Type Checking in VIMVAL

by

Bradley C. Kuszmaul

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

at the

Massachusetts Institute of Technology

June 1984

Signature of Author . . . .

Department of Electrical Engineering and Computer Science

10 May 1984

Certified by . . . .

Jack B. Dennis

Thesis Supervisor

Accepted by . . . .

John Guttag

Chairman, Departmental Committee

Type Checking in VIMVAL

by

Bradley C. Kuszmaul

Submitted to the Department of Electrical Engineering and Computer Science on 20 June 1984 in partial fulfillment of the requirements for the Degree of Bachelor of Science.

Abstract

A type system is developed for the revised version of the VAL programming language (VIMVAL) which has the following features:

1. Type Inference: allows programs to be written with incomplete type specifications. The type checker infers the types of the expressions from their context.

2. Polymorphism: allows modules to be written which operate on more than one type, performing analogous operations on different types of data.

3. Higher order functions: functions are first class data in VIMVAL.

4. Recursive types: a type may refer to itself.

A theory of types is developed which applies to a large class of programming languages, including VIMVAL. First the notion of *type* is defined, then the interaction between *types* and *programs* is described, with a definition of *type correctness. Type correctness* is shown to be well defined and decidable, and a type checking algorithm is given which performs type checking for VIMVAL.

Thesis Supervisor: Jack B. Dennis
Title: Professor of Computer Science and Engineering
Keywords: Polymorphism, Static Type Checking, VIMVAL, VAL, Finite State Automata, Type Inference.

Acknowledgments

I would like to thank the following for their contribution to this work:

Jack Dennis, my thesis advisor, for providing a good topic, and constructive criticism of my work.

Bhaskar Guharoy for proofreading and discussing this work.

Gary Leavens for proofreading and offering valuable suggestions.

Lyman Hurd for proofreading.

Eric Anderson for never actually telling me he did not want to hear any more about my thesis: he probably wanted to.

Kimberly K. Lewis for listening anyway.

To Kimberly, who likes being silly with me.

Table of Contents

Table of Figures

Chapter One

Introduction

VAL (Value-Oriented Algorithmic Language), developed by Ackerman and Dennis, of M.I.T.'s Computation Structures Group [1], explored static data flow architecture [5] for a side-effect free language. Side-effect free languages implement functions which, when given a particular set of arguments, always return the same result (as opposed to languages which allow side effects, and the result of calling a function depends on the state of the environment as well as the explicit arguments). Such languages are sometimes called "functional" because they implement mathematical functions. Functional languages are well suited to highly parallel computers because changing the order in which different parts of a program are run (or running them in parallel) does not change the semantics of the program [7, 3]. The Computation Structures Group is now developing a new implementation of a revised VAL, based on an abstract data flow machine called the *VAL Interpretive Machine* (VIM), which executes data flow instructions directly. The revised version of VAL is called VIMVAL. The original VAL does not support polymorphism, recursive data types, recursive functions, higher order functions, or type inference. Because it was not expected that the static architecture would implement proper function application using data flow, VAL function calls are actually implemented by compile time "macro expansion", precluding higher order functions in general, and recursive functions in particular. VAL is a strongly typed language which requires that the type of every variable and formal argument be completely and explicitly specified. The VIM abstract machine includes mechanisms for function application, and the Computation Structures Group is developing an implementation of VIMVAL. Since higher order functions introduce extra complexity, we decided to rework the type system for VIMVAL. Several desired features for the type scheme of VIMVAL were proposed, most of which boiled down to: ease of use for the programmer. Ease of use has at least two components: "writeability" and "readability": it is easier to write programs (at least it involves fewer characters to write a program) in a language which requires a minimum of symbols, while it is typically easier to read programs written in a language which requires the programmer to add redundant information to a program. Thus "Ease of use" has different meanings for different people. Here is a set of criteria for evaluating the ease of use of a type system.

- The type rules must be easy to remember, and express: they should be simple and consistent.

- The programmer should not be required to write a lot of extra symbols just to facilitate type checking. "A lot" is subjective: Some programmers like to explicitly specify types, and some programmers find that requiring such type specification hinders them.

- The language should be strongly typed, so that no type errors can occur at run time, and so that no type information needs to be represented at run time.

To meet these goals, we have decided to incorporate *type inference* into VIMVAL. Type inference allows the programmer to write a program with a minimum of type declarations. Most types can be deduced from their context, for example the type of the constant **3.1415** must be **REAL** in VIMVAL, and multiplication of a **REAL** value by some variable x would mean that x must also be **REAL**. The VIMVAL compiler automatically determines the type of every expression, or gives an error saying that some expressions are ambiguously typed (i.e. expressions which have more than one possible type), or overconstrained (i.e. expressions which have no possible type). The type checking algorithm guarantees that no type errors will occur at run time. We adopt the strategy that the programmer should be required to write a minimum number of extra symbols to facilitate type checking, while allowing a programmer to optionally add extra type information to a program. We will discuss how well our type inference system meets our goals in the conclusion of this paper.

VIMVAL has the following additional features which improve the expressive power of the language, while adding some new difficulties to type inference that have not been covered by [16, 15, 14].

Polymorphism

Allows programmers to write functions which perform analogous operations on different types of data. One example of a built in polymorphic function is ARRAY – LIMH, which maps from any array to an integer. Polymorphism and type inference are loosely coupled in VIMVAL because we allow any type to be explicitly written, thus we need a way to denote polymorphic types. The main restriction on polymorphism is that a formal argument to a function can not be used polymorphically, only free variables can be used polymorphically.

Recursive data types

Recursive types are allowed. In fact any type that can be written is allowed. Recursive types are not the same as recursive data. It is not possible to construct a recursive data object in VIMVAL because VIMVAL requires that all data objects be "semantically" constructed after their components are constructed. (There are two "exceptions" to this rule. It is possible for a function to operate on a copy of itself, but the circularity involved is very stylized, and the functions are not actually being constructed with self-references. VIMVAL has "early completion structures" [4], which have certain advantages which do not effect the fact that recursive data can not be built in VIMVAL.)

Higher order and recursive functions

Functions are first class data in VIMVAL: functions can be passed to and returned from functions, and functions can be used as parts of structures. Recursive functions are a special case of higher order functions. All recursive functions are defined to have the same semantics as a program written with explicit function arguments to replace recursion. (In a language with higher order functions, explicitly specifying the type of a function can be troublesome. See [10] for a discussion of this.)

Previous Work

Semantics of Types and Type Checking

Much work has been done recently on types. Scott [21] and McCracken [14] view types as retracts of the universal domain (e.g. special functions on the set of all objects which can be represented using strings of bits). Milner [16] views types as ideals (which is a special set of objects meeting certain closure conditions). Donahue [6] and Demers [2] claim that types are sets of operations (as opposed to sets of objects). This approach is contrasted with the algebraic approach, where any particular type is specified by its algebraic properties. We unify some of these views, and following Solomon [22], we see types as sets of objects with certain restrictions.

Type Inference, Polymorphism and Undecidability

Langmack [8] showed that two of VIMVAL's features, type inference and polymorphism, can combine to make the type correctness of a program an undecidable problem. Langmack showed that by either requiring all formal arguments to be "monomorphic" (i.e. the arguments must have exactly one type), or requiring all formal arguments to be explicitly typed, the undecidability can be avoided. Our solution to this problem is to require all formal arguments to have exactly one type, i.e. formals must be "monomorphic" [16] at run-time. This rules out certain programs, but we believe, with the support of Milner [16], that most useful programs have the property that all their formals are monomorphic anyway.

Type Inference Algorithms

Solomon [22] implicitly described a type checking algorithm for certain kinds of languages, where types can be described by regular sets, and the type declarations are complete and explicit. This thesis will extend Solomon's work to embrace type inference. (Also relevant is the work on type equivalence for types in Algol68 [20], which uses finite state machines to perform comparisons of types, but we are not directly concerned with such comparisions.)

Peacock [19] designed a type checking algorithm for VIMVAL based on constraint propagation through a graph representing a VIMVAL program. As Peacock pointed out, his algorithm was driven by side effects (which is not aesthetically pleasing to a group working on a purely applicative language such as VIMVAL), lacked a correctness proof, and was not implemented. This thesis corrects and extends Peacock's work by presenting a type checking algorithm, proving it correct, and supplying an implementation of the algorithm.

Overview

Our work involves *type inference*, and we argue that the sets of objects that are of a given type are in one to one correspondence with the sets of operations that define a type. We note an isomorphism between sets of restrictions and certain sets of objects: A given set of restrictions completely and uniquely describes a type, and a type completely and uniquely describes certain sets of objects. We go on to use that isomorphism between the restrictions and our intuitive understanding of types, to define types, because the restrictions are easy to formalize. The types then have certain algebraic properties (those of regular sets) which are dependent on the restrictions placed on them by a programming language.

We are interested in applying the algebraic properties of types directly to implement the type checker, falling closer to Milner [16] and Scott [21] who are modeling *type checking*, than the algebraists who are modeling the *type objects*.

Synopsis

Chapter 2 defines *type* in terms of regular sets and finite state automata: *types* are regular sets with a certain decidable property. Chapter 3 describes the interaction between *types* and *programs*, defining *type assignments*. Chapter 3 goes on to define *type-correctness* in terms of the number of possible type assignments, and shows that *type-correctness* is well defined, and decidable, and that the type assignment for a given program is computable. Chapter 4 describes the application of our type checking system to VIMVAL. In conclusion we will examine the type system in VIMVAL, and compare it with our ease-of-use goals.

Chapter Two

Types

The goal of this chapter is to define the notion of *type* rigorously. We discuss our intuitive notions of types, and how well they fit some currently available programming languages. Then using examples from a dialect of LISP, we motivate several definitions, which lead to a definition of *type-systems* and *types*, which formalize our intuition. A *type* is a description of a set of objects, which have a certain property (the *type* of the objects). The description can be written as a *regular expression*, thus *types* are isomorphic to *regular sets.*

2.1 A Discussion of Type Checking

Types are easy to use, but difficult to describe. Intuitively, type checking is something which can catch certain programming errors (type errors), such as adding an integer to a string, or using an array as if it were a function. Many LISP implementations provide *run time* type checking, which detects type errors when they happen. This approach is not robust because it is difficult to determine when all the type errors in a program have been removed. Another approach, which we take, is that programs are checked statically for type correctness. In order to perform such static type checking, we traditionally have to put up with a loss of notational convenience: we may have to add extra symbols to a program to help the type checker, or the extra restrictions required for static type checking might mean that we are not be able to express a program in the way we want to. Another possibility is that the type checking system might not find all type errors (e.g. the *lint* program on UNIX does some type checking on C programs, but it is not guaranteed to find all type errors.) It is difficult to "retrofit" a programming language with static type checking because it is often impossible to perform complete static type checking. (In LISP the property that **cdr** of **nil** is never taken can not be statically checked, and in C it is not possible to statically check that a pointer value actually points to a valid address.)

Our type theory will follow Leivant [9] and Solomon [22], who model types as structural conditions on data objects: given a data object *O*, and a type *T*, it is possible to decide whether *O* is of type *T* by examining the structure of *O*. This approach means that types are sets of objects. In this case, *T* is a description of the possible "shapes" of *O*. We specifically follow Solomon, and claim that *T* describes a regular set of paths, where a path is a sequence of symbols in some alphabet (called the *selectors*) which corresponds to a legal sequence of operations on object *O*. This approach means that types are isomorphic to regular sets, and everything we want to know about a type can be rephrased in terms of regular sets.

2.2 The Properties of *Types*

Our goal is to define *type* rigorously. In order to do this we need to deal with some of the restrictions that we intuitively associate with types (for example no object is both an **integer** and a **real**, and **arrays** have a "subtype", but **integers** do not.) First we will describe *selectors*, then *paths*. Then we will discuss the restrictions, leading to the definition of a *type-system*. Finally we will define *type*.

We will use LISP examples in this chapter, even though the types of LISP do not necessarily match the types of VIMVAL. We use the words "path" and "selector" informally to motivate our definitions, which appear below. The dialect of LISP that our examples will use has two base types:

Integers	The only selector for an integer is *INT*. There is only one path from an integer, and that is $\langle INT \rangle$.
Cons cells	Cons cells have a CAR and a CDR, so the selectors for a cons cell are *CAR* or *CDR*. All paths from a cons cell start with *CAR* or *CDR*. Cons cells can be built with the CONS function. There is a special cons-cell called NIL, which has CAR and CDR both NIL. The LIST operator builds a list of cons cells in the standard way, ending with a NIL. For example:

$$\text{(LIST X Y Z)} \equiv_{def} \text{(CONS X (CONS Y (CONS Z NIL)))}$$

We will be a little sloppy with the type of NIL in our examples, because NIL is a "polymorphic" value (it could be an empty LIST of anything), and we have not developed the tools to discuss NIL's type.

Paths for LISP are sequences with elements in $\{INT, CAR, CDR\}$. This set is called the set of *selectors* for LISP.

Notation: The set of *selectors* for a program, denoted Σ, is some finite set, which is dependent on the program being type checked.

Elements of Σ will be written in uppercase italics, e.g. *INT* and *CAR*.

Notation: A *path* is a sequence, with each element of the sequence in Σ. Paths are possibly infinitely long.

The length of a path x is denoted $|x|$.

If x is a path with $|x| \geq i$, then x_i is the *ith* element of x. The first element of x is x_1.

We write finite paths with angle brackets: $x=\langle INT, CAR\rangle$ is a path with $x_1=INT$ and $x_2=CAR$. The symbol $\langle\rangle$ denotes the path of length zero (the empty path).

Paths can be concatenated: if x and y are paths, then $z=x\circ y$ is a path, where if x is infinite then $z=x$, otherwise $z_i=x_i$ for $i \in \{1,...,|x|\}$, and $z_{|x|+i}=y_i$ for all finite $i \in \{1,...,|y|\}$.

The words *tuple*, and *string*, are often used for things which are similar to *paths*, but typically tuples and strings are finite in length.

Consider the LISP value, *O*, generated by

(CONS 1 2).

Here, *O* is a cons cell containing an integer in both its car and its cdr, the set of paths for *O* is { <*CAR*, *INT*>, <*CDR*, *INT*> }, and this set defines the "type" of *O* (see Figure 2-1).

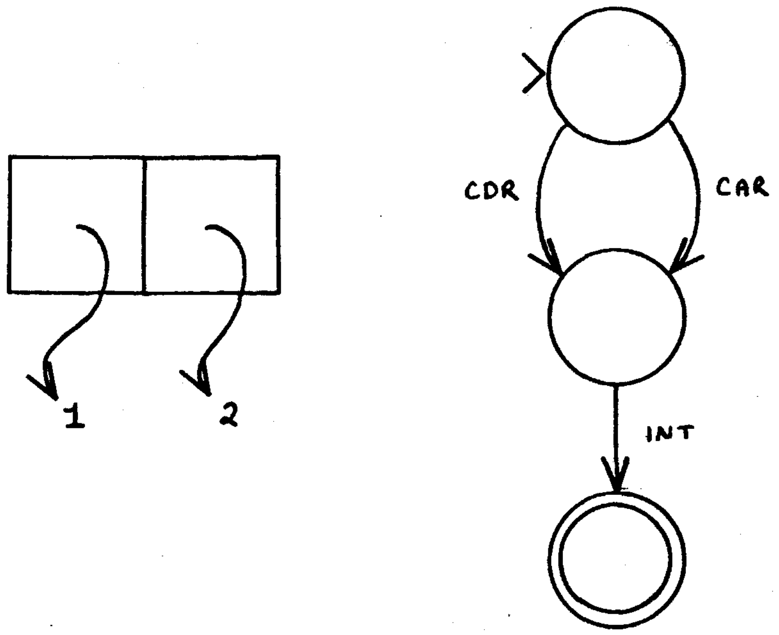


Figure 2-1:(CONS 1 2) Cell, with paths: { <*CAR*, *INT*>, <*CDR*, *INT*> }

The previous example describes a type which is a finite set of finite paths. The next example illustrates a type which is an infinite set of infinitely long paths. Consider the type equation T = CONS[T,T]. The paths for this type are infinitely long, and consist of any sequence of *CARs* and *CDRs*.

For the next example (which will give an example of a type which is an infinite set, all the elements of which are finite except for one) we need a few standard definitions, adapted from [11]. We will also need the following definitions to define *type.*

Definition 2-1: If A and B are sets of paths, then the *composition* of A and B, is

$$A \circ B \equiv_{def} \{ \sigma \circ \rho \mid \sigma \in A, \rho \in B \},$$

where $\sigma \circ \rho$ is the concatenation of path σ and path ρ.

The definition of concatenation of paths automatically takes care of the case where some of the elements of A or B are infinite.

We want to compose i copies of a set of paths, A, where i can be a finite integer, or it can be ∞. The case of a finite integer is adapted directly from [11], while we need an extra definition to define the case of i infinite.

Definition 2-2: If A is a set of paths, and i is a finite integer, then A^i is defined recursively:

- $A^0 \equiv_{def} \{ \langle\rangle \}$ (i.e. the empty path, not $\emptyset$)
- $A^i (i>0) \equiv_{def} A \circ A^{i-1}$

Definition 2-3: A path σ is an *initial segment* of a path γ if there is some path ρ, such that $\sigma \circ \rho = \gamma$.

Definition 2-4: If A is a set of paths, then

$$A^{\infty} \equiv_{def} \{ \sigma \mid \forall j<\infty, \exists \rho \in A^j, \text{such that } \rho \text{ is an initial segment of } \sigma \}.$$

If A is a set of paths, then A^i is the set of paths which are made by concatenating i elements of A together. A^{∞} is the set of paths which are made by concatenating an infinite number of elements of A together.

Definition 2-5: The *Kleene star* operator on sets, written *, denotes the operation

$$A^* \equiv_{def} \cup_{i \in \{0,\ldots,\infty\}} A^i.$$

Intuitively, A^* is the set of all paths which are concatenations of zero or more elements of A. Note that we allow an infinite concatenation of elements of A.

