



massachusetts institute of technology — artificial intelligence laboratory

---

# A Virtual Machine for a Type-omega Denotational Proof Language

Teodoro Arvizo III

AI Technical Report 2002-004

June 2002

**A Virtual Machine for a  
Type- $\omega$  Denotational Proof Language**

by

Teodoro Arvizo III

Submitted to the  
Department of Electrical Engineering and Computer Science  
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and  
Computer Science

and

Master of Engineering in Electrical Engineering and  
Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

© Teodoro Arvizo III, 2002. All rights reserved.

Certified by: Howard E. Shrobe  
Principal Research Scientist  
Thesis Supervisor

Accepted by: Arthur C. Smith  
Chairman, Department Committee on Graduate Theses

# A Virtual Machine for a Type- $\omega$ Denotational Proof Language

by  
Teodoro Arvizo III

Submitted to the  
Department of Electrical Engineering and Computer Science on May  
23, 2002, in partial fulfillment of the requirements for the degrees of  
Bachelor of Science in Electrical Engineering and Computer Science  
and  
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

In this thesis, I designed and implemented a virtual machine (VM) for a monomorphic variant of Athena, a type- $\omega$  denotational proof language (DPL). This machine attempts to maintain the minimum state required to evaluate Athena phrases. This thesis also includes the design and implementation of a compiler for monomorphic Athena that compiles to the VM. Finally, it includes details on my implementation of a read-eval-print loop that glues together the VM core and the compiler to provide a full, user-accessible interface to monomorphic Athena. The Athena VM provides the same basis for DPLs that the SECD machine does for pure, functional programming and the Warren Abstract Machine does for Prolog.

Thesis Supervisor: Howard E. Shrobe  
Title: Principal Research Scientist

## Acknowledgments

I would like to extend the warmest thanks to the researchers of the Dynamic Languages Group of the Artificial Intelligence Laboratory. Greg Sullivan and Jonathan Bachrach, thank you for your advice and help. I would especially like to thank Kostas Arkoudas for the opportunity to work on this project, the countless hours spent discussing and answering my questions regarding Athena, and for his helpful suggestions while completing this research.

I would also like to thank the Course VI Departmental Secretary, Anne Hunter, for all of her advice, suggestions, and last-minute required signatures. She helps makes MIT a better place.

Kate, thank you for your love and support.

# Contents

<b>1</b>	<b>Overview</b>	<b>8</b>
1.1	Goals . . . . .	9
<b>2</b>	<b>Athena</b>	<b>10</b>
2.1	Values in Athena . . . . .	11
2.1.1	Basic Values . . . . .	11
2.1.2	Symbols and Terms . . . . .	12
2.1.3	Propositions . . . . .	13
2.1.4	Equality Testing of Values . . . . .	14
2.2	Athena Semantics . . . . .	15
2.2.1	Expressions . . . . .	15
2.2.2	Deductions . . . . .	21
2.2.3	Patterns . . . . .	25
2.3	Athena Directives . . . . .	28
2.3.1	Introducing New Sorts and Symbols . . . . .	28
2.3.2	Numerals . . . . .	31
2.3.3	Other Directives . . . . .	32
2.3.4	Sort Checking . . . . .	32
<b>3</b>	<b>The Virtual Machine</b>	<b>35</b>
3.1	The Stacks . . . . .	36
3.1.1	The Initial Environment Stack . . . . .	36
3.1.2	Sorts and Symbols . . . . .	37
3.2	Operations and Transitions . . . . .	38
3.2.1	Value Loading . . . . .	38
3.2.2	Lists and List Manipulation . . . . .	40
3.2.3	Closure Creation . . . . .	41
3.2.4	Application . . . . .	41
3.2.5	Deductions and Method Application . . . . .	43
3.2.6	Recursive Closures and Applications . . . . .	44

