) ;
               evaluate ;
               exit ;
            end
         else begin
            error('cannot have rep except in a cluster') ;
            type := std-error ;
            exit ;
         end
   else begin
      son ;
      if rule='array' then
         begin
            type := type+'array';
            son(type-spec) ;
            evaluate ;
            exit ;
         end ;

      if rule='sequence' then
         begin
            type := type+'sequence'
            son(type-spec) ;
            evaluate ;
            exit ;
         end ;

      { similarly for struct, record, oneof, variant except
        that the son(type-spec) becomes son(field-spec) }

        if rule = 'typeid' and visible(clusters)
           then begin
              type := type + contents ;
              exit ;
           end ;

        if rule = 'field-spec' then
           { string together the types of the
             components, as in a Pascal record }


        if rule='proctype' then
           begin
              type := type + 'proctype' ;
              saveposition ;
```

```
                son(type-spec) ;
                ¦ string together argument types using
                  the args list and the same strategy
                  as for records ¦
                restoreposition
                if subtree(returns) then
                  ¦ add in the return types ¦
              end ;

            if rule = 'itertype'
              then
                ¦ similar strategy as for proctypes ¦

      ¦ finally a catch-all as all the other alternatives have a
        similar structure ¦

            son(type-spec) ;
            evaluate ;
            exit ;
        end
  end
]
```

The type evaluation tables restricts the types to be compatible with themselves only, or the type ANY. We therefore do not bother to give the tables here.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-85-1232	2.	3. Recipient's Accession No.

4. Title and Subtitle

Specification and Verification of Context Conditions for Programming Languages

5. Report Date: Nov. 1, 1985

6.

7. Author(s)

Simon M. Kaplan

8. Performing Organization Rept. No. R-85-1232

9. Performing Organization Name and Address

Department of Computer Science, University of Illinois
1304 W. Springfield Ave., 240 Digital Computer Lab
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address

13. Type of Report & Period Covered

Thesis

14.

15. Supplementary Notes

16. Abstracts

We introduce CFF/AML, a metalanguage that defines syntax and context conditions of programming languages to a programming metatool such as a compiler generator. CFF is the context free part, and AML the context condition specification part of the metalanguage. Other metalanguages to perform this function exist. However, CFF/AML is particularly useful to language designers and compiler constructors because it provides a natural and straightforward way of describing a programming language, thus decreasing the overheads involved in the specification of languages.

A number of examples of language specifications are given, including full language specifications for languages such as Pascal and CLU.

CFF/AML is formally defined using a temporal proof system. This allows us the ability to reason about the correctness of specification. A number of example proofs of specification correctness are given.

17. Key Words and Document Analysis. 17a. Descriptors

Compiler Generators
Metatools
Static Semantics
Context Conditions
Context Sensitive Languages
Metalanguages
Temporal Proof Systems
Verification of Specifications

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
unlimited	UNCLASSIFIED	275
	20. Security Class (This Page) UNCLASSIFIED	22. Price