Definition 2-6: The *Kleene plus* operator on sets, written $^+$, is

$$A^+ \equiv_{def} \cup_{i\in\{1,\ldots,\infty\}} A^i.$$

Note that $A^* = \{ \langle\rangle \} \cup A^+$.

Now we have the tools to examine an interesting type in our LISP dialect. The type LIST[U] is useful in LISP, and our type system can express the semantics of this type.

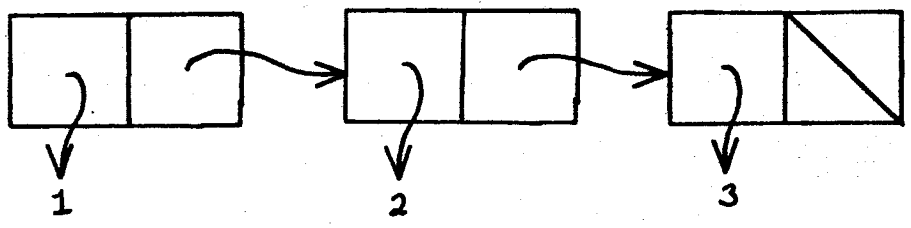
Given a cons cell O of type T with **car** of type U (where U is the set of legal paths for an object of type U), and **cdr** of the same type as O (i.e. any operation legal on O is also legal on **cdr**(O), making T a recursive type), we have

$$T = \{ \langle CDR \rangle \}^* \circ \{ \langle CAR \rangle \} \circ U.$$

An object of the type shown in Figure 2-2 might be generated by (LIST 1 2 3), where U is $\{ \langle INT \rangle \}$ in this case. Note that T is a regular set, and can thus be accepted by a finite state automaton if U is a regular set.

Note also that one of the elements of T is the infinite path x, such that $x_i = CDR$ for all positive integers i.

The examples we have presented have types which can be represented by regular sets. Solomon [22, 23] showed that the only types we should consider are the ones which can be represented by regular sets. We place an additional restriction, (the details of which are dependent on the programming language that the type system is being implemented for), that some regular sets are illegal as types. In our dialect of LISP, for example, the set $\{ \langle CAR \rangle, \langle INT \rangle \}$ is illegal, because there is nothing which has a CAR and is an integer. Thus for a given programming language there



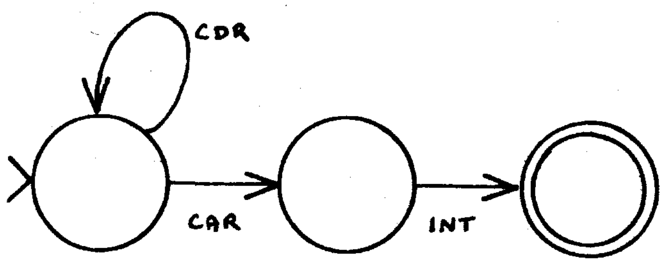


Figure 2-2: Recursive type, with an object of the type T=CONS(INT,T)
(Also known as LIST[INT]), along with the FSA which accepts T.

are *selector classes* which provide the information to check for illegal sets like $\{\langle CAR\rangle, \langle INT\rangle\}$.

We require that the *selector classes* for a given programming language, partition Σ into equivalence classes.

In VIMVAL, each equivalence class in the selector classes represents a different "type class" or "type generator" (such as ARRAY, RECORD or INT). This method of partitioning Σ would allow us to generalize our type system to include abstraction, and this possibility is discussed briefly in the conclusion. It is not essential to our work on the type checking algorithm that the selector classes are formed according to the rule that each class corresponds to a "type generator".

The selector classes for our LISP dialect are

- { *CAR*, *CDR* }

- { *INT* }

Some selector symbols can not be followed by any other selectors. Our LISP dialect does not allow paths of the form $\langle INT, CAR, ...\rangle$, because that would imply that there is some object which is an integer, which has a CAR. (It is not clear what such a path would mean, but we do not want it.) Thus, for a given programming language, some elements of Σ can only appear as the last element of a finite path.

Notation: The set of *terminators*, a subset of Σ, is the set, defined by the programming language, such that any path having a terminator in a non-final position is illegal.

In our LISP dialect, { *INT* } is the set of terminators.

In VIMVAL and our LISP dialect, the terminators correspond to "scalar" types, or "base" types. We do not, however, require that such a correspondence hold for our type checking algorithm to work.

A few extra definitions are needed to define types. We want to be able to talk about the "first part" of a set of paths, and the "last part" of a set of paths, so that types can be described in terms of these properties.

Definition 2-7: The *head* of $U \subseteq \Sigma^+$ is the set of first elements of the paths in *U*.

$$head(U) \equiv_{def} \cup_{x \in U} x_1$$

Definition 2-8: The *rest* of a path $\sigma \in \Sigma^+$ is σ with the first element removed.

$$rest(\sigma) \equiv_{def} \rho \text{ such that } \langle \sigma_1 \rangle \circ \rho = \sigma$$

Definition 2-9: The *tail* of $U \subseteq \Sigma^+$ is *U* with the first element of every path removed.

$$tail(U) \equiv_{def} \{ \sigma \mid \exists \rho \in U \text{ where } \sigma = rest(\rho) \}$$

Definition 2-10: If $X \in \Sigma$ then the *X-selected tail* of $U \subseteq \Sigma^+$ is

$$tail_X(U) \equiv_{def} \{ rest(y) \mid y \in U \text{ and } y_1 = X \}.$$

Now we can encapsulate all the type information for a given programming language into a *type system*. Type systems are dependent on their programming language: the correctness of a type system depends on the semantics of the programming language associated with it. We often refer to a *type system* as a *programming language* in this paper, because of this dependence.

Definition 2-11: A *type system* L is a three-tuple $\langle \Sigma, C, \mathcal{T} \rangle$ where

- Σ is the set of selectors in L,
- C is the set of selector classes, which partitions Σ,
- and $\mathcal{T}$ is the set of terminators. $\mathcal{T} \subseteq \Sigma$.

In order to define *type*, we need to be able to talk about certain properties of regular sets which are easily defined recursively. One such property is that for any selector σ, the σ-selected-tail of a type, T, must also be a type (or be empty). This recursion could be a real problem: e.g. for the type LIST[U], the *CDR*-selected-tail of the type LIST[U], is LIST[U]. There is no obvious way to terminate the recursion. By constructing a finite state automaton (FSA) which accepts the regular set, we can perform the tests we are interested in without resorting to such infinite recursion. The following definitions, which describe properties of FSA, were adapted from [11].

Definition 2-12: A *FSA* is a tuple $(K, \Sigma, \delta, s, F, \mathfrak{R})$ where

- K is a finite set of states,

- Σ is an input alphabet,

- δ is a function mapping some subset of $K \times \Sigma$ into K,

- s is a start state ($s \in K$),

- F is a set of accepting states ($F \subseteq K$),

- and $\mathfrak{R}$ is a reject state ($\mathfrak{R} \in K$),

and $\delta(\mathfrak{R}, \sigma)$ is undefined for all $\sigma \in \Sigma$.

Definition 2-13: A *configuration* of an FSA is a pair (k,σ) with $k \in K$ and $\sigma \in \Sigma^*$

Definition 2-14: A binary relation $\vdash_M$ holds between configurations of M, an FSA. $(k,\sigma) \vdash_M (k',\sigma') \Leftrightarrow \delta(k,\sigma_1) = k'$, and $rest(\sigma) = \sigma'$. In which case we say that (k,σ) *yields* (k',σ') *in one step.* We denote the reflexive transitive closure of $\vdash_M$ as $\vdash_M^*$. If $\delta(k,\sigma_1)$ is undefined, then $(k,\sigma) \vdash_M (\mathfrak{R},\sigma)$. ($\sigma_1$ is the first element in the path σ.)

Definition 2-15: An FSA, M, *accepts* a path σ if the following hold:

- If σ is finite then $(s,\sigma) \vdash_M^* (f,\langle\rangle)$ for some $f \in F$.

- If σ is infinite then it is not true that $(s,\sigma) \vdash_M^* (\mathfrak{R},\sigma')$ for any path σ'.

Note that if a FSA reaches a configuration (k, σ), where δ is undefined, then the FSA "hangs", and never accepts its input. Specifically, if a FSA reaches the state $\mathfrak{R}$, then the input is not accepted.

The set of paths accepted by an FSA is a *regular set of paths*, and is called the set that the FSA accepts.

We now have everything needed to define *type.*

Definition 2-16: T, a regular set of paths, is a *type* in a programming language $\langle \Sigma, C, \mathcal{T} \rangle$ if there is some FSA, $M = (K, \Sigma, \delta, s, F, \mathcal{R})$, accepting T such that

- M rejects $\langle\rangle$.

- Given a state k, if $H_k = \{ \sigma \in \Sigma \mid \delta(k, \sigma)$ is defined $\}$, then H_k is a subset of some selector class in C.

- For every state $k \in K$, and every symbol $X \in \Sigma$, if $\delta(k, X) \in F$, then $X \in \mathcal{T}$. (Terminators occur only at the end of finite paths.)

It is not necessary to force M to be unique in the definition of *type*, because if T is a type, and N is an FSA which accepts T, then N meets the conditions imposed on M in the definition of *type.* We leave this assertion without proof.

Chapter Three

Type Checking

Now that we have defined types, we can define *type-checking* by specifying the interactions between types, and programs. A program has a set of nodes[1] that we want to type (to type node N is to assign a type to N), and some information about the types of the nodes (which we call *operators*). We first lay some groundwork, defining concepts such as *program* and *type-assignments*, and then define *type-correctness* in terms of the number of possible type-assignments for a program. We conclude this chapter by showing that type-correctness is well defined and decidable.

3.1 Type Assignments and Programs

Our type checking algorithm will try to infer the type of every node in a program from its "context". We need to specify what we mean by the "context" of a node, and to do that we will define three kinds of "operators" on nodes: *parameterized restrictions*, *containers*, and *closures*.

> **Notation:** The set of node names is denoted $\mathcal{N}$. $\mathcal{N}$ must be disjoint from Σ.

$\mathcal{N}$ might be infinite, but any given program will only use a finite subset of $\mathcal{N}$.

A *type assignment* gives us a way to associate a *type* with a *node* in a program.

[1]Nodes are roughly equivalent to expressions, except that there may be some expressions that we will not want to type (for example expressions in a module which is never used), and there some things that we might want to type which are not expressions (for example type declarations). See [19] for a more complete discussion of nodes.

Definition 3-1: A *type assignment* R is a regular subset of $\mathcal{N}\circ\Sigma^*$ such that $\forall x \in head(R)$, $tail_x(R)$ is a *type*.
Notation: The set of all type assignments is denoted SOTA_{all}. Subsets of SOTA_{all} are elements of the power set of SOTA_{all}, written $\mathcal{P}(\mathrm{Sota}_{all})$.

There is an interesting isomorphism between *type assignments* and mappings from *P.NodeNames* to *types.* Given a program P, and a type assignment T, there is a mapping $U{:}\mathcal{N}\rightarrow\mathcal{T}$ such that $U(n) = tail_n(T)$. Conversely, given a mapping U, there is a type assignment T, such that $tail_n(T) = U(n)$. We named type assignments *type assignments* because they are isomorphic to mappings which assign a type to every node, and we will freely, without warning, use this isomorphism when it is convenient.

We are interested in finding which type assignments are consistent with the "context" in which each node appears in a program.

Definition 3-2: Given an alphabet $\mathcal{A}$, if $\sigma \in \mathcal{A}^*$, σ finite, and R is a regular set over $\mathcal{A}$, then the *regular set after* σ in R is

$$after_\sigma(R) \equiv_{def} \{ \sigma' \mid \sigma\circ\sigma' \in R \}.$$

Note that for a symbol $x \in \mathcal{A}$, $tail_x(R) = after_{\langle x\rangle}(R)$.

A *parameterized restriction* gives us the ability to say that two nodes are the same type. First we can specify the two nodes n and n' that we are interested in by giving two paths, σ and σ' respectively. Any FSA which represents a type assignment which is consistent with a given parameterized restriction has the property that if we start from the start state of the FSA and σ and σ' lead to states k and k' respectively, then the languages accepted by starting at k and k' must be the same. This is equivalent to saying that there must be some FSA accepting the same language such that $k = k'$. We formalize this with the definition of *state-equivalent,* and then define *parameterized restriction.*

Definition 3-3: Given a regular set R, two paths σ and σ' are *state-equivalent* if $after_{\sigma}(R) = after_{\sigma'}(R)$, in which case we write $\sigma \equiv_{R} \sigma'$.

Definition 3-4: Given a set of regular sets A (with every regular set in A over a fixed alphabet $\mathcal{A}$), for every pair $(\sigma, \sigma') \in \mathcal{A}^{*} \times \mathcal{A}^{*}$, with σ and σ' finite, there is a set of regular sets $\{ R \mid R \in \mathrm{A} \text{ and } \sigma \equiv_{R} \sigma' \}$. We call this set the *parameterized restriction* of (σ, σ'), and write the set as $\mathrm{A}_{\sigma \equiv \sigma'}$.

A *container* gives us the ability to say that the type assignment for our program has a given path σ in it.

Definition 3-5: Given a set of regular sets A (with every regular set in A over a fixed alphabet $\mathcal{A}$), for every $\sigma \in \mathcal{A}^{*}$ there is a set of regular sets $\{ R \mid R \in \mathrm{A} \text{ and } \sigma \in R \}$. We call this set the *container of σ in* A, and write the set as A_{σ}.

A *closure* gives us the ability to say that a given node must have selectors which are a subset of some finite set of selectors. We choose the node by giving a path, and specify the set by listing it.

Definition 3-6: Given a set of regular sets A (with every regular set in A over a fixed alphabet $\mathcal{A}$), a finite set of symbols $\mathfrak{B} \subseteq \mathcal{A}$, and a finite path $\sigma \in \mathcal{A}^{*}$, there is a set of regular sets $\{ R \mid R \in \mathrm{A}, \text{ and } head(after_{\sigma}(R)) \subseteq \mathfrak{B} \}$. We call this set the *closure under $\mathfrak{B}$ of R selected by σ*, and write the set as $\mathrm{A}_{\sigma[\mathfrak{B}]}$.

Now that we have defined the kinds of restrictions we want to make on type assignments, we define an operator to be one of those restrictions. A program will actually consist of some nodes and some operators.

Definition 3-7: An *operator OP* is a subset of $\mathrm{SOTA}_{\mathrm{all}}$ which is either a *parameterized restriction*, a *container*, or a *closure* of $\mathrm{SOTA}_{\mathrm{all}}$.

Notation: If *OP* is an operator, then the *operands* of *OP* are the node names mentioned *OP*.

The meaning of an operator is that if there is some restriction on the types of some nodes in a program, the operator contains the information describing the restriction.

For example, if, given a program, we have an operator which requires (informally) that "if the type of node 1 is T, then the type of node 2 is LIST[T]" then the operator is $\{ R \mid R \in SOTA_{all} \text{ and } \langle 1 \rangle \equiv_R \langle 2, LIST \rangle \}$. A more concise way of writing this set is $(SOTA_{all})_{\langle 1 \rangle \equiv \langle 2, LIST \rangle}$.

Definition 3-8: A *program* P is an ordered pair *(NodeNames, ops)*,

- where *NodeNames* is a set of node names (a finite subset of $\mathcal{N}$), referred to as *P.NodeNames*,

- and *ops* is a finite set of *operator*'s, where each *operator*'s operands are a subset of the names of the nodes in a program. (i.e. $\forall x \in ops$, *head*(x) $\in$ *P.NodeNames.*) This set is referred to as *P.ops.*

Notation: The set of all programs is referred to as Π.

By taking all of the operators in a program, and combining their information, we can deduce the type assignment for a program.

Notation: The intersection of all the operators in a program is called the *complete-restriction* of the program.

Definition 3-9: $\mathcal{OK}S\mathcal{ET}: \Pi \rightarrow \mathcal{P}(SOTA_{all})$ is a function mapping programs into sets of type assignments. Given a program P, $\mathcal{OK}S\mathcal{ET}(P)$ is defined by

$$\mathcal{OK}S\mathcal{ET}(P) \equiv_{def} \bigcap_{x \in P.ops} x.$$

3.2 Type Correctness - "There is a solution"

Definition 3-10: A program P is *type correct* if $|\mathcal{OK}S\mathcal{ET}(P)| = 1$.

Definition 3-11: A program P is *type ambiguous* if $|\mathcal{OK}S\mathcal{ET}(P)| > 1$.

Definition 3-12: A program P is *type overconstrained* if $|\mathcal{OK}S\mathcal{ET}(P)| = 0$.

Theorem 3-13: Type correctness is well defined, and is independent of the order in which the restrictions are examined for a given program.

Proof: Set intersection is associative and commutative. ∎

Peacock's proposed implementation of type checking for VIMVAL [19] used a graph, through which information about the restrictions of the operators of a program was propagated. Peacock's thesis posed the question: "Can changing the order in which constraints are propagated through the graph change the final answer?". We can answer "no" to this question because if α and β are such that given a regular set R, R_α and R_β are operators, then it is true that:

$$(R_\alpha)_\beta = (R_\beta)_\alpha$$

We accept without proof the following:

> **Proposition 3-14:** If a program is *type correct*, then no "type errors" (in the intuitive sense) will occur while running the program.

This is difficult to prove, because it is dependent on the semantics of the language the program is written in. Even if the language's type system conforms to our model, the correctness of *type correctness* depends on how accurately the set of operators for the language is described. Given a careful semantic model for a programming language, and a set of operators which are consistent with the model, a proof of this proposition would involve showing that if the local constraints imposed by the operators are true then no type errors will occur at run-time. Milner [16] proves this proposition for the language he considers. We will leave this proposition unproven for VIMVAL.

3.3 An Algorithm for Determining Type Assignments

Theorem 3-13 shows that we can talk about type correctness for incompletely typed programs with recursive types, and gives a definition of type correctness, but it does not give us an algorithm for determining those types. In this section we will prove that there is an algorithm for computing the type assignment for a given program.