3.2.7	Select . . . . .	45
3.2.8	While . . . . .	46
3.2.9	Try and the Error Transitions . . . . .	47
3.2.10	Substitutions . . . . .	48
3.2.11	Propositions . . . . .	49
3.2.12	Store Access and Manipulation . . . . .	50
3.2.13	Assumption Base Access . . . . .	51
3.2.14	Tests . . . . .	52
3.3	Pattern Matching Operations and Transitions . . . . .	55
3.3.1	Simple Patterns . . . . .	56
3.3.2	Bracket Patterns . . . . .	58
3.3.3	Compound Patterns . . . . .	59
3.3.4	Match-Fail Transitions . . . . .	62
<b>4</b>	<b>Compilation and Code</b>	<b>63</b>
4.1	The Compiler . . . . .	64
4.1.1	Expressions . . . . .	65
4.1.2	Deductions . . . . .	71
4.1.3	Patterns . . . . .	76
4.2	Primitive Methods . . . . .	80
4.3	Primitive Functions . . . . .	85
4.3.1	Math and Internal . . . . .	89
<b>5</b>	<b>Design and Implementation</b>	<b>92</b>
5.1	Design . . . . .	92
5.1.1	Errors and Error Handling . . . . .	92
5.1.2	Closure Application . . . . .	93
5.1.3	Sufficiently Powerful Operations . . . . .	93
5.1.4	The Compiler . . . . .	94
5.1.5	The Backup Stack $B$ . . . . .	94
5.2	Implementation . . . . .	95
5.2.1	Class Hierarchy . . . . .	95
5.2.2	Lists . . . . .	97
5.2.3	Blurring of Types . . . . .	97
5.2.4	Numbers and Meta-Identifiers . . . . .	98
5.2.5	Recursive Closures and Circular Environments . . . . .	98
5.3	An Example Deduction . . . . .	99
5.4	Future Work . . . . .	100
5.4.1	Stack Optimizations . . . . .	100
5.4.2	Transition Optimizations . . . . .	101

5.4.3	Pattern Matching Optimizations . . . . .	101
5.4.4	Full Athena Implementation . . . . .	102
5.4.5	Garbage Collection . . . . .	102
5.4.6	Alternative VM Implementation . . . . .	103
<b>6</b>	<b>Conclusion</b>	<b>104</b>

# List of Figures

2.1	Syntax of Athena Phrases . . . . .	16
2.2	Syntax of Athena Directives . . . . .	29



# Chapter 1

## Overview

This thesis describes the specification of a virtual machine (VM) for a type- $\omega$  denotational proof language (DPL) called Athena [Ark01]. The core of the machine, its structure and state transitions, provide the basis for type- $\omega$  DPLs. Hence, this machine could be used for the implementation for any type- $\omega$  DPL, but this thesis will treat Athena in particular.

Athena, like most type- $\omega$  DPLs, is a combination of a natural-deduction language and a functional programming language, allowing both for logical inference and for conventional computation. It was originally developed by Konstantine Arkoudas, and descriptions of the semantics, theory, and uses of Athena are detailed in [Ark99a] and [Ark99b]. Type- $\omega$  DPLs are specifically discussed in [Ark01]. The syntax and semantics of Athena, as given in Chapter 2, are merely the kernel of Athena, the essential subset in terms of which the full Athena language could be defined.

The VM that is presented by this thesis implements a monomorphic variant of Athena. The full specification of polymorphic Athena is given in [Ark99b], but the details of monomorphic Athena are specified in Section 2.3, with the introduction of sorts and symbols. In general, polymorphism is a powerful tool and can be used to generalize and reuse theorems and code; polymorphic sorts and symbols can be viewed as templates for the creation of monomorphic instances. In terms of logical completeness, the power of monomorphic Athena is the same — nothing is lost by removing polymorphism, other than convenience. However, a great deal is simplified in the implementation, as polymorphism requires and adds considerable complexity if it is to be handled efficiently. These efficiencies are orthogonal to the design and

implementation of the virtual machine. For these reasons, this research does not address polymorphic Athena.

## 1.1 Goals

The research summarized in this thesis consists of:

- A virtual machine specification, namely, a mathematical definition of a set of structures for describing the state of an Athena program, as well as a list of low-level operations with their associated transitions to manipulate the machine structures (Chapter 3).
- A compiler that takes as input either a file or a read-evaluate-print loop and produces an instruction stream for the VM. The code produced can then be executed by the VM interpreter. Also of importance is the code for Athena's primitive functions and methods (Chapter 4).
- Additional details of my design and implementation of the VM interpreter and compiler, with emphasis toward that which other researchers might find interesting (Chapter 5). Also included are an example deduction and possible future work in implementing the VM more efficiently.
- Conclusion and summary.

To these ends, Athena must first be described in detail.

## Chapter 2

# Athena

Athena is a type- $\omega$  DPL for multi-sorted first-order logic [Ark01]. As a functional programming language, Athena provides higher-order functions and has a lexically scoped environment and call-by-value semantics. Athena is an impure functional language, like Scheme, as state is available via cells. (Athena does not have unrestricted destructive assignment, however.) Athena also has powerful data structures and pattern matching capabilities similar to (but considerably more sophisticated than) those of ML.

Athena is a deductive framework that allows for the creation of universes of discourse, the concise description of their vocabulary, and the explicit assertion of their axioms. As a deductive framework, Athena provides a medium for the formulation, deduction, and proof of theorems with respect to their relevant universe of discourse.