If x is the intersection of a finite collection of operators, we need to show that it is possible to compute whether $|x|=0$, $|x|=1$ or $|x|>1$. If $|x|=1$, i.e. $x = \{ y \}$ for some type assignment y, then we need to show that we can actually compute y.

Specifically, we need to be able to build a FSA which accepts y, so that the VIMVAL compiler can use the type information to compile a program. (Other representations of regular sets would be equivalent to building a FSA which accepts y [11].)

Theorem 3-15: Given a program P, the type correctness of P is decidable. If P is type correct, then the type assignment is computable.

Proof: Suppose P has operators equal to the union of some containers described by the set of paths $\{ x_i \mid i=1,\dots,n \}$, and some parameterized restrictions described by the set of pairs of paths $\{ (y_i, z_i) \mid i=1,\dots,m \}$, and some closures $\{ (\mathcal{B}_i, w_i) \in \mathcal{P}(\mathcal{A}) \times \mathcal{A}^* \mid i=1,\dots,l \}$, where $\mathcal{A} = \mathcal{N} \cup \Sigma$.

We need to determine how many type assignments (which are regular sets) there are that are elements of every operator in P. Since type assignments are regular expressions, we can consider the FSA's which accept the type assignments. In general, there will be more than one FSA which accepts a given type assignment, but we can consider, without loss of generality, the set of FSA's with no more than p states, where

$p = |P.NodeNames| + \Sigma_{i=1}^{n} |x_i| + \Sigma_{i=1}^{m} (|y_i| + |z_i|) + \Sigma_{i=1}^{l} |w_i| + 3.$

The reason we can make this reduction is that the set of FSA which accept the languages described by any operator all have a bounded number of states, thus the set of FSA which accept languages in the *complete restriction* of a program also have a bounded number of states. Our bound is correct because if there are two languages meeting the restrictions of operators of the program, then there are two which need at most p states: it is possible that every time a node or symbol is mentioned by a operator, another state will be needed, plus we add one for the rejecting state, one for an accepting state, and one for an "unconstrained" state which can be used to make type assignments different for two FSA (assuming the unconstrained state is reachable from the start state).

Since it is decidable whether the language accepted by a given FSA is in a given operator, we simply need to generate a list of all FSA's with less than p states, filter out the ones which do not accept a type assignment, and determine which of them are members of every operator in P. Given this new set of FSA which are in every operator of P, we need to determine whether they all accept the same language, which is decidable. If they do, then the program is type correct. If they do not, then the program is type ambiguous. Of course, if there is no FSA which accepts a language which is in every operator of P, then P is type overconstrained.

If a program P is type correct, then the type assignment is the language accepted by one of FSA's that is found by the algorithm described above.

It is not really satisfying to be forced to use an algorithm as inefficient as the algorithm described above for determining type correctness. This algorithm is exponential in the size of the input program since the the number of FSA's of size p is exponential in p.

VIMVAL, the actual language we are trying to type check, has very stylized operators, we were able to find an algorithm for type checking which is usually more efficient. Chapter Four describes VIMVAL in more detail.

Chapter Four

Type Checking in VIMVAL

This chapter describes the types of VIMVAL [24], and how VIMVAL interacts with the type system developed in chapters 2 and 3. We deal with function recursion and polymorphism so that our type system can handle VIMVAL, then we describe the *operators* of the VIMVAL language.

4.1 The Semantics of *Modules*

A VIMVAL program consists of a set of modules, which can be compiled separately. Modules may use free names, which are references to other modules. The bindings of the free names are resolved at link time, possibly with the explicit help of the programmer. VIMVAL allows a module *M* with a free name "P" to to bind "P" to *N*, even though the name of module *N* is not "P". Unfortunately, the programmer may be required to help the linker resolve free names.

Every module is really a generator: when a module is bound to a free name, the module is augmented in whatever ways are possible and necessary to bring it into conformance with its use (i.e. it is copied, and then modified). Thus, when a programmer uses the built-in **ARRAY-SIZE** function in VIMVAL, a copy is made so that whatever type constraints are added to the **ARRAY-SIZE** function (for example if the programmer uses it on an array of integers) are not propagated to other uses of the **ARRAY-SIZE** function.

Note that we do not require that there be a unique type assignment for each module, only that there be a unique type assignment for each *augmented version* of every module. The semantics of modules does not specify that a module must be a function. A module could be some other kind of value, or even a second-class value such as a type, since the type restrictions for each of these cases could be expressed as operators.

After a copy of a module is made, the type checking system must decide on exactly one type for the module. This implies that all the types of the subexpressions of the module must have exactly one type: in particular the arguments to functions must have exactly one type. This precludes certain programs which use "run-time" polymorphism (such as the "standard" LISP interpreter).

4.2 Recursive Functions

VIMVAL allows functions to call each other recursively, with the restriction that there can be no mutual recursion between modules. (Mutual recursion between functions defined inside a module is allowed.) All recursive functions are really treated as higher order functions, which pass other functions, perhaps copies of themselves, around. This implies that recursive functions, whether directly or indirectly recursive, must be converted to passed arguments. Because arguments must have a fixed type, functions must be of fixed types when used recursively. Recursion is treated as a syntactic sugar for functions which explicitly pass other functions around [7]. Program examples 4-1 and 4-2 illustrate a simple case of the desugaring process.

Program Example 4-1:

```
% An example of recursion
function fact(i:INT) RETURNS (INT)
  IF i<=1 THEN 1
   ELSE i*fact(i-1)
   ENDIF
 ENDFUN
```

Program example 4-2 shows program example 4-1 "desugarfied". The approach taken is to translate fact into a routine which calls **dofact**, which does the actual computation.

Program Example 4-2:

```
function fact(i:int) RETURNS (INT)
  facttype = FUNCTYPE(INT,FACTTYPE) RETURNS (INT)
  function dofact(i:int,f:facttype) RETURNS (INT)
    if i<=1 then 1
     else i*f(i-1,f)
     endif
   endfun % dofact
  dofact(i,dofact)
 endfun % fact
```

There are more complex cases of mutually recursive functions. They are dealt with in the general case by translating

```
α : FUNCTION(<args>) (<rets>) IS
  expression_α-with-these-subexpressions:
    β(...)
    γ(...)
  END α
β : FUNCTION(...) .... END β
γ : FUNCTION(...) .... END γ
```

where β and γ call α (directly or indirectly) into

```
α : FUNCTION(<args>) (<rets>) IS
  do-α : FUNCTION(<args>,a,b,c) (<rets>) IS
   expression_α-with-these-subexpressions:
    b()(...,a,b,c)
    c()(...,a,b,c)
   END do-α
  do-β : FUNCTION(...,a,b,c) .... END do-β
  do-γ : FUNCTION(...,a,b,c) .... END do-γ

  do-α(<args>,do-α,do-β,do-γ)
  end α
```

Of course this only translates α. A similar translation would need to be made for β, so that β could be called directly. The following are some design considerations that we took into account when we made this decision:

- We wanted VIMVAL to have a decidable type system, and found that, theoretically, if we do not "fix" the type of recursive calls, the type becomes undecidable [8].

- We wanted an easy to understand type system. Aesthetically, an unfixed type becomes very confusing on even rather simple examples of recursion (see program example 4-3).

- Practically, very few programs need the extra expressive power of unfixed types on recursion [16].

Program Example 4-3:

```
function F(A,B)
   ...
    F(A,B)
   ...
    F(B,A)
   ...
   ENDFUN % F
F(1,1.0) % difficult to type

function F1(A,B,F2,F3)
   ...
    F2(A,B,F2,F3)
   ...
    F3(B,A,F3,F2)
   ..
   ENDFUN % F1
F1(1,1.0,F1,F1) % Much easier to type:
```

It is very difficult to give F a type in this example, because it is acceptable to pass F anything as arguments, but the arguments are switched halfway, resulting in a confusing type. If we write Fl instead, we can get the same meaning, but the program is much easier to type:

FlaTYPE=FUNCTYPE(INT,REAL,FlaTYPE,FlbTYPE) RETURNS(...)
FlbTYPE=FUNCTYPE(REAL,INT,FlbTYPE,FlbTYPE) RETURNS(...)
The type of F1 when called at the top level is FlaTYPE.
The type of the third argument is FlaTYPE.
The type of the last argument is FlbTYPE.

An example of the power of this kind of recursion is given in program example 4-4, which shows how a standard LISP function, is easily written recursively in VIMVAL.

(We also omit of the type declarations to demonstrate the ease of use of type inference.)

Program Example 4-4:

```
function LENGTH(l)
  tagcase l
    tag NullVal: 0
    tag ConsVal:
       1+length(l.cdr)
    endtag
  endfun % length
```

4.3 "Constant" copying

After dealing with recursion, the remaining free variables in each module are treated as invocations of a generator (either of a type, or a value), which does away with polymorphism (since after being copied, every node must be assigned exactly one type).

4.4 The Restrictions for VIMVAL's operators

The actual restrictions for the operators of VIMVAL are presented in appendix A. VIMVAL does not need the full expressive power of *operators*: we have described VIMVAL using:

Simplified closures

Closures are specified by a set of symbols $\mathfrak{B}$, and a path σ. VIMVAL operators are simple enough that σ can always be written as a path of length zero or one. If the path is of length zero, then the closure gives a complete list of all the node-names. Our implementation assumes that the node-names mentioned in the operators are all the node-names in the program, which is slightly easier to use than if the implementation required that an explicit list of all the node-names be presented to the type checker. If the path is of length one, then σ must be of the form $\langle n \rangle$ where n is a node-name.

Simplified containers

Containers are specified by a path σ. VIMVAL's operators can be written in such a way that all the containers are specified by paths of length two: the first element is a node-name, and the second is a terminator (which is a selector).

Simplified parameterized restrictions

Either, we have $A_{\langle n\rangle\equiv\langle m\rangle}$ or $A_{\langle n,\sigma\rangle\equiv\langle m\rangle}$, where m and n are node-names, and σ is a selector. (The general form of operators allows parameterized restrictions of the form $A_{\alpha\equiv\beta}$, where α and β are arbitrary elements of $\mathcal{N}\circ\Sigma^*$).

These restrictions allow a great improvement in the implementation of type checking in VIMVAL.

4.5 An Efficient Algorithm for Type Checking in VIMVAL

Our technique is to maintain an equivalence relation over node-names, which reflects which nodes are of the same type, information about the closure for each node, and information about the transitions that any FSA which represents some member of our complete-restriction, must follow. Hence, in most cases, we are able to rapidly reduce the upper bound of the number of states that FSA which accept our complete-restriction, by considering each equivalence class in the equivalence relation to represent one state of the FSA. The system requires at least one node-name in every equivalence class to have a closure restriction (because otherwise, it might be possible to have extra transitions leading from any state, destroying the uniqueness of the type assignment).

Definition 4-5: A *Meta Finite State Automaton* (MFSA) is a tuple $(K, \mathcal{A}, \mathcal{K}, s, F, \Sigma, \delta, \mathfrak{C})$, where

- K is a set of states,

- $\mathcal{A}$ is an accepting state ($\mathcal{A} \in K$),

- $\mathcal{K}$ is an equivalence relation over K (If $k \in K$ then $\mathcal{K}(k)$ the class of k under $\mathcal{K}$),

- s is a start state ($s \in K$),

- F is a set of final states (F is the union of some of the classes of $\mathcal{K}$, which implies $F \subseteq K$. $\mathcal{A} \in F$),

- Σ is a set of symbols,

- δ is a function mapping $(\mathcal{K} \times \Sigma) \rightarrow (\mathcal{K} \cup \{\emptyset\})$,

- and $\mathfrak{C}$ is a function mapping $\mathcal{K} \rightarrow \mathcal{P}(\Sigma)$.

Definition 4-6: A configuration of a MFSA is a pair (k, σ) where $k \in K$ and $\sigma \in \Sigma^*$

Definition 4-7: A binary relation $\vdash_M$ holds between configurations of M, a MFSA. $(k, \sigma) \vdash_M (k', \sigma') \Leftrightarrow \sigma' = \mathit{rest}(\sigma)$ and $\delta(\mathcal{K}(k), \sigma_1) = \mathcal{K}(k')$. The reflexive transitive closure of $\vdash_M$ is denoted as $\vdash_M^*$.

So far, MFSA are very similar to FSA. Now we are going to define some interesting operations which allow us to perform our type checking algorithm. First we are interested in restricting the set of FSA's that our MFSA represents to those which correspond to one of the cases of a *simplified parameterized restriction.* (See section 4.4.)

Definition 4-8: If M is an MFSA, and i and j are states then

$$equate(M,i,j) \equiv_{def} (K, \mathcal{K}', s, F', \Sigma, \delta', \mathfrak{C}'),$$

where $b \in \mathcal{K}'(a)$ (i.e. a and b are in the same class under $\mathcal{K}'$) $\Leftrightarrow$ there is some finite path σ such that $(i, \sigma) \vdash_M^* (a, \langle\rangle)$, and $(j, \sigma) \vdash_M^* (b, \langle\rangle)$,

and F' is the union of all the elements of $\mathcal{K}'$ which have some element in F,

and $\delta'(\mathcal{K}'(a), \alpha) = \mathcal{K}'(b) \Leftrightarrow \exists\ (x, y) \in \mathcal{K}'(a) \times \mathcal{K}'(b)$, such that $\delta(\mathcal{K}(x), \alpha) = \mathcal{K}(y)$,

and $\mathfrak{C}'(y) = \cap_{z \in \mathcal{K}'(y)} \mathfrak{C}(\mathcal{K}(z))$.

The *equate* operation on MFSA gives us the set of FSA's in which a given pair of states are always *state equivalent.*

Next we are interested in the case of a *container.*

Definition 4-9: If M is an MFSA, $k \in K$, and $\alpha \in \Sigma$, then

$$has-path(M, \alpha) \equiv_{def} M',$$

where if there is some $x \in \delta(\mathcal{K}(k), \alpha)$ then $M' = equate(M, x, k)$, otherwise $M' = M$, except for the transition function δ', which is the same as δ, except that $\delta'(\mathcal{K}(k), \alpha) = \mathcal{K}(\bot)$.

The next definition allows us to deal with the second case of a *simplified parameterized restriction.* (See section 4.4.)

Definition 4-10: If M is an MFSA, $i, j \in K$, and $\alpha \in \Sigma$, then

$$has-subpath-to(M, i, \alpha, j) \equiv_{def} M',$$

where if there is some $x \in \delta(\mathcal{K}(i), \alpha)$ then $M' = equate(M, x, j)$, otherwise $M' = M$ except for the function δ', which is the same as δ, except that $\delta'(\mathcal{K}(i), \alpha) = \mathcal{K}(j)$.

Note that a MFSA describes a set of type assignments if the following conditions hold:

1. For any node-name n, $\{ \alpha \mid \delta(\mathcal{K}(n), \alpha) \neq \emptyset \}$ is a subset of some selector class, and is also a subset of $\mathfrak{C}(\mathcal{K}(n))$.

2. If $n \in K$, $m \in F$, $\alpha \in \Sigma$, and $\delta(\mathcal{K}(n), \alpha) = \mathcal{K}(m)$ then α is a terminator.

3. For every $X \in \Sigma$, $\delta(\mathcal{K}(\mathcal{A}), X) = \emptyset$.

Note that a MFSA describes a single type assignment if the following condition holds:

1. For every node-name n, $\{ \alpha \mid \delta(\mathcal{K}(n), \alpha) \neq \emptyset \} = \mathfrak{C}(\mathcal{K}(m)) \neq \emptyset$.

To compute *equate*(M,i,j), *has−path*(M,i,α), and *has−subpath−to*(M,i,α,j) only takes on the order time n^2 in the worst case, and usually is much better.

To compute the type assignment for a program, we perform the following:

1. Build the MFSA with all the closures matching the closure operators in the program. (This is easy: if a node z is closed with the set $\mathfrak{B}$ in the program, we have the function $\mathfrak{C}(z) = \mathfrak{B}$. If a node z has no closures in the program, then $\mathfrak{C}(z) = \Sigma$.)

2. Construct new MFSA's, by composing the MFSA operations which correspond to the operators in the program. It does not matter which order they are composed in, since the MFSA operations describe set intersection: if A is a program operator corresponding to some MFSA operation F, and B is a set of type assignments corresponding to some MFSA M, then $A \cap B$ is a set of type assignments corresponding to the MFSA $F(M)$. Here is the correspondence between program operators and MFSA operations:

Program Operator	**MFSA Operation**
$(\mathrm{SOTA}_{\mathrm{all}})_{\langle n\rangle \equiv \langle m\rangle}$	*equate*(M,n,m)
$(\mathrm{SOTA}_{\mathrm{all}})_{\langle n,\, \sigma\rangle \equiv \langle m\rangle}$	*has−subpath−to*(M,n,σ,m)
$(\mathrm{SOTA}_{\mathrm{all}})_{\langle n,\sigma\rangle}$	*has−path*(M,n,σ)

3. Test to see if the MFSA represents a set of type assignments (in which

case we know that the program is not *type-overconstrained*), and if the MFSA represents a single type assignment (in which case we know that the program is not *type-ambiguous*).

Appendix C contains the listing of a CLU [13] program to perform type checking on VIMVAL.

Chapter Five

Conclusion

Did We Meet Our Goals?

While the VIMVAL compiler is not yet finished, and we have no actual experience using VIMVAL, we feel confident that VIMVAL has much the power and ease of use stated in our original goals. This power is illustrated by a few examples in Appendix B. We believe that VIMVAL provides a notation for polymorphic programs that is easy to learn and use, and we proved that VIMVAL is type safe, meeting the high level goals outlined in the introduction. The actual type rules of VIMVAL are fairly simple:

- There must be exactly one legal type for every value in a VIMVAL program.

- The type of a value is constrained by the operators that operate on the value. The VIMVAL manual [24], and appendix A, describe the constraints that each operator places on its operands. Intuitively, the arguments have to be used in a "consistent" way. (This is easy to state, but sometimes rather difficult to apply in practice, since the human programmer may have to actually use our algorithm to determine the type assignments of a program.)

- Recursive functions are of a fixed type, but other modules are *copied* before they are compiled, which allows polymorphic functions to be written.

VIMVAL requires a fairly complex type checking algorithm, which may require quite a bit of computation in the worst case. We believe that this complexity is acceptable in the light of VIMVAL's ease of use, and given that VIMVAL is designed to run on a highly parallel computer.

Type inference allows programmers to write code which is difficult to read. Empirically, we could argue that if type inference is difficult for a computer, it is probably also difficult for people who are reading a program. (e.g. We found it difficult to infer "in our heads" the type of the Y-combinator (shown below) but our type checking algorithm correctly computed the Y-combinator's type.)

Comparison with other Work

VIMVAL's type system is different from Milner's [16], in that we allow "ad hoc polymorphism" in the case of certain built in operators (such as +, which can take real or integer arguments). Milner discussed the possibility of adding such ad hoc polymorphism.

A more important difference between our type system and Milner's is that we allow recursive types. The recursive types allow us to type Curry's Y combinator (which Milner's system can not type).

Program Example 5-1:

```
function Y(f)
 function f1(x)
   f(x(x))
   endfun
 f1(f1)
 endfun
```

which could be re-written without type inference.

Program Example 5-2:

```
Ytype = Functype(Ytype) returns(Ytype)
function Y(f:Ytype) returns(Ytype)
  function f1(x:Ytype) returns(YType)
    f(x(x))
    endfun
  f1(f1)
  endfun
```

Except for the above differences, our concepts of *type* and sets of type assignments are not really different from Milner's. Instead of finding the "most general type" of an expression, and then instantiating the expression with specific types to get a "monotype", as Milner does, we copy the expression, and then deduce what the type of the expression must be. These approaches are equivalent, because a "monotype" is a member of a "most general type" if and only if there is context in which the expression could have *type* corresponding to the "monotype".

Our approach to types can be generalized to include type abstraction [12] by defining a correspondence between the legal operations on user defined abstract types and an augmented selector alphabet: abstract types are sets of objects with a set of operations [17], and a type checking algorithm would simply generate the additional selectors that the abstract type needs (which are different from the previously defined selectors), and put them all in the same *selector class*. None of the new selectors would be *terminators*. The rest of our type checking system would apply to this new system. We did not make this generalization because we wanted to limit the scope of this work, and because VIMVAL is perceived as a "number-crunching" language, which does not require the powerful and easy to use abstraction mechanisms that are found in CLU [13]. VIMVAL does have a type abstraction mechanism, which involves encapsulating a data type inside a procedure, but the mechanism is not easy to use (syntactic sugar would help solve this problem [14]), and it is impossible to maintain a representation invariant for objects of a given abstract type [12] (such as a requirement that an array be a sorted array).

A View from Above

The "high level goals" for the MIT Computations Structures group were well stated in [3]:

> to present a system model for a kind of ideal multiprogrammed computer system, one that would serve many users in a way permitting sharing of the products of their individual programming efforts consonant with the principles of program modularity -- the ability to build program units which can be combined to form higher units, etc.

We believe that the development of the type system for VIMVAL is an important milestone in the development of the VIMVAL language, which in turn represents an important step on the path to that high level goal.

Appendix A

VIMVAL Operators and their Restrictions

This appendix describes the actual operators that are in VIMVAL. Much of this appendix is borrowed from Peacock's [19] appendix A.

We adopt the convention that every operator has n input nodes, named $x_1,...,x_n$ and m result nodes named $y_1,...,y_m$. An operator is set of regular sets, and we give the set for each operator.

Σ =
{ *REAL, INT, CHAR, BOOL, NULL, ARRAY, STREAM,*
GET-α, IS-α, ARG-n, RET-n
| α is a legal VIMVAL identifier, and n is a positive integer }

The correspondence between selectors in our *type system*, and the "type classes" in VIMVAL are as follows:

selector		*type class*
REAL	→	REAL
INT	→	INT
CHAR	→	CHAR
BOOL	→	CHAR
ARRAY	→	ARRAY
STREAM	→	STREAM
GET-α	→	RECORD
IS-α	→	ONEOF
RET-n, ARG-n	→	FUNCTION

The terminators are

{ *REAL, INT, CHAR, BOOL, NULL* }.

The selector classes are:

{ *REAL* },
{ *INT* },
{ *CHAR* },
{ *BOOL* },
{ *NULL* },
{ *ARRAY* },
{ *STREAM* },
{ *GET*-α | α Is a legal VIMVAL identifier },
{ *IS*-α | α Is a legal VIMVAL identifier },
{ *ARG-n*, *RET-n* | *n* is a positive integer }.

We will call the set of all type assignments $\mathcal{O}$.

There is a little bit of added complexity due to the non-uniform polymorphism of some of the operators in VIMVAL. The + operator, for example allows arguments which are either all integers or all reals. We can deal with such finite disjoint unions of operators, by computing a separate *complete-restriction* for every possibility. We will refer to { *INT*, *REAL*, *CHAR*, *BOOL* } as RICB, { *REAL*, *INT* } as RI, and { *REAL*, *CHAR* } as RC.

Most operators in the VIMVAL language correspond to more than one *operator* as defined in definition 3-7. Rather than write the operators in the form $(\mathrm{SOTA}_{\mathrm{all}})_{\alpha_i}$ for *i* in some set of integers, we will write the restrictions in standard set notation.

We will also choose not to mention the closure operator for operators which mention selectors which are in selector classes of order one. This set of selectors is *OWNCLASS* = { *REAL*, *INT*, *CHAR*, *BOOL*, *NULL*, *ARRAY*, *STREAM* }. In general, if an operator specifies that there is some path $\langle z, \sigma\rangle$, with $\sigma \in$ *OWNCLASS*, then there is an implied closure operator of the form $(\mathrm{SOTA}_{\mathrm{all}})_{z[\{\sigma\}]}$.

A.1 Basic Operators

A.1.1 Error Tests

There are three universal error tests in VIMVAL. Their names are *is-undef*, *is−miss−elt*, and *is−error*. They have 1 input and 1 output. Their only constraint is that the output must be boolean.

{ S∈O | $\langle y_1, BOOL\rangle$ ∈ S }

A.1.2 Equal and Not Equal

Equal, (=), and not equal, (~=), are in a special class because they constrain their argument types not to a specific type but to a set of four possible types, namely real, integer, char, or bool. They have 2 inputs and 1 output. The inputs must be the same type and the output is a bool: Thus there is one operator for every p ∈ RICB.

∀ p ∈ RICB:
  { S∈O | { $\langle x_1, p\rangle$, $\langle x_2, p\rangle$, $\langle y_1, BOOL\rangle$ } ⊆ S }

A.1.3 Boolean Operators

There are two classes of boolean operators in VIMVAL. The first class has two arguments, the second has one.

A.1.3.1 Two Argument Boolean Operators

The members of the class with two arguments are *and*, (&); and *or*, (|). Their constraints are that all the inputs and results must be bool.

{ S∈O | { $\langle x_1, BOOL\rangle$, $\langle x_2, BOOL\rangle$, $\langle y_1, BOOL\rangle$ } ⊆ S }

A.1.3.2 One Argument Boolean Operators

The second class has only one member, the *not*, (~) operator. The input and result are both bool.

{ S∈O | { $\langle x_1, BOOL\rangle$, $\langle y_1, BOOL\rangle$ } ⊆ S }

A.1.4 Type Conversion Operations

There are three operations intended to convert one data type into another. These are *real*, *character*, and *integer*. They all have one input and one result.

real	$\{ S \in O \mid \{ \langle x_1, INT\rangle, \langle y_1, REAL\rangle \} \subseteq S \}$
integer	$\forall\, p \in RC\text{:}\ \{ S \in O \mid \{ \langle x_1, p\rangle, \langle y_1, INT\rangle \} \subseteq S \}$
character	$\{ S \in O \mid \{ \langle x_1, INT\rangle, \langle y_1, CHAR\rangle \} \subseteq S \}$

A.1.5 Real and Integer Operations

Most real and integer operations have the same names. Those that do are divided into four classes. There are some special cases, which are described after the four classes.

A.1.5.1 Binary Operators

The first class takes two arguments and returns one result, all three types being the same type, and being real or integers. The members of this class are *plus*, (+); *minus*, (-); *multiply*, (*); *divide*, (/); *max*; and *min*.

$$\forall\, p \in RI$$
$$\{ S \in O \mid \{ \langle x_1, p\rangle, \langle x_2, p\rangle, \langle y_1, p\rangle \} \subseteq S \}$$

A.1.5.2 Unary Operators

The next class has one argument and one result, both of the same type, and both either integer or real. The members of this class are *negation*, '-'; and *abs*.

$$\forall\, p \in RI$$
$$\{ S \in O \mid \{ \langle x_1, p\rangle, \langle y_1, p\rangle \} \subseteq S \}$$

A.1.5.3 Relational Operators

The next class has two arguments and one result. The arguments must be the same type, and be integer or real. The result is a boolean. The members of this class are >, <, >=, and <=.

$$\forall\, p \in RI$$
$$\{ S \in O \mid \{ \langle x_1, p\rangle, \langle x_2, p\rangle, \langle y_1, BOOL\rangle \} \subseteq S \}$$

A.1.5.4 Exception Predicates

The fourth and final class of real/integer operations has one argument and one result. The argument can be real or integer, and the result is a boolean. The members of this class are *is−pos−over*, *is−neg−over*, *is−unknown*, *is−zero−divide*, *is−over*, and *is−arith−error*.

$\forall\ p \in \mathrm{RI}$
$\{\ S \in O \mid \{ \langle x_1, p\rangle, \langle y_1, BOOL\rangle \} \subseteq S\ \}$

A.1.5.5 Special Cases

There are five operations that operate on real and integer types which do not fit into the above classes. The first of these special cases is *mod*, with two arguments and one result, all of which are integer.

$\{\ S \in O \mid \{ \langle x_1, INT\rangle, \langle x_2, INT\rangle, \langle y_1, INT\rangle \} \subseteq S\ \}$

The second special case is *exp* (which computes $x_1{}^{x}2$), with two inputs and one result. If x_2 is REAL then all are real, and if y_1 is INT then all are integers.

$\{\ S \in O \mid \{ \langle x_1, REAL\rangle, \langle x_2, REAL\rangle, \langle y_1, REAL\rangle \} \subseteq S\ \}$

$\{\ S \in O \mid \{ \langle x_1, INT\rangle, \langle x_2, INT\rangle, \langle y_1, INT\rangle \} \subseteq S\ \}$

$\{\ S \in O \mid \{ \langle x_1, REAL\rangle, \langle x_2, INT\rangle, \langle y_1, REAL\rangle \} \subseteq S\ \}$

The final three special cases are *is−pos−under*, *is−neg−under*, and *is−under*, with one input (a real) and one output (a boolean).

$\{\ S \in O \mid \{ \langle x_1, \text{"REAL"}\rangle, \langle i(y)_1, \text{"BOOL"}\rangle \} \subseteq S\ \}$

A.1.6 The empty operation

The empty operation has no inputs, and one result: a string or an array. There is a "dummy" node called *z* which is used for technical reasons:

$\{\ S \in O \mid \langle y_1, ARRAY\rangle \equiv_S \langle z\rangle\ \}$

$\{\ S \in O \mid \{ \langle y_1, STREAM\rangle \equiv_S \langle z\rangle\ \}$

A.1.7 Array Operators

A.1.7.1 Array-fill

The *array-fill* operator has three inputs and one output. The first two inputs are integers, and the output is an array of type x_3.

$\{ S \in O \mid \{ \langle x_1, INT \rangle, \langle x_2, INT \rangle \} \subseteq S,$
$\quad \text{and } \langle y_1, ARRAY \rangle \equiv_S \langle x_3 \rangle \}$

A.1.7.2 Select

The *select* operator ([]) has two inputs, an array and an integer. and an output, an element of the array.

$\{ S \in O \mid \{ \langle x_2, INT \rangle \} \subseteq S, \text{ and } \langle x_1, ARRAY \rangle \equiv_S \langle y_1 \rangle \}$

A.1.7.3 Append

The *append* operation takes three inputs and gives one result. The first input, the last input, and the output are all arrays of the same type. The second input is an integer.

$\{ S \in O \mid \{ \langle x_2, INT \rangle \} \subseteq S,$
$\quad \text{and } \langle x_1, ARRAY \rangle \equiv_S \langle x_3, ARRAY \rangle \equiv_S \langle y_1, ARRAY \rangle \}$

A.1.7.4 Create-by-elements

The *create-by-elements* operator [:] is takes n>1 inputs and gives one result. The first input is an integer, the output is an array of the second input. The rest of the inputs must be the same type as the second input.

$\{ S \in O \mid \{ \langle x_1, INT \rangle \} \subseteq S, \text{ and } \langle x_i \rangle, \langle y_1, ARRAY \rangle \text{ for } i \in \{2, \ldots, n\} \}$

A.1.7.5 Array To Integer Operators

The following three operators have the same constraints: *array-limh, array-liml* and *array-size.* They take an array input and give an integer result. We need a dummy node named *z*.

$\{ S \in O \mid \{ \langle y_1, INT \rangle \} \subseteq S, \text{ and } \langle x_1, \text{ARRAY} \rangle \equiv_S \langle z \rangle \}$

A.1.7.6 Array-adjust

The *array-adjust* operator takes three inputs and gives an output. The first two inputs are integers. The last input and the output are arrays of the same type.

$\{\ S \in O \mid \{ \langle x_1, INT \rangle, \langle x_2, INT \rangle \} \subseteq S, \text{ and } \langle x_3, ARRAY \rangle \equiv_S \langle y_1 \rangle \ \}$

A.1.7.7 Array-addh and Array-addl

The operations *array-addh* and *array-addl* both take two inputs and yield an output. The first input and the output are arrays of the the second input's type.

$\{\ S \in O \mid \langle x_1, ARRAY \rangle \equiv_S \langle x_2 \rangle \equiv_S \langle y_1, ARRAY \rangle \ \}$

A.1.7.8 Array-remh and Array-reml

The operations *array-remh* and *array-reml* both take one input and give one output. The input is an array of the output's type.

$\{\ S \in O \mid \langle x_1, ARRAY \rangle \equiv_S \langle y_1 \rangle \ \}$

A.1.7.9 Array-setl and Array-seth

The operations *array-setl* and *array-seth* take an array and an integer and give an array output. The first input and the output are arrays of the same type.

$\{\ S \in O \mid \{ \langle x_2, INT \rangle \} \subseteq S, \text{ and } \langle x_1, ARRAY \rangle \equiv_S \langle y_1, ARRAY \rangle \ \}$

A.1.7.10 Concatenate and Join

The operations *concatenate* and *array-join* takes two arrays, and give one array, all of the same type.

$\{\ S \in O \mid \langle x_1, ARRAY \rangle \equiv_S \langle x_2, ARRAY \rangle \equiv_S \langle y_1, ARRAY \rangle \ \}$

A.1.8 Stream Operations

A.1.8.1 Stream Creation

The *stream* operator allows *n* inputs and one output. There is really one operator for every non-negative number *n*. (We will assume that there is at least one input. If not, we need a dummy input, which we can call x_1.) The inputs must all be the same type, and the output is a stream of that type.

$\{ S \in O \mid \langle x_i \rangle \equiv_S \langle y_1, \mathit{STREAM} \rangle \text{ for } i \in \{ 1, \ldots, n \} \}$

A.1.8.2 Stream Null

The *null* operator takes a stream and returns a boolean. We need a dummy node named *z*.

$\{ S \in O \mid \langle y_1, \mathit{BOOL} \rangle \in S, \text{ and } \langle x_1, \mathit{STREAM} \rangle \equiv_S \langle z \rangle \}$

A.1.8.3 Stream First

The *first* operator takes a stream[T] and returns a T.

$\{ S \in O \mid \langle x_1, \mathit{STREAM} \rangle \equiv_S \langle y_1 \rangle \}$

A.1.8.4 Stream Rest

The *rest* operator takes a stream and returns a stream of the same type. We need a dummy node *z* to describe this restriction.

$\{ S \in O \mid \langle x_1 \rangle \equiv_S \langle y_1 \rangle, \text{ and } \langle x_1, \mathit{STREAM} \rangle \equiv_S \langle z \rangle \}$

A.1.8.5 Stream affix

The *affix* operator takes a stream[T] and a T, and returns a stream[T].

$\{ S \in O \mid \langle x_1, \mathit{STREAM} \rangle \equiv_S \langle x_2 \rangle \equiv_S \langle y_1, \mathit{STREAM} \rangle \}$

A.1.9 Record Operators

A.1.9.1 The Record Constructor

The $record_{\alpha_1,\ldots,\alpha_n}$ operator takes n inputs and gives one output. Note that there is one record operator for every finite set of VIMVAL identifiers. Assume that $\alpha_1, \ldots, \alpha_n$ are sorted lexicographically. We must be sure to exclude other selectors on the output.

$\{\ S \in \mathcal{O} \mid \langle y_1, \mathit{GET}\text{-}\alpha_i \rangle \equiv_S \langle x_i \rangle$ for $i \in \{1, \ldots, n\}$,
and $\langle y_1, \mathit{GET}\text{-}\beta, \ldots \rangle \notin S$ if $\beta \notin \{\ \alpha_1, \ldots, \alpha_n\ \}\ \}$

A.1.9.2 Record Selection

The $select_\alpha$ operation on records takes a record and gives a value which was stored in the record. Note that we must be careful to allow paths that start with $\mathit{GET}\text{-}\beta$, for all $\beta \neq \alpha$, because the $select_\alpha$ path does not say anything about the other selectors.

$\{\ S \in \mathcal{O} \mid \langle x_1, \mathit{GET}\text{-}\alpha \rangle \equiv_S \langle y_1 \rangle\ \}$

A.1.9.3 Record Replace

The $replace_\alpha$ operation on records takes a record and a value, and returns a new record of the same type.

$\{\ S \in \mathcal{O} \mid \langle x_1 \rangle \equiv_S \langle y_1 \rangle$, and $\langle x_1, \mathit{GET}\text{-}\alpha \rangle \equiv_S \langle x_2 \rangle\ \}$

A.1.10 Union Types

A.1.10.1 Union Make

The $make_\alpha$ operator takes an object and returns a oneof.

$\{\ S \in \mathcal{O} \mid \langle x_1 \rangle \equiv_S \langle y_1, \mathit{IS}\text{-}\alpha \rangle\ \}$

A.1.10.2 Union Is

The is_α operator takes a oneof and returns a boolean. We need a dummy node named z.

$\{\ S \in \mathcal{O} \mid \langle x_1, \mathit{IS}\text{-}\alpha \rangle \equiv_S \langle z \rangle$, and $\langle y_1, \mathit{BOOL} \rangle \in S\ \}$

A.1.11 Constants

Integer, real, and character constants have no inputs and one output. The output must be the type of the constant.

Real	$\{ S \in \mho \mid \langle y_1, REAL\rangle \in S \}$
Integer	$\{ S \in \mho \mid \langle y_1, INT\rangle \in S \}$
Character	$\{ S \in \mho \mid \langle y_1, CHAR\rangle \in S \}$

A.2 Type Declarations

Variables and Formal arguments may have type information explicitly given about them through a type specification. The type specification is treated just like an expression for the purposes of typing.

A.2.1 Basic Type Specifications

Reals, integers, characters, booleans, and null can each be specified by their names, which have selectors associated with them: *REAL*, *INT*, *CHAR*, *BOOL*, and *NULL* respectively. If we see a basic type α, with selector β, then there is only one "output", and that is the type α.

$\{ S \in \mho \mid \langle y_1, \beta\rangle \in S \}$

A.2.2 Array and Stream Type specifications

If we see a type specification ARRAY[A], where A is a type specification, then we say the "output" is an array of A, and the "input" is A.

$\{ S \in \mho \mid \langle y_1, ARRAY\rangle \equiv_S \langle x_1\rangle \}$

Similarly for streams:

$\{ S \in \mho \mid \langle y_1, STREAM\rangle \equiv_S \langle x_1\rangle \}$

A.2.3 Record and Oneof Type specifications

If we see a record type specification RECORD[α_1:A_1, ..., α_n:A_n], then we treat it exactly the same as the record constructor in section A.1.9.1.

Similarly for oneof type specifications: There is no oneof constructor that specifies all the arms, but it should be treated like a record constructor, just replace all the *GET*-α's with *IS*-α's:

$\{ S \in O \mid \langle y_1, IS\text{-}\alpha_i \rangle \equiv_S \langle x_i \rangle$ for i $\in \{1,\ldots,n\}$,
and if $\langle y_1, IS\text{-}\beta, \ldots \rangle \in R$, then $\beta = \alpha_j$ for some j $\}$

A.2.4 Function Type Specifications

Function type specifications are treated just like function applications in section A.4.1. Instead of having subexpressions, we have subtypes.

A.2.5 Free Variables as Type Specifications

A free variable just names a single node, as is true for any VIMVAL expression.

A.3 Basic Constructs

A.3.1 If then else

The *if then else* operator appears in the form:

```
IF <exp1> THEN <exp2> ELSE <exp3> ENDIF
```

We require that <exp1> be a boolean 1-valued expression, <exp2> and <exp3> be m-valued expressions, where $\langle exp2\rangle_i$ is the same type as $\langle exp3\rangle_i$ for i=1 through m. The IF is a m-valued expression.

We label the <exp1> node $x_{1,1}$, the <exp2> nodes $x_{2,1}$, ..., $x_{2,m}$, the <exp3> nodes $x_{3,1}$, ..., $x_{3,m}$, and the result nodes y_1, ..., y_m.

$\{ S \in O \mid \langle x_{1,1}, BOOL \rangle \in S$,
and $\langle x_{2,i} \rangle \equiv_S \langle x_{3,i} \rangle \equiv_S \langle y_i \rangle$ for i in $\{1,\ldots,m\}$ $\}$

A.3.2 Tagcase

The *tagcase* construct appears in the form:

```
TAGCASE <exp>
  TAG α₁ (n₁): <exp₁>
  TAG α₂ (n₂): <exp₂>
   ...
  TAG αₙ (n₃): <expₙ>
{ OTHERWISE : <expₙ₊₁> }
  ENDTAG
```

The requirements are that <exp$_1$>..<exp$_{n+1}$> are the same type, and T(<exp>) must be a oneof type with $\alpha_1..\alpha_n$ as tag values. (If the OTHERWISE is not included, then there must be no other tag values.) The value of a TAGCASE can be a m-valued expression.

We label the node of <exp> as x_0, the nodes of <exp$_i$> as x_{ij} for j=1,...,m. The resulting nodes of the tagcase are y_j for j=1...,m.

If the OTHERWISE is included we have:

```
{ S∈O | <expᵢ> ≡ₛ <y₁> for i in { 1,...,n+1 },
      and <exp, GET-αᵢ> ≡ <nᵢ> }
```

If the OTHERWISE clause is not included, add the extra restriction that there are no other tags:

```
{ S∈O | <expᵢ> ≡ₛ <y₁> for i in { 1,...,n+1 },
      and <exp, GET-αᵢ> ≡ <nᵢ>,
      and if <exp,GET-β, ...> ∈ S, then β = αᵢ for some i }
```

A.3.3 Forall construct

The *forall* construct appears as:

```
FORALL <var> IN [ <exp1> , <exp2> ]

  CONSTRUCT or EVAL <exp3>
  ENDALL
```

There are two cases, the CONSTRUCT and the EVAL case: In every case, <exp1> and <exp2> must be integer. We label <exp1>'s node x_1, <exp2>'s node x_2, and <exp3>'s node x_3. The result node is y_1.

A.3.3.1 Forall with CONSTRUCT

The restrictions for the CONSTRUCT case are that if <exp3> is of type T, then y_1 is type ARRAY[T].

$\{\ S \in O \mid \{ \langle x_1, INT\rangle, \langle x_2, INT\rangle \} \subseteq S, \text{ and } \langle x_3\rangle \equiv_S \langle y_1, ARRAY\rangle \ \}$

A.3.3.2 Forall with EVAL

There are six possible "evaluation operators" for the EVAL clause of a forall statement. In each case we have the additional restriction that the type of exp3 must be the same as the type of the output.

There are more restrictions, based on which evaluation operator is used:

In the case of +, * ,*min*, or *max* we have the restriction that <exp3> must be an integer or a real:

$\forall\ p \in RI$
$\{\ S \in O \mid \{ \langle x_1, INT\rangle, \langle x_2, INT\rangle, \langle x_3, p\rangle, \langle y_1, p\rangle \} \in S\ \}$

In the case of &, or *or* we have the restriction that <exp3> must be boolean.

$\{\ S \in O \mid \{ \langle x_1, INT\rangle, \langle x_2, INT\rangle, \langle x_3, BOOL\rangle, \langle y_1, BOOL\rangle \} \in S\ \}$

A.4 Functions

There are two ways that a function is encountered in VIMVAL. The first is the declaration of a function, which is first treated by the compiler to get rid of polymorphism and recursion. The second is when the function is passed as an argument (either to a built in operator such as apply, or as another function).

A.4.1 Function Declaration

After a function has been copied and modified to deal with polymorphism and recursion, the type checker sees a "function declaration" node, which we can write as

FUNCTION($\alpha_1,\ldots,\alpha_n$) RETURNS ($\beta_1,\ldots,\beta_m$) <EXPRESSION> END FUNCTION

where the α_i's and β_j's actually are node names of nodes inside <expression>. We assume that <expression> is m-valued, and that β_j is the name of *jth* output node of <EXPRESSION>. The resulting type constraints of a function declaration is that the output is a function taking n values, such that the *ith* value is of type α_i, and returning m values, such that the *jth* returned value is of type β_j. y_1 refers to the node of the actual function.

{ REO | $\langle y_1$, *ARG-i*$\rangle \equiv_S \langle \alpha_i \rangle$,
    and $\langle y_1$, *RET-i*$\rangle \equiv_S \langle \beta_i \rangle$ for appropriate *i*'s }

A.4.2 Function Application

If we see

<exp>(<exp$_1$>, ..., <exp$_n$>)

then we have a function application. The requirements are that <exp$_i$> be the same type as the *ith* argument of <exp>, and that the *jth* output of this function application is the type of the *jth* return value of <exp>. Here <exp> is labeled x_1, and <exp$_i$> is labeled $x_{2,i}$. The outputs are labeled y_j for appropriate values of *j*.

{ SEO | $\langle x_1$, *ARG*-i$\rangle \equiv_S \langle x_{i+1} \rangle$, and $\langle x_1$, *RET*-i$\rangle \equiv_S \langle y_i \rangle$ }

Appendix B

Examples of the power of VIMVAL

Program example 5-3 composes two functions to give a new one: One weakness in our type system is that one can not write a function which takes an arbitrary number of arguments. (This weakness is a result of the syntax of VIMVAL, rather than the type system itself.)

Program Example 5-3:

```
function compose (F:functype(B) returns (C),
                  G:functype(A) returns (B))
            returns (functype(A) returns (C))
  function composer (aval:A) returns (C)
    F(G(aval))
    endfun % composer
  composer % return the composer
  endfun % compose
```

Program example 5-4 implements the same function, with type inference instead.

Program Example 5-4:

```
function compose (F,G)
  function composer (aval)
    F(G(aval))
    endfun % composer
  composer
  endfun % compose
```

Program example 5-5 shows how a multiplier, the encapsulation of multiplication by a constant, can be implemented in VIMVAL:

Program Example 5-5:

```
% MakeMul takes an integer I and returns a
%  function which multiplies integers by I
function MakeMul(i:INT) returns (FUNCTYPE(INT) returns (INT))
  function doIt(j:int) returns (int) i*j endfun
  doIt % return doit
  endfun
```

Program example 5-6 shows how the multiplier in example 5-5 can be written without explicit type declarations. This example is slightly more powerful, in that in can operate on reals or integers.

Program Example 5-6:

```
function MakeMul(i)
  function doIt(j) i*j endfun
  doIt
  endfun
```

Program example 5-7 demonstrates a "password hider" program, which can be used to hide information, which will only be released upon presentation of the correct password. See [18] for further details on this sort of protection.

Program Example 5-7:

```
type hider=functype(givenpass:T,
                    command  :oneof[store:T; fetch])
              returns(oneof[badpass;
                            didstore:hider;
                            didfetch:T])
type pfuntype = functype(T,T) returns(boolean)

function makePassword(password:T,
                      passfun:pfuntype,
                      hiddenObject:T)
                returns (hider)
% makePassword returns a function which knows the password and knows the
% hidden object, but will not reveal the hidden object unless the user
% presents the correct password. There is also no way to uncover the
% password itself, except by subverting the type system, e.g. using
% a debugger (or perhaps by trial and error).
    function doIt(givenpass,command)
      % doIt is the function that is returned by makePassword. doIt
      % knows the password, because the password is in doIt's lexical
      % scope.
      % doIt returns the value iff the password presented causes
      %  PASSFUN(PASSWORD,GIVENPASS) to return true.
        if ~passfun(password,givenpass) then
            make[BadPass:nil]
        else
          tagcase o:=command
            tag store: make[DidStore:makePassword(password,passfun,o)]
            tag fetch: make[DidFetch:hiddenObject]
          endtag
        endif
    endfun % doIt
    doIt % return doIt
endfun % makePassword
```

Finally, we have an example which implements lisp primitives in VIMVAL.

Program Example 5-8:

```
function cons(a,b)
  make[ConsVal:record$[car:a,cdr:b]])
  endfun % cons

% The car of null is null
function car(a)
  Tagcase b:=a
     tag Consval: b.car
     tag Nullval: a
   endtag
  endfun % car

% the cdr of a null is null
function cdr(a)
  TagCase b:=a
     tag Consval: b.cdr
     tag Nullval: a
   endtag
  endfun % cdr

function nullp(a)
  is NullVal(a)
  endfun % nullp

function lispnil()
  make[nullval:null]
  endfun % lispnil

function length(a)
  if nullp(a) then 0
   else 1+length(cdr(a))
   endif
  endfun % length

function append(a,b)
  if nullp(a) then b
   else cons(car(a), append(cdr(a), b))
   endif
  endfun % append

function ith(a,i)
  if i>0 then ith(cdr(a),i-1)
   else car(a)
   endif
  endfun % ith
```

```
function reverse(a)
  % doreverse returns the first i elements of a in reverse
  function doreverse(a,i)
    if i=0 then lispnil()
     else cons(ith(a,i), doreverse(a,i-1))
     endif
    endfun % doreverse
  doreverse(i,length(a))
  endfun % reverse
```

Appendix C

Listing of the VIM-VAL type checker

This appendix contains a listing of the VIM-VAL type checker which is written in the CLU [13] programming language. The style is "functional", i.e. we have been careful to avoid side-effects, so that the eventual translation of the VIM-VAL compiler into VIM-VAL will not be too painful.

SOTA	A cluster which implements the MFSA defined in definition 4-5, along with its operations and the predicates which can be used to determine type correctness.
SET	A cluster which implements the mathematical object *set*.
EQUIVREL	A cluster which implements equivalence relations.
MAP	A cluster which implements maps from one set of objects to another set of objects.
SOTATEST	A procedure which tests SOTA.

```
#extend
sota = cluster[alphabet, nodename, classname:type] is
        create, equate, has_subpath_to, has_path, has_closed_path,
        close,
        get_unique_type_assignment,
        export % for debugging only
            where
            alphabet has get_class:proctype(alphabet) returns(classname),
                         equal:proctype(alphabet, alphabet) returns(bool),
                         get_is_terminator:proctype(alphabet) returns(bool),
            % requires: if two alphabet items A and B then
            %   A.class=B.class implies A.is_terminator=B.is_terminator
            nodename has equal:proctype(nodename, nodename) returns(bool),
            classname has equal:proctype(classname, classname) returns(bool)

    abstract = sota[alphabet,nodename,classname]
    rep = struct[equivs:ERNN,
                 closures:TNSA,
                 transitions:tntano]
    ERNN=EquivRel[NodeName]
    TNSA=map[NodeName,SA]
    SA=Set[Alphabet]
    tntano = map[NodeName, tano]
    tano = map[alphabet, no]
    no = oneof[acceptor:null,
               node:nodename]
    nopair=struct[first,second:no]
    %   nodepair=struct[first,second:nodename]
    agenda=set[nopair]

    % representation invariant I(R)
    %  R.equivs agrees with R.transitions: i.e.
    %    Equivrel[NodeName]$Equivalent(R.equivs,n,m) implies
    %      R.transitions[n] = r.transitions[m]
    %  R.transitions preserves well-typeness: i.e.
    %      R.transitions[n][a] and R.transitions[n][b] are defined implies
    %        a.class=b.class
    %  R.closures agrees with R.transitions: i.e.
    %      If R.closures[n] is defined then
    %        Domain(R.transitions[n]) Is a subset of R.closures[n]

    % abstraction function R corresponds to A iff
    %   equivrel[NodeName]$equivalent(r.equivs,n,m) iff
    %      for all Q in A,  <n> is state-equivalent to <m>
    %   equivrel[nodename]$equivalent(r.equivs,m,
    %                                  no$value_node(r.transitions[n][a])) iff
    %      for all Q in A <n,a> is state equivalent to <m>
    %   no$is_acceptor(r.transitions[n][a]) iff
    %      for all Q in A <n,a> is in Q

    create = proc() returns(cvt)
        % returns the set of all type assignments
        return(rep${equivs:ERNN$Create(),
                    closures:TNSA$create(),
                    transitions:TNTANO$create()})
        end create

    equate = proc(os:cvt, nodei,nodej:nodename) returns(cvt) signals(empty)
        % returns OS[nodei=nodej], (signals empty if there is none)
        if ERNN$Equivalent(os.equivs, nodei,nodej) then return(os) end
        ttd:agenda:=agenda$[nopair${first:no$make_node(nodei),
                                    second:no$make_node(nodej)}]
        %  ttd: things to do, but these things have to be checked for
        %       compatability AND put into the equivrel
        newequivs:ernn:=os.equivs
        while ~agenda$is_empty(ttd) do
            nowdo:nopair
            nowdo,ttd:=agenda$pick_rest(ttd)
            % the first thing to check is previous equivalence.  If they
        %  are already equivalent, then we don't need to add more.
        % after that, we should check for compatibility.  The class of the
        %  labels on the output transitions should be the same.  We really
        %  only need to test one of them from each node.
        % After that, we should gather a list of the ones that should be
        %  made if these two nodes are equivalent.  This really must be
        %  done for the whole class of them
        if no$is_node(nowdo.first) cand no$is_node(nowdo.second) then
           nowdo1:nodename:=no$value_node(nowdo.first)
           nowdo2:nodename:=no$value_node(nowdo.second)
           if ~ernn$equivalent(newequivs, nowdo1, nowdo2) then
              % now we actually have to equate them, but are they compatible?
              if ~compatible(os,nowdo1,nowdo2)
                then signal empty end
              % we must go to the mapping and add stuff
              ttd:=ttd | pairs_which_must_be_same(os, newequivs[nowdo1],
                                                 newequivs[nowdo2])
              newequivs:=ernn$equate(newequivs, nowdo1, nowdo2)
              end
         elseif no$is_node(nowdo.first) cor no$is_node(nowdo.second)
           then signal empty end
        end
     % built up newequivs, but not done yet
     % now we have to actually create the new object to return
     % we must extend the old maps
     %  (not because newequivs does not partition everything correctly, it
     %   does, but because we can only get the non_trivial_classes out, and
     %   that is not everything)
     rettrans:tntano:=os.transitions
     retclos:tnsa:=os.closures
     for eclass:set[NodeName] in ernn$non_trivial_classes(newequivs) do
        everclosed:bool:=false % did we ever hit a closure for this class?
        thistran:tano:=tano$create()
        thisclose:sa:=sa$create()
        for elt:nodename in set[nodename]$elements(eclass) do
            for al:alphabet,n:no in tano$entries(os.transitions[elt]) do
                thistran:=tano$define_override(thistran,al,n)
                end except when undefined: end
            begin
                if everclosed then
                   thisclose:=thisclose&os.closures[elt]
                   else
                        thisclose:=os.closures[elt]
                        everclosed:=true
                   end
                end % this is so we can keep track of if we closed it
              except when undefined: end
            end
        % check for the closure restriction one last time
        if everclosed cand ~tano$domain_is_in(thistran,thisclose)
           then signal empty end % not an error if never closed
        for elt:nodename in set[nodename]$elements(eclass) do
            rettrans:=tntano$define_override(rettrans,elt,thistran)
            if everclosed then
               retclos:=tnsa$define_override(retclos,elt,thisclose)
               end % don't define unless we actually closed it
            end
        end
     return(rep${equivs:newequivs,
                 closures:retclos,
                 transitions:rettrans})
     end equate

 % internal routine decides if two nodes are compatible.
 % does check the closure condition
 % we have to do is look at a rep from the domain of the transitions to
 %  see if they are the same class.  If there is none, then its ok on this.
 % we also have to check the clsoure condition
 %  check that both of these are true:
```

```
%   os.closures[n1] is undefined or contains domain(os.transitions[n2])
%   os.clusures[n2] is undefined or contains domain(os.transitinos[n1])
%  if they are both defined then this is equivalent to testing
%    if the intersection of the closure conditions contains the union
%     of the domains (This is equivalent because we already knew
%     that the closures contained the domain of their own functions
compatible = proc(os:rep, n1,n2:nodename) returns(bool)
    t1:tano:=os.transitions[n1]
       except when undefined: t1:=tano$create() end
    t2:tano:=os.transitions[n2]
       except when undefined: t2:=tano$create() end
    if tano$pick_from_domain(t1).class ~=
       tano$pick_from_domain(t2).class
       then return(false) end
       except when none: end % ok so far
    begin
        c1:sa:=os.closures[n1]
        if ~tano$domain_is_in(t2,c1) then return (false) end
        end except when undefined:
                   end % it is ok if os.closures[n1] is undefined
    begin
        c2:sa:=os.closures[n2]
        if ~tano$domain_is_in(t1,c2) then return(false) end
        end except when undefined: end % it is ok
    return(true)
    end compatible


% internal routine which returns a set of pairs that must be the same
% if the elements of s1 and s2 are to be the same under a modified OS.
% the reason we don't accept the union of s1 and s2 is that we
% would have to return all the pairs in (S1|S2) CROSS (S1|S2),
% which is no fun.
% this way, we won't have to return any such pairs, which speeds things up
%  (of course, we can if we want to, no guarantees here.)
% the pairs that we return are the ones where
pairs_which_must_be_same = proc(os:rep, s1,s2:set[nodename])
                                returns(agenda)
    % let s:=s1 union s2
    % for each element in s
    %
    % for each element in s1
    retset:agenda:=agenda$create()
    sn:sequence[nodename]:=set[nodename]$set2seq(s1)
                             || set[nodename]$set2seq(s2)
    for i:int in sequence[nodename]$indexes(sn) do
        thisname:nodename:=sn[i]
        thistano:tano:=os.transitions[thisname]
           except when undefined: thistano:=tano$create() end
        for symbol:alphabet in tano$domain_iter(thistano) do
            for j:int in int$from_to(i+1,sequence[nodename]$size(sn)) do
                thatname:nodename:=sn[j]
                % add what you get if you follow SYMBOL from thisname and thatname
                retset:=retset+
                        nopair${first:os.transitions[thisname][symbol],
                                second:os.transitions[thatname][symbol]}
                   except when undefined:
                          end % if a symbol is not there, don't worry
                end
            end
        end
    return(retset)
    end pairs_which_must_be_same

% if has_subpath exists, then we would like this to mean the same thing as
%   a:rep,b:nodename:=has_subpath(os,node_from,sym)
%   return(equate(a,b,node_to))
% but we don't use the intermediate node name
% note that in any event, if node_from.is_terminator then signals terminator
```

```
has_subpath_to = proc(os:cvt, node_from:nodename, sym:alphabet, node_to:nodename)
                   returns(cvt) signals(empty,terminator)
    % if sym is a terminator, then node_to would have to be an acceptor,
    %  which is impossible
    if sym.is_terminator then signal terminator end

    % worry about closure first
    if ~sa$elementof(sym,os.closures[node_from])
       then signal empty end
       except when undefined: end % its ok

    nmap:tano:=os.transitions[node_from]
       except when undefined: nmap:=tano$[] end

    % check for the class restriction
    if tano$pick_from_domain(nmap).class~=sym.class
       then signal empty end
       except when none: end % it is ok

    already_to:no:=os.transitions[node_from][sym]
       except
          when undefined:
               % just build the new object and return it
               return(rep${equivs:os.equivs,
                           closures:os.closures,
                           transitions:
                           tntano$define_override(
                               os.transitions,
                               node_from,
                               tano$define_override(
                                   nmap, sym, no$make_node(node_to)))})
          end

    % if it is an acceptor, it can't equate to node_to
    if no$is_acceptor(already_to) then signal empty end

    nat:nodename:=no$value_node(already_to)
    if ernn$equivalent(os.equivs,nat,node_to) then
       return(os)
       end

    % it is defined, and it meets the closure condition, but the node
    % is not equivalent. Checks again to see if meets the class property
    % inside equate
    return(down(equate(up(os),nat,node_to))) resignal empty
    end has_subpath_to

% has_subpath does the following:
%  if os.transitions[node][sym] is defined, returns os
%  otherwise, checks to see if the transitions that are already there
%   are compatible with sym (if not signals empty)
%   then creates an anonymous node which is transitioned to
%   there  if sym.is_terminator then
% returns the nodename that we go to on sym
%has_subpath = proc(os:cvt, node:nodename, sym:alphabet)
%             returns(cvt,nodename) signals(empty)
% has_subpath is not actually a defined function
%   end has_subpath

% has_path adds path <node,sym> to the transitions
% if ~sym.is_terminator then you get "non_terminator" signalled
% if sym is incompatible with the current version, signals "empty"
%   it could either be incompatible with the closure
%    or the transition class
has_path = proc (os:cvt, node:nodename, sym:alphabet)
             returns(cvt) signals(empty,non_terminator)
    % check for a terminator
    if ~sym.is_terminator then signal non_terminator end
```

```
        % check for the transition already existing, if it does then just
        % return os because it is guaranteed to be an accepting node, because
        % sym.is_terminator is true
        if tano$defined(os.transitions[node],sym) then return(os) end
           except when undefined: end % it is ok

        % check the closure condition
        if ~sa$ElementOf(sym,os.closures[node]) then signal empty end
           except when undefined: end % it is ok

        % check the transition compatiblity
        if tano$pick_from_domain(os.transitions[node]).class~=sym.class
           then signal empty end
           except when undefined:
                  when none:
                  end % it is ok

        % now return the new object
        old_tano:tano:=os.transitions[node]
           except when undefined: old_tano:=tano$create() end
        new_tano:tano:=tano$define_override(old_tano,sym,
                                            no$make_acceptor(nil))
        newtntano:tntano:=os.transitions
        for affected:nodename in set[nodename]$elements(os.equivs[node]) do
            newtntano:=tntano$define_override(newtntano,affected,new_tano)
            end
        return(rep$replace_transitions(os,newtntano))
        end has_path

    % if os can't meet the closure condition, then signal empty
    % otherwise return os with the new closure condition
    close = proc(os:cvt, node:nodename, syms:set[alphabet])
              returns(cvt) signals(empty)
        if ~tano$domain_is_in(os.transitions[node],syms)
           then signal empty end
           except when undefined: end % it is ok
        % now create the new os
        isyms:sa:=syms&os.closures[node]
           except when undefined: isyms:=syms end
        if sa$is_empty(isyms) then signal empty end
        retclosures:tnsa:=os.closures
        eclass:set[nodename]:=os.equivs[node]
           except when undefined: eclass:=set[nodename]$[node] end
        % all the equivalent nodes should have equal maps
        for ntofix:nodename in set[nodename]$elements(eclass) do
            retclosures:=tnsa$define_override(retclosures,ntofix,isyms)
            end
        return(rep$replace_closures(os,retclosures))
        end close

    % has_closed_path does close(has_path(os,node,sym),node,{sym})
    has_closed_path = proc(os:abstract, node:nodename, sym:alphabet)
                        returns(abstract) signals(empty)
        return(close(has_path(os,node,sym), node, sa$[sym]))
           resignal empty
        end has_closed_path

    % returns the map, which describes the transition function for the
    % fsa which accepts the type assignment.
    % signals ambiguous if any of the nodes named dont have some transition
    %  leading away from them.  Nodes can be named in closures, equivs, or
    %  they could have transition functions which are undefined everywhere
    % also signals ambiguous if the closure of a node is not exactly
    %  equal to the domain of the of the transition function.  This
    %  has two special cases:
    %    1) a node does not have a closure (nodes
    %       without a closure are ambiguous)
    %    2) a node has a closure, but some element of the closure does not
    %       have a transition.
    % (we are guaranteed that the domain of the transition is in the closure)
    get_unique_type_assignment =
    proc(os:cvt)
      returns(map[nodename,map[alphabet,oneof[acceptor:null,node:nodename]]])
      signals(ambiguous)
        % check for ambiguity by finding mentioned nodes that are never used
        % several ways for it to be ambiguos: an entry could have a tano
        %   with no entries, or there could be a named node somewhere with
        %    not entry in transitions
        %  or there could be a node mentioned in the closure that has no entry
        %    in transitions
        %  or there could be a node named in equivs with no entry in
        %    transitions
        for nname:nodename,ntano:tano in tntano$entries(os.transitions) do
            % if the named node does not have a closure then ambiguous
            myclosure:sa:=os.closures[nname]
               except when undefined: signal ambiguous end
            % if any of the symbols in the closure don't have a transition
            %   then ambiguous
            for symindom:alphabet in sa$elements(myclosure) do
                tano$fetch(ntano,symindom)
                end
               except when undefined: signal ambiguous end
            % if the named node has a completely undefined transition
            %  function then then ambiguous
            tano$pick_from_domain(ntano)
               except when none: signal ambiguous end
            % if any of the nodes in range of the transition
            %  dont have closures or have undefined transition
            %  functions then ambiguous
            for sym:alphabet,nrslt:no in tano$entries(ntano) do
                tagcase nrslt
                   tag acceptor: % do nothing
                   tag node(nto:nodename):
                       %% % if the node does not have a closure then it is
                       %% %  ambiguous
                       %% tnsa$fetch(os.closures,nto)
                       %%  except when undefined: signal ambiguous end
                       %% note: all the nodes are checked for this

                       % if there is no transition from nto, to another node
                       % it is ambiguous
                       tano$pick_from_domain(os.transitions[nto])
                          except when undefined,none: signal ambiguous end
                   end
                end
            end
        % if any of the nodes mentioned in the equivalence classes
        % dont have closures or have undefined transitions
        % then ambigous
        for nt_classes:set[nodename] in ernn$non_trivial_classes(os.equivs) do
            for mentioned:nodename in set[nodename]$elements(nt_classes) do
                tnsa$fetch(os.closures,mentioned)
                   except when undefined: signal ambiguous end
                tano$pick_from_domain(os.transitions[mentioned])
                   except when undefined,none: signal ambiguous end
                end
            end
        % if any of the nodes mentioned in the closures
        % dont have closures or have undefined transitions
        % then ambiguous
        for c_node:nodename in tnsa$domain_iter(os.closures) do
            tnsa$fetch(os.closures,c_node)
               except when undefined: signal ambiguous end
            tano$pick_from_domain(os.transitions[c_node])
               except when undefined,none: signal ambiguous end
            end
        return(os.transitions)
        end get_unique_type_assignment
```

```
% export returns a copy of the internal representation for os
% note that since everything is functional, this is perfectly safe
export = proc(os:cvt) returns(rep)
    return(os)
    end export
end sota
```

```
#extend
set = cluster[t:type] is
        create, new, % these are the same
        add, % add a new element
        contains,gt, % these are the same
        elementof, % other direction for contains
        mem, % does some element of a set satisfy a predicate
        elements,cons,pick,pick_rest,is_empty, % misc
        equal, % are they the same set?
        union,or, % these are the same
        intersection, and, % these are the same
        sub, % set subtraction
        set2seq
            where t has equal:proctype(t,t) returns (bool)

    rep = sequence[t]
    % create the empty set
    new = proc() returns(cvt) return(rep$[]) end new

    % add an element
    add = proc(s:cvt, el:t) returns(cvt)
        if up(s)>el then return(s) else return(rep$addh(s,el)) end
        end add
    %   low:int:=1
    %   high:int:=rep$size(s)
    %   while low<=high do
    %       i:int:=(low+high)/2
    %       if s[i]=el then return(s)
    %        elseif s[i]<el then low:=i+1
    %        else high:=i-1
    %        end
    %       end
    %   return (rep$subseq(s,1,high)
    %               || rep$[el]
    %               || rep$subseq(s,low,rep$size(s)-high))
    %   end add

    % membership operator
    gt = proc(s:cvt, el:t) returns(bool)
        for elin:t in rep$elements(s) do
            if elin=el then return (true) end
            end
        return(false)
        %       low:int:=1
        %       high:int:=rep$size(s)
        %       while low<=high do
        %           i:int:=(low+high)/2
        %           if s[i]=el then return(true)
        %            elseif s[i]<el then low:=i+1
        %            else high:=i-1
        %            end
        %           end
        %       return (false)
        end gt

    % the other name for the membership operator
    contains = proc(s:set[t], el:t) returns(bool)
        return(s>el)
        end contains

    % the other direction for the membership operator
    elementof = proc(el:t, s:set[t]) returns(bool)
        return(s>el)
        end elementof

    % return true iff there is an element K in S, such that PRED(EL,K)
    mem = proc(el:t, s:set[t], pred:proctype(t,t) returns(bool)) returns(bool)
        for knownel:t in set[t]$elements(s) do
            if pred(el,knownel) then return(true) end
            end
        return(false)
        end mem

    elements = iter(s:cvt) yields(t)
        for e:t in rep$elements(s) do yield(e) end
        end elements

    cons = proc(s:sequence[t]) returns(set[t])
        retval:set[t]:=set[t]$[]
        for e:t in sequence[t]$elements(s) do
            retval:=retval+e
            end
        return(retval)
        end cons

    pick = proc(s:cvt) returns(t) signals(empty)
        return(s[1]) except when bounds: signal empty end
        end pick

    pick_rest = proc(s:cvt) returns(t,cvt) signals(empty)
        return(s[1],rep$reml(s))
           except when bounds: signal empty end
        end pick_rest

    is_empty = proc(s:cvt) returns(bool)
        return (rep$empty(s))
        end is_empty

    % two sets are the same if they have exactly the same elements
    equal = proc(s1,s2:cvt) returns(bool)
        if s1=s2 then return(true) end % might as well optimize
        if rep$size(s1)~=rep$size(s2) then return(false) end
        for el:t in elements(up(s1)) do
            if up(s2)~>el then return(false) end
            end
        % everything in s2 is in s1, and they are in 1-1 correspondance, so
        return(true)
        end equal

    or = proc(s1,s2:set[t]) returns(set[t])
        for el:t in elements(s1) do
            s2:=s2+el
            end
        return(s2)
        %       size1:int:=rep$size(s1)
        %       size2:int:=rep$size(s2)
        %       retval:array[t]:=array[t]$predict(1,size1+size2)
        %       indx1:int:=1
        %       indx2:int:=1
        %       while indx1<=size1 cand indx2<=size2 do
        %           if s1[indx1]=s2[indx2] then
        %              array[t]$addh(retval,s1[indx1])
        %              indx1:=indx1+1
        %              indx2:=indx2+1
        %            elseif s1[indx1]<s2[indx2] then
        %              array[t]$addh(retval,s1[indx1])
        %              indx1:=indx1+1
        %            else
        %                 array[t]$addh(retval,s2[indx2])
        %                 indx2:=indx2+1
        %            end
        %           end
        %       % one of the indx's is over
        %       if indx1<size1 then
        %          return(rep$a2s(retval)||rep$subseq(s1,indx1+1,size1-indx1))
        %        elseif indx2<size2 then
        %          return(rep$a2s(retval)||rep$subseq(s2,indx2+1,size2-indx2))
        %        else return(rep$a2s(retval))
    %       end
    end or

union = proc(s1,s2:set[t]) returns(set[t]) return(s1|s2) end union

and = proc(s1,s2:set[t]) returns(set[t])
    retset:set[t]:=set[t]$[]
    for el:t in elements(s1) do
        if s2>el then retset:=retset+el end
        end
    return(retset)
    %       size1:int:=rep$size(s1)
    %       size2:int:=rep$size(s2)
    %       retval:array[t]:=array[t]$predict(1,int$min(size1,size2))
    %       indx1:int:=1
    %       indx2:int:=1
    %       while indx1<=size1 cand indx2<=size2 do
    %           if s1[indx1]=s2[indx2] then
    %              array[t]$addh(retval,s1[indx1])
    %              indx1:=indx1+1
    %              indx2:=indx2+1
    %            elseif s1[indx1]<s2[indx2] then indx1:=indx1+1
    %            else indx2:=indx2+1
    %            end
    %           end
    %       return(rep$a2s(retval))
    end and

intersection=proc(s1,s2:set[t]) returns(set[t]) return(s1&s2) end intersection

sub = proc(s1,s2:set[t]) returns(set[t])
    retset:set[t]:=set[t]$[]
    for el:t in elements(s1) do
        if s2~>el then retset:=retset+el end
        end
    %       size1:int:=rep$size(s1)
    %       size2:int:=rep$size(s2)
    %       retval:array[t]:=array[t]$predict(1,int$min(size1,size2))
    %       indx1:int:=1
    %       indx2:int:=1
    %       while indx1<=size1 cand indx2<=size2 do
    %           if s1[indx1]=s2[indx2] then
    %              indx1:=indx1+1
    %              indx2:=indx2+1
    %            elseif s1[indx1]<s2[indx2] then
    %              array[t]$addh(retval,s1[indx1])
    %              indx1:=indx1+1
    %            else indx2:=indx2+1
    %            end
    %           end
    %       if indx1<size1 then
    %          return(rep$a2s(retval)||rep$subseq(s1,indx1+1,size1-indx1))
    %          else return(rep$a2s(retval))
    %          end
    end sub

set2seq = proc(s:cvt) returns(sequence[t])
    return(s)
    end set2seq

create = proc() returns(cvt)
    return(rep$new())
    end create
end set
```

```
#extend
equivrel = cluster[T:type] is
        create,equate,non_trivial_classes,fetch,equivalent,cons,new,equal
            where T has equal:proctype(t,t) returns (bool)
    % this is immutable
    rep=map[T,set[T]]
    % abstraction function A(e:rep). if e[x] is defined, then x is in
    % the class with elements of e[x]. If e[x] is undefined, x is in { x }
    % by itself

    % rep invariant R(r:rep) if r[x] is defined then |r[x]|>1 and
    %  x is in r[x], and for all y in r[x] r[y] is defined

    % return an equivalence relation with no relations.
    % every element of T has it's own class
    create = proc() returns(cvt)
        return(rep$create())
        end create

    % create an equivalence relation with the added relationship vali=valj
    equate = proc(er:cvt,vali,valj:T) returns(cvt)
        if set[T]$ElementOf(valj,up(er)[vali]) then return(er) end
        newclass:set[T]:=set[T]$Union(up(er)[vali],up(er)[valj])
        for affected:T in set[T]$elements(newclass) do
            er:=rep$define_override(er,affected,newclass)
            end
        return(er)
        end equate

    % yield all the classes which have more than one element in them
    % watch out! This does not yield all the classes because there
    % is no way to generate a complete list of T.  Anything
    % not yielded is in its own class
    non_trivial_classes = iter(er:cvt) yields(set[T])
        did:set[T]:=set[T]$create()
        for elt:T,i:set[T] in rep$entries(er) do
            if ~set[T]$ElementOf(elt,did) then
               did:=did+elt
               yield(i)
               end
            end
        end non_trivial_classes

    % returns the class that  val is in
    fetch = proc(er:cvt, val:t) returns(set[t])
        % if t is not defined, then return set[t]$[val]
        return(er[val]) except when undefined: return(set[t]$[val]) end
        end fetch

    % if vali is in er[valj] then return true, else false
    equivalent = proc(er:equivrel[T], vali,valj:t) returns(bool)
        return(set[T]$elementOf(vali,er[valj]))
        end equivalent

    new = proc() returns(equivrel[T]) return(create()) end new

    cons = proc(ss:sequence[set[T]]) returns(cvt) signals(not_well_defined)
        ret:rep:=rep$create()
        for cl:set[T] in sequence[set[T]]$elements(ss) do
            for el:T in set[T]$elements(cl) do
                ret:=rep$define(ret,el,cl)
                   except when already_defined: signal not_well_defined end
                end
            end
        return(ret)
        end cons

    % this depends on the fact that there are no singletons!
    equal = proc(a,b:cvt) returns(bool)
```

```
        return(a=b)
        end equal
    end equivrel
```

```
map = cluster[domain,range:type] is
        create,fetch,
        define,define_override,cons,new,
        defined,
        pick_from_domain,domain_is_in,domain_iter,entries,
        equal
            where domain has equal:proctype(domain,domain) returns(bool),
                  range has equal:proctype(range,range) returns(bool)
    rep = oneof[empty:null,
                onedefined:entry]
    entry = struct[d:domain,
                   r:range,
                   rest:map[domain, range]]

    % returns a function which is undefined everywhere
    create = proc() returns(cvt)
        return(rep$make_empty(nil))
        end create

    % if fun(x) is defined, then fun(x) is returned, else signals undefined
    fetch = proc(fun:map[domain,range], x:domain) returns(range)
              signals(undefined)
        for d:domain,r:range in entries(fun) do
            if d=x then return(r) end
            end
        signal undefined
        end fetch

    % if fun(x) is defined, then returns true, else false
    defined = proc(fun:map[domain,range], x:domain) returns(bool)
        fetch(fun,x) except when undefined: return(false) end
        return(true)
        end defined

    % if fun(x) is defined to be different from f_fo_x,
    %   then signals already_defined
    % otherwise, returns a function which is the same as fun, except that
    % it is defined to be f_of_x at x.
    define = proc(fun:map[domain,range], x:domain, f_of_x:range)
               returns(cvt) signals(already_defined)
        if fun[x]=f_of_x then return(down(fun))
           else signal already_defined end
           except when undefined:
                       return(rep$make_onedefined(
                                   entry${d:x,r:f_of_x,rest:fun}))
                  end
        end define

    % an internal routine which signals SAME if fun(x)=f_of_x
    % if fun(x) is undefined signals undefined
    % and otherwise returns a function which is the same as fun, except that
    %  fun[x]=f_of_x
    do_define_override = proc(fun:cvt, x:domain, f_of_x:range)
                           returns(cvt) signals(same,undefined)
        tagcase fun
           tag empty: signal undefined
           tag onedefined(e:entry):
               if e.d=x then
                  if e.r=f_of_x then signal same
                     else return(rep$make_onedefined(
                                     entry$replace_r(e,f_of_x)))
                     end
                  else return(rep$make_onedefined(
                                  entry$replace_rest(e,do_define_override(
                                                           e.rest,x,f_of_x))))
                            resignal same,undefined
                  end
           end
        end do_define_override
    % returns fun, except that it is defined to be f_of_x at x
    % this overrides any old defns that fun had
    define_override = proc(fun:map[domain,range], x:domain, f_of_x:range)
                        returns(map[domain,range])
        % we must get rid of the previous definition, so we can't do it
        % smoothly by just consing a new thing onto the head
        return(do_define_override(fun,x,f_of_x))
           except when same: return(fun)
                  when undefined:
                       return(up(rep$make_onedefined(
                                     entry${d:x,r:f_of_x,rest:fun})))
                  end
        end define_override

    new = proc() returns(map[domain,range]) return(create()) end new

    cons = proc(ents:sequence[struct[d:domain,r:range]])
             returns(map[domain,range])
             signals(not_well_defined)
        en=struct[d:domain,r:range]
        ret:map[domain,range]:=map[domain,range]$create()
        for e:en in sequence[en]$elements(ents) do
            ret:=define(ret,e.d,e.r)
               except when already_defined: signal not_well_defined end
            end
        return(ret)
        end cons

    % if fun is undefined forall values then signals (none),
    % else returns a value for which fun is defined
    pick_from_domain = proc(fun:cvt) returns(domain) signals(none)
        tagcase fun
           tag empty: signal none
           tag onedefined(e:entry): return(e.d)
           end
        end pick_from_domain

    % if domain(fun) is in superdomain returns true, else false
    domain_is_in = proc(fun:map[domain,range], superdomain:set[domain])
                     returns(bool)
        for d:domain in domain_iter(fun) do
            if ~set[domain]$ElementOf(d,superdomain) then return(false) end
            end
        return(true)
        end domain_is_in

    % yields all the values in domain(fun)
    domain_iter = iter(fun:map[domain,range]) yields(domain)
        for d:domain,r:range in entries(fun) do
            yield(d)
            end
        end domain_iter

    % yields the pairs (d,r) where r=fun[d], and d is in the domain(fun)
    entries = iter(fun:cvt) yields(domain,range)
        while (true) do
            tagcase fun
               tag empty: return
               tag onedefined(e:entry): yield(e.d,e.r) fun:=down(e.rest)
               end
            end
        end entries

    equal = proc(f1,f2:map[domain,range]) returns(bool)
        d:domain r:range
        begin
            for d,r in entries(f1) do
                if f2[d]~=r then return(false) end
                end
            for d,r in entries(f2) do
                if f1[d]~=r then return(false) end
                end
            end
           except when undefined: return(false) end
        return(true)
        end equal
    end map
```

```
#extend
alphabet=struct[class:string,
                is_terminator:bool,
                name:string]
nodename=int % we will use negatives if we need dummy's
classname=string
vimsota=sota[alphabet,nodename,classname]
tmap=map[alphabet,oneof[acceptor:null,node:nodename]]
vimsotarep=struct[equivs:ernn, closures:tnsa, transitions:tntano]
ernn=equivrel[nodename]
snn=set[nodename]
tnsa = map[nodename,sa]
sa=set[alphabet]
tntano=map[nodename,tano]
tn_ent=struct[d:nodename,r:tano]
ta_ent=struct[d:alphabet,r:no]
ts_ent=struct[d:nodename,r:sa]
tano=map[alphabet,no]
no=oneof[acceptor:null, node:nodename]

% this routine does some testing on the sota
sotatest = proc()
    vsr=vimsotarep
    putl=stream$putl
    po:stream:=stream$primary_output()
    % first test, do a create, and get the rep which should be totally empty
    s_create:vimsota:=vimsota$create()
    sexpect("s_create",s_create,
            vsr${equivs:ernn$[], closures:tnsa$[], transitions:tntano$[]},
            true)

    % now we have really tested the create out. That really only
    % gives us a little confidence in the lower level objects,
    % since create is so simple.
    noa:no:=no$make_acceptor(nil)
    no1:no:=no$make_node(1) no2:no:=no$make_node(2) no3:no:=no$make_node(3)
    no4:no:=no$make_node(4) no5:no:=no$make_node(5) no6:no:=no$make_node(6)
    a_int:alphabet:=alphabet${class:"INT", is_terminator:TRUE, name:"INT"}
    a_string:alphabet:=
        alphabet${class:"STRING", is_terminator:TRUE, name:"STRING"}
    a_real:alphabet:=alphabet${class:"REAL", is_terminator:TRUE, name:"REAL"}
    a_array:alphabet:=
        alphabet${class:"ARRAY", is_terminator:FALSE, name:"ARRAY"}
    a_geta:alphabet:=
        alphabet${class:"STRUCT", is_terminator:FALSE, name:"GET_A"}
    a_getb:alphabet:=
        alphabet${class:"STRUCT", is_terminator:FALSE, name:"GET_B"}
    a_getc:alphabet:=
        alphabet${class:"STRUCT", is_terminator:FALSE, name:"GET_C"}

    % lets try equating two nodes.  We should then get an ambiguous error
    % if we try to get the typemap
    s_1e2:vimsota:=vimsota$equate(s_create,1,2)
    sexpect("s_1e2",s_1e2,
            vsr${equivs:ernn$[snn$[1,2]],
                 closures:tnsa$[], transitions:tntano$[]},
            false)
    % the transitions and closures should be completely undefined
    % the equivclass should have exactly {1,2} in it

    % try something really fancy:
    %  a real problem:  N1 is an array of n2
    %                   N2 is an int
    %          does it work?
    %
    %    N1 = ARRAY[N2]
    %    N2 = ARRAY[N1]
    %          does it work?
    %
```

```
%    N1 = INT
%    N2 = ARRAY
%          does it not work?
%
%    N1 = ARRAY[N2]
%    N3 = ARRAY[N4]
%    N1 = N3
%          does it work?
%
%    N1 = ARRAY[N2]
%    N3 = ARRAY[N4]
%    N2 = INT
%    N4 = STRING
%    N1 = N3
%          does it not work
%
%    N1 = CLOSED_STRUCT[A:N2,B:N3]
%    N2 = INT
%    N3 = STRING
%    N1 = OPEN_STRUCT[A:N4]
%          does it work
%
%    N1 = CLOSED_STRUCT[A:N2,B:N3]
%    N2 = INT
%    N3 = STRING
%    N1 = OPEN_STRUCT[C:N3]
%          does it not work
%
%    N1 = CLOSED_STRUCT[A:n2,B:N3]
%    N2 = INT
%    N4 = CLOSED_STRUCT[A:n5:b:n6]
%    n6 = STRING
%    N1 = N4
%          does it work
%
%   that pretty well tests the closure with equates
%   now for some recursion
%    N1 = ARRAY[N1]
%          deos it work?
%
%    N1 = ARRAY[N2]
%    N2 = ARRAY[N1]
%          does it work?
%
%    N1 = CLOSED_STRUCT[a:N2, b:N3]
%    N2 = CLOSED_STRUCT[a:N2, b:N4]
%    N4 = N1
%          does it work?
%
%    N1 = CLOSED_STRUCT[a:N2, b:N3]
%    N2 = CLOSED_STRUCT[a:N1, c:N3]
%          does it work?
%
%    N1 = CLOSED_STRUCT[A:N2, B:N3]
%    N2 = N1
%    N3 = N2
%    N3 = CLOSED_STRUCT[A:N2, C:N3]
%          does it not work?
%
% test the terminators to see if it won't allow has_path to be a non-terminator
% N1 = ARRAY[N2]
%   does it not work (ambiguity)

% the comments are repeated:

% try something really fancy:
%  a real problem:  N1 is an array of n2
%                   N2 is an int
%          does it work?
```

```
%
ev:vsr:=vsr${equivs:ernn$[],
             closures:tnsa$[ts_ent${d:1,r:sa$[a_array]},
                            ts_ent${d:2,r:sa$[a_int]}],
             transitions:tntano$[
                            tn_ent${d:1,r:tano$[ta_ent${d:a_array,r:no2}]},
                            tn_ent${d:2,r:tano$[ta_ent${d:a_int,r:noa}]}]}
sexpect("N2=INT,N1=ARRAY[N2]",
        vimsota$has_subpath_to(
            vimsota$close(vimsota$has_closed_path(s_create,2,a_int),
                          1,sa$[a_array]),
            1,a_array,2),
        ev, true)
sexpect("N1=ARRAY[N2],N2=INT",
        vimsota$has_closed_path(
            vimsota$close(vimsota$has_subpath_to(s_create,1,a_array,2),
                          1,sa$[a_array]),
            2,a_int),
        ev,true)

%    N1 = ARRAY[N2]
%    N2 = ARRAY[N1]
%         does it work?
sexpect("N1=ARRAY[N2],N2=ARRAY[N1]",
        vimsota$close(
            vimsota$has_subpath_to
                (vimsota$close(
                     vimsota$has_subpath_to(s_create,1,a_array,2),
                     1,sa$[a_array]),
                 2,a_array,1),
            2,sa$[a_array]),
        vsr${equivs:ernn$[],
             closures:tnsa$[ts_ent${d:1,r:sa$[a_array]},
                            ts_ent${d:2,r:sa$[a_array]}],
             transitions:
             tntano$[tn_ent${d:1,r:tano$[ta_ent${d:a_array,r:no2}]},
                     tn_ent${d:2,r:tano$[ta_ent${d:a_array,r:no1}]}]},
        true)

%    N1 = INT
%    N2 = ARRAY[N1]
%         does it work?
sexpect("N1=INT,N2=ARRAY[N1]",
        vimsota$close(vimsota$has_subpath_to(
                          vimsota$has_closed_path(s_create,1,a_int),
                          2,a_array,1),
                      2,sa$[a_array]),
        vsr${equivs:ernn$[],
             closures:tnsa$[ts_ent${d:1,r:sa$[a_int]},
                            ts_ent${d:2,r:sa$[a_array]}],
             transitions:
             tntano$[tn_ent${d:1,r:tano$[ta_ent${d:a_int,r:noa}]},
                     tn_ent${d:2,r:tano$[ta_ent${d:a_array,r:no1}]}]},
        true)

% same thing without the closure on the int
sexpect("N1=NC_INT,N2=ARRAY[N1]",
        vimsota$close(vimsota$has_subpath_to(
                          vimsota$has_path(s_create,1,a_int),
                          2,a_array,1),
                      2,sa$[a_array]),
        vsr${equivs:ernn$[],
             closures:tnsa$[ts_ent${d:2,r:sa$[a_array]}],
             transitions:
             tntano$[tn_ent${d:1,r:tano$[ta_ent${d:a_int,r:noa}]},
                     tn_ent${d:2,r:tano$[ta_ent${d:a_array,r:no1}]}]},
        false)

% same thing, without the closure on the array
```

```
sexpect("N1=INT,N2=NC_ARRAY[N1]",
        vimsota$has_subpath_to(
            vimsota$has_closed_path(s_create,1,a_int),
            2,a_array,1),
        vsr${equivs:ernn$[],
             closures:tnsa$[ts_ent${d:1,r:sa$[a_int]}],
             transitions:
             tntano$[tn_ent${d:1,r:tano$[ta_ent${d:a_int,r:noa}]},
                     tn_ent${d:2,r:tano$[ta_ent${d:a_array,r:no1}]}]},
        false)

%    N1 = ARRAY[N2]
%    N3 = ARRAY[N4]
%    N1 = N3
%    N2 = INT
%      should work when all closed
tmp1:vimsota:= vimsota$close(
                   vimsota$close(
                       vimsota$has_subpath_to(
                           vimsota$has_subpath_to(s_create,1,a_array,2),
                           3,a_array,4),
                       3,sa$[a_array]),
                   1,sa$[a_array])
tmp1:= vimsota$has_closed_path(vimsota$equate(tmp1, 1,3),
                               2,a_int)

sexpect("N1=A[N2],N3=A[N4],N1=N3,N2=INT",
        tmp1,
        vsr${equivs:ernn$[snn$[1,3],snn$[2,4]],
             closures:tnsa$[ts_ent${d:1,r:sa$[a_array]},
                            ts_ent${d:2,r:sa$[a_int]},
                            ts_ent${d:3,r:sa$[a_array]},
                            ts_ent${d:4,r:sa$[a_int]}],
             transitions:
             tntano$[tn_ent${d:1,r:tano$[ta_ent${d:a_array,r:no2}]},
                     tn_ent${d:2,r:tano$[ta_ent${d:a_int,r:noa}]},
                     tn_ent${d:3,r:tano$[ta_ent${d:a_array,r:no4}]},
                     tn_ent${d:4,r:tano$[ta_ent${d:a_int,r:noa}]}]},
        true)

%    N1 = ARRAY[N2]
%    N3 = ARRAY[N4]
%    N2 = INT
%    N4 = STRING
%    N1 = N3
%         can't build it, don't even bother with the closures
tmp:vimsota:=
    vimsota$has_path(vimsota$has_path(
                         vimsota$has_subpath_to(
                             vimsota$has_subpath_to(s_create,1,a_array,2),
                             3,a_array,4),
                         2,a_int),
                     4,a_string)
% should work up to here
begin
    vimsota$equate(tmp,1,3)
    stream$putl(stream$primary_output(),"Can build exA, wrong")
    signal failure("Can build exA, wrong")
    end
   except when empty:
              stream$putl(stream$primary_output(),"Can't build exA, ok")
          end

%    N1 = CLOSED_STRUCT[A:N2,B:N3]
%    N2 = INT
%    N3 = STRING
%    N1 = OPEN_STRUCT[A:N4]
%      should work
tmp:=vimsota$has_subpath_to(
        vimsota$has_subpath_to(
            vimsota$close(s_create,1,sa$[a_geta,a_getb]),
            1,a_geta,2),
        1,a_getb,3)
tmp:=vimsota$has_closed_path(tmp,2,a_int)
tmp:=vimsota$has_closed_path(tmp,3,a_string)
tmp:=vimsota$has_subpath_to(tmp,1,a_geta,4)
sexpect("N1=CS[A:N2,B:N3],N2=INT,N3+S,N1=O[a:n4]",tmp,
        vsr${equivs:ernn$[snn$[2,4]],
            closures:tnsa$[ts_ent${d:1,r:sa$[a_geta,a_getb]},
                           ts_ent${d:2,r:sa$[a_int]},
                           ts_ent${d:3,r:sa$[a_string]},
                           ts_ent${d:4,r:sa$[a_int]}],
            transitions:
            tntano$[tn_ent${d:1,r:tano$[ta_ent${d:a_geta,r:no2},
                                        ta_ent${d:a_getb,r:no3}]},
                    tn_ent${d:2,r:tano$[ta_ent${d:a_int,r:noa}]},
                    tn_ent${d:3,r:tano$[ta_ent${d:a_string,r:noa}]},
                    tn_ent${d:4,r:tano$[ta_ent${d:a_int,r:noa}]}]},
        true)


%
%    N1 = CLOSED_STRUCT[A:N2,B:N3]
%    N1 = OPEN_STRUCT[C:N3]
%        does it not work because of closure violation
tmp:=vimsota$close(vimsota$has_subpath_to(
                    vimsota$has_subpath_to(s_create,1,a_geta,2),
                    1,a_getb,3),
                1,sa$[a_geta,a_getb])
begin
    vimsota$has_subpath_to(tmp,1,a_getc,3)
    signal failure("Could build s_cab_pc, wrong")
    end
   except when empty: stream$putl(stream$primary_output(),
                                  "Couldnt buld s_cab_pc, ok") end

%    N1 = OPEN_STRUCT[A:n2,B:N3]
%    N2 = INT
%    N4 = OPEN_STRUCT[A:n5:b:n6]
%    n6 = STRING
%    N1 = N4
%        does it not work
tmp:=vimsota$has_subpath_to(vimsota$has_subpath_to(s_create,1,a_geta,2),
                            1,a_getb,3)
tmp:=vimsota$has_closed_path(tmp,2,a_int)
tmp:=vimsota$has_subpath_to(vimsota$has_subpath_to(tmp,1,a_geta,5),
                            1,a_getb,6)
tmp:=vimsota$has_closed_path(vimsota$equate(tmp,1,4),6,a_string)
_14_trans:tano:=tano$[ta_ent${d:a_geta,r:no2},
                      ta_ent${d:a_getb,r:no3}]
_25_trans:tano:=tano$[ta_ent${d:a_int,r:noa}]
_36_trans:tano:=tano$[ta_ent${d:a_string,r:noa}]
mytrans:tntano:= tntano$[tn_ent${d:1, r:_14_trans},
                         tn_ent${d:4, r:_14_trans},
                         tn_ent${d:2, r:_25_trans},
                         tn_ent${d:5, r:_25_trans},
                         tn_ent${d:3, r:_36_trans},
                         tn_ent${d:6, r:_36_trans}]

sexpect("Two-defined struct unclosed",tmp,
        vsr${equivs:ernn$[snn$[2,5],snn$[3,6],snn$[1,4]],
            closures:tnsa$[ts_ent${d:2,r:sa$[a_int]},
                           ts_ent${d:5,r:sa$[a_int]},
                           ts_ent${d:3,r:sa$[a_string]},
                           ts_ent${d:6,r:sa$[a_string]}],
            transitions:mytrans},
        false)
```

```
sexpect("Two-defined struct closed",
        vimsota$close(tmp,1,sa$[a_geta,a_getb]),
        vsr${equivs:ernn$[snn$[2,5],snn$[3,6],snn$[1,4]],
             closures:tnsa$[ts_ent${d:2,r:sa$[a_int]},
                            ts_ent${d:5,r:sa$[a_int]},
                            ts_ent${d:3,r:sa$[a_string]},
                            ts_ent${d:6,r:sa$[a_string]},
                            ts_ent${d:1,r:sa$[a_geta,a_getb]},
                            ts_ent${d:4,r:sa$[a_geta,a_getb]}],
             transitions:mytrans},
        true)

%   that pretty well tests the closure with equates
%   now for some recursion
%    N1 = ARRAY[N1]
%          does it work?
sexpect("N1=A[N1]",
        vimsota$close(vimsota$has_subpath_to(s_create,1,a_array,1),
                      1,sa$[a_array]),
        vsr${equivs:ernn$[],
             closures:tnsa$[ts_ent${d:1,r:sa$[a_array]}],
             transitions:
             tntano$[tn_ent${d:1, r:tano$[ta_ent${d:a_array,r:no1}]}]},
        true)

%    N1 = ARRAY[N2]
%    N2 = ARRAY[N1]
sexpect("N1=A[N2] N2=A[N1]",
        vimsota$has_subpath_to(
            vimsota$has_subpath_to(
                vimsota$close(
                    vimsota$close(s_create,2,sa$[a_array]),
                    1,sa$[a_array]),
                1,a_array,2), 2,a_array,1),
        vsr${equivs:ernn$[],
             closures:tnsa$[ts_ent${d:1,r:sa$[a_array]},
                            ts_ent${d:2,r:sa$[a_array]}],
             transitions:
             tntano$[tn_ent${d:1, r:tano$[ta_ent${d:a_array,r:no2}]},
                     tn_ent${d:2, r:tano$[ta_ent${d:a_array,r:no1}]}]},
        true)

%    N1 = closed_STRUCT[a:N2, b:N3]
%    N2 = open_STRUCT[a:N2, b:N4]
%    N4 = N1
%    N2 = N3
%    N3 = N4
% everything should come out to be the same thing
tmp:=vimsota$close(vimsota$has_subpath_to(
                     vimsota$has_subpath_to(s_create,1,a_geta,2),
                     1,a_getb,3),
                   1,sa$[a_geta,a_getb])
tmp:=vimsota$has_subpath_to(vimsota$has_subpath_to(tmp,2,a_geta,2),
                            2,a_getb,4)
tmp:=vimsota$equate(vimsota$equate(vimsota$equate(tmp,1,4),2,3),3,4)
_trans:tano:=tano$[ta_ent${d:a_geta,r:no1},
                   ta_ent${d:a_getb,r:no2}]
_close:sa:=sa$[a_geta,a_getb]
sexpect("Ex C",tmp,
        vsr${equivs:ernn$[snn$[1,2,3,4]],
             closures:tnsa$[ts_ent${d:1,r:_close},
                            ts_ent${d:2,r:_close},
                            ts_ent${d:3,r:_close},
                            ts_ent${d:4,r:_close}],
             transitions:tntano$[tn_ent${d:1,r:_trans},
                                 tn_ent${d:2,r:_trans},
                                 tn_ent${d:3,r:_trans},
                                 tn_ent${d:4,r:_trans}]},
        true)
```

```
%     N1 = CLOSED_STRUCT[a:N2, b:N3]
%     N2 = CLOSED_STRUCT[a:N1, c:N3]
%     N3 = INT
%           does it work?
tmp:=vimsota$has_closed_path(s_create,3,a_int)
tmp:=vimsota$close(vimsota$has_subpath_to(
                        vimsota$has_subpath_to(tmp,1,a_geta,2),
                        1,a_getb,3),
                    1,sa$[a_geta,a_getb])
tmp:=vimsota$close(vimsota$has_subpath_to(
                        vimsota$has_subpath_to(tmp,2,a_geta,1),
                        2,a_getc,3),
                    2,sa$[a_geta,a_getc])
sexpect(
    "Ex D",tmp,
    vsr${equivs:ernn$[],
         closures:tnsa$[ts_ent${d:1,r:sa$[a_geta,a_getb]},
                        ts_ent${d:2,r:sa$[a_geta,a_getc]},
                        ts_ent${d:3,r:sa$[a_int]}],
         transitions:tntano$[
                         tn_ent${d:1,r:tano$[ta_ent${d:a_geta,r:no2},
                                             ta_ent${d:a_getb,r:no3}]},
                         tn_ent${d:2,r:tano$[ta_ent${d:a_geta,r:no1},
                                             ta_ent${d:a_getc,r:no3}]},
                         tn_ent${d:3,r:tano$[ta_ent${d:a_int,r:noa}]}]},
    true)


sexpect("Closure, but not all there",
        vimsota$close(vimsota$has_subpath_to(
                          vimsota$has_closed_path(s_create,2,a_int),
                          1,a_geta,2),
                      1,sa$[a_geta,a_getb]),
        vsr${equivs:ernn$[],
             closures:tnsa$[ts_ent${d:1,r:sa$[a_geta,a_getb]},
                            ts_ent${d:2,r:sa$[a_int]}],
             transitions:
             tntano$[tn_ent${d:1,r:tano$[ta_ent${d:a_geta,r:no2}]},
                     tn_ent${d:2,r:tano$[ta_ent${d:a_int,r:noa}]}]},
        false)

% check for class error
tmp:=vimsota$has_subpath_to(s_create,1,a_geta,2)
begin
    tmp:=vimsota$has_subpath_to(tmp,1,a_array,2)
    signal failure("class error 1 not caught")
    end except when empty: stream$putl(stream$primary_output(),
                                       "class error 1 caught ok")
           end

tmp:=vimsota$has_path(s_create,1,a_int)
begin
    tmp:=vimsota$has_path(tmp,1,a_string)
    signal failure("class error 2 not caught")
    end except when empty: stream$putl(stream$primary_output(),
                                       "class error 2 caught ok")
           end

% check for path with non-terminator error
begin
    tmp:=vimsota$has_path(s_create,1,a_array)
    signal failure("has_path with non terminator not caught")
    end except when non_terminator:
                stream$putl(stream$primary_output(),
                            "has_path with non terminator caught ok")
           end
    % check for subpath_to with terminator error
    begin
        tmp:=vimsota$has_subpath_to(s_create,1,a_int,2)
        signal failure("has_subpath_to with terminator not caught")
        end except when terminator:
                        stream$putl(stream$primary_output(),
                                    "has_subpath_to with terminator caught ok")
                  end

    end sotatest

% if the rep of the mysota is not equal to expected_rep then prints an
% error, other wise prints "ok"
sexpect=proc(name:string, mysota:vimsota, expected_rep:vimsotarep,guta:bool)
    own po:stream:=stream$primary_output()
    died:bool:=false
    exp:vimsotarep:=vimsota$export(mysota)
    stream$puts(po,name)
    if exp.equivs=expected_rep.equivs then
       stream$puts(po," equivs ok,")
       else
            stream$puts(po," equivs broken,")
            died:=true
       end
    if exp.closures=expected_rep.closures then
       stream$puts(po," closures ok,")
       else
            stream$puts(po," closures broken,")
            died:=true
       end
    % have to do the mapping test badly, sigh, this is because
    % i am really modeling (nodename,alphabet)->nodename, but
    % ended up using nodename->(nodename->alphabet)
    trandied:bool:=false
    begin
        for tn:nodename, ta:tano in tntano$entries(exp.transitions) do
            for ts:alphabet, tno:no in tano$entries(ta) do
                etn:no:=expected_rep.transitions[tn][ts]
                tagcase tno
                   tag acceptor: if etn~=tno then exit bad_map end
                   tag node(num:int):
                       if ~set[int]$ElementOf(no$value_node(etn),
                                              exp.equivs[num])
                          then exit bad_map end
                   end
                end
            end
        for tn:nodename, ta:tano in tntano$entries(expected_rep.transitions) do
            for ts:alphabet, tno:no in tano$entries(ta) do
                etn:no:=exp.transitions[tn][ts]
                tagcase tno
                   tag acceptor: if etn~=tno then exit bad_map end
                   tag node(num:int):
                       if ~set[int]$ElementOf(no$value_node(etn),
                                              expected_rep.equivs[num])
                          then exit bad_map end
                   end
                end
            end
        end
      except when undefined,bad_map,wrong_type: trandied:=true died:=true end

    if trandied then stream$puts(po," transitions broken,")
      else stream$puts(po," transitions ok,") end

    begin
        vimsota$get_unique_type_assignment(mysota)
        if guta then stream$putl(po," guta defined ok")
           else
```

```
                stream$putl(po," expected guta ambiguity, it wasn't")
                died:=true
           end
        end
       except when ambiguous:
                    if guta then
                       stream$putl(po," but expect guta defined, it wasn't")
                       died:=true
                       else
                            stream$putl(po," guta ambiguous ok")
                       end
              end
    if died then signal failure("died---") end
    end sexpect
```

References

[1] Ackerman, W.B., Dennis, J. B.
VAL--A Value-Oriented Algorithmic Language: Preliminary Reference Manual.
Technical Report MIT/LCS/TR-218, Massachusetts Institute of Technology Laboratory for Computer Science, June, 1979.

[2] Demers, A., Donahue. J., Skinner, G.
Data Types as Values: Polymorphism, Type-checking, Encapsulation.
In *Fifth Annual ACM Symposium on Principles of Programming Languages,* pages 23-30. ACM, January, 1978.
Presents a uniform treatment of explicitly paramterized types.

[3] Dennis, J.B.
Data Should Not Change: A Model for a Computer System.
Massachusetts Institute Technology, Cambridge, Massacusetts, Laboratory for Computer Science, Computation Structures Group Memo 209.

[4] Dennis, J. B.
An Operational Semantics for a Language with Early Completion Data Structures.
In *Presented at the International Colloquium on The Formalization of Programming Concepts, Peniscola, Spain.* April 19-25, 1981.

[5] Dennis, Gao, Todd.
Modeling the Weather With a Data Flow Supercomputer.
IEEE Transactions on Computers , 1984.
To appear.

[6] Donahue, J.
On the Semantics of Data Types.
Technical Report TR-77-311, Cornell Department of Computer Science, June, 1977.

[7] Henderson, P.
Functional Programming Application and Implementation.
Prentice-Hall International, 1980.

[8] Langmack, H.
On Correct Procedure Paramter Transmission in Higher Programming Languages.
Acta Informatica 2:110-142, 1973.
Proves undecidability of type correctness for languages which allow formal arguments to be used polymorphically.

[9] Leivant, D.
Structural Semantics for Polymorphic Data Types.
In *Tenth Annual ACM Symposium on Principles of Programming Languages,* pages 155-166. ACM, January, 1983.

[10] Leivant, D.
Polymorphic Type Inference.
In *Tenth Annual ACM Symposium on Principles of Programming Languages,* pages 88-98. ACM, January, 1983.
Contains a summary of several competing type systems, with an algorithm that works for all of them.

[11] Lewis, H.R., Papadimitriou, C. H.
Elements of the Theory of Computation.
Prentice-Hall, 1981.

[12] Liskov, B., Snyder, M., Atkinson, R., and Schaffert, C.
Abstraction Mechanisms in CLU.
CACM 20(8):564, August, 1977.

[13] Liskov, B., Atkinson, R., Bloom, T., Moss, T., Schaffert, C., Scheifler, B., and Snyder, A.
CLU Reference Manual.
Technical Report MIT/LCS/TR-225, Massachusetts Institute Technology, October, 1979.
Also available from Springer-Verlag as the "Lecture Notes in Computer Science" series.

[14] McCracken, N. J.
An Investigation of a Programming Language with a Polymorphic Type Structure.
PhD thesis, School of Computer and Information Science, Syracuse University, June, 1979.
Discusses parameterized and user defined types.

[15] Meertens, L.
Incremental Polymorphic Type Checking in B.
In *Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 265-275. ACM, January, 1983.
B has nonrecursive types without higher order functions. Contains a presentation of types and incremental type checking with polymorphism.

[16] Milner, R.
A Theory of Type Polymorphism in Programming.
Journal of Computer and System Sciences 17(3):348-375, December, 1978.

[17] Morris, J.H.
Types are not Sets.
In *Proc. ACM Symposium on Principles of Programming Languages*, pages 120-124. ACM, October, 1973.

[18] Morris, J.
Protection in Programming Languages.
CACM 16(1):15-21, June, 1973.

[19] Peacock, T.
Type Checking in Generalized VAL.
Massachusetts Institute Technology, Laboratory for Computer Science, Computations Structures Group Memo 227, May 1983.
Undergraduate thesis.

[20] Scheidig, H.
Representation and Equality of Modes.
Inf. Proc. Letters 1:61-65, 1971.

[21] Scott, D.B.
Data Types as Lattices.
SIAM Journal of Computing 5(3):522-578, September, 1976.

[22] Solomon, M.
Modes, Values and Expressions.
In *Second Annual ACM Symposium on Principles of Programming Languages*, pages 149-159. ACM, 1975.
Includes a proof that recursive types can be compared (but requires declarations, and does not discuss higher order functions).

[23] Solomon, M.
Type Definitions with Parameters.
In *Fifth Annual ACM Symposium on Principles of Programming Languages.* ACM, January, 1978.
Extended abstract, shows that restrictions are needed for recursively defined types.

[24] Dennis, J.B. et al.
VIM-VAL Manual.
In progress.