The most distinguishing aspect of Athena is the use of assumption base semantics. Since Athena is a DPL, every deduction (or expression, for that matter) is evaluated with respect to an assumption base. Evaluating a deduction is equivalent to checking the validity of the deduction. Type- $\omega$  DPLs require that deductions be syntactically distinct from expressions; deductions perform inference, whereas expressions perform unrestricted computation. As a result, Athena can be used both as a programming language and as a proof checker/theorem prover with strong soundness guarantees.

Interaction with Athena usually occurs via a read-evaluate-print loop, also known as a read-eval-print loop or repl. Athena presents a prompt, the user enters phrases or directives, and Athena replies. This chapter is devoted to specifying exactly what Athena accepts as input and returns as output.

## 2.1 Values in Athena

In Athena, **values** are the result of **expressions** and **deductions**. Collectively, expressions and deductions are referred to as **phrases**. A given phrase may evaluate to a value. Discounting non-determinism, there will be at most one value; there will not be a value if the phrase produces an error or fails to terminate. A deduction, when it produces a value, returns a **proposition** that has been proved relative to the assumption base in which the deduction took place. A proved proposition is also called a **theorem**. The syntax of expressions and deductions is shown in Figure 2.1 (page 16), and the semantics is detailed in Section 2.2.

### 2.1.1 Basic Values

The **unit value** is a special value distinct from all other values. The unit value is the return value of some basic Athena expressions, most notably **set!** and **while**. The expression `()` also evaluates to the unit value, which is displayed by Athena as `()`.

A **character** represents an ASCII character in the range 0–127. When entered directly, they begin with a `'` character, usually followed by the ASCII character they represent, e.g. `'a`, `'B`, and `''`. One way to enter a non-printable character is to specify the ASCII number in decimal with the addition of a backslash, such as `'\000`, although there are other ways. Strings in Athena are represented merely as lists of characters.

A **list** is just that, a list of values. The empty list is both written and displayed as `[]`, and nonempty lists can be created by listing the values within brackets: `[V1 V2 ... Vn]`. Lists can contain other lists, since lists are values. Lists are of finite size; however, they can be used to create potentially infinitely sized structures, for example, using **thunks** or **cells**.

A **cell** is a value that contains another value. It acts as memory into which other values can be placed. A cell is created by giving it an initial value. Cells (and only cells) can also be modified by **set!**. Note that this is different from other functional languages, such as Scheme, which allow for unrestricted destructive assignment. A cell can contain other cells, since cells are values.

A **function**, or **function closure**, is an object that takes a number of other values as arguments,  $V_1, V_2, \dots, V_n$  where  $n \geq 0$ , and returns a value and a store as the result.<sup>1</sup> The **store** can be thought of as an

---

<sup>1</sup>Both functions and methods, instead of returning a value, might instead produce

infinite list of cells, or a mapping of cells to values. Function semantics is discussed further in Section 2.2.1. Functions are higher-order — they may take other functions as arguments and may produce functions as results.

A **method**, or **method closure**, is an object that represents a deductive process. It, too, takes other values as arguments,  $V_1, V_2, \dots, V_n$  where  $n \geq 0$ , however it always returns a proposition as a value, as well as a store.<sup>1</sup> Semantically, both function and method closures also have as arguments an **assumption base** and a store; these are discussed further in Section 2.2.1. Methods can, of course, have other methods as arguments.

## 2.1.2 Symbols and Terms

A **variable** is a type of term, written as `(var I)`, or `?I` for short, for a given identifier  $I$ . (What exactly constitutes an identifier is discussed in Section 2.2.)

A **symbol**, also known as a **function symbol**, has an arity  $n \geq 0$  associated with it. Symbols are used to make terms from other terms.

A **term** is defined as follows:

- Every variable is a term.
- Every **constant symbol** is a term. Constant symbols have arity  $n = 0$ , and are also called **nullary symbols** or **nullary function symbols**.
- If  $f$  is a symbol of arity  $n > 0$  and  $t_1 \dots t_n$  are terms, then  $(f t_1 \dots t_n)$  is a term. Terms of this type are also called **applications** or **function symbol applications**.

Mathematically, terms of the above form are referred to as Herbrand terms. Note that the use of the word “function” in the above definition has nothing to do with the value-type function; here, “function” refers to mathematics, not computer science.

Terms are often associated with a **sort**. Sorts are basically the unit of currency within the universe of discourse, and are discussed later in Section 2.3.1. Only terms that are *well-sorted* are legal in Athena; an object that is not well-sorted is called *ill-sorted*. What constitutes a well-sorted term is discussed in Section 2.3.4.

Athena provides some terms and sorts by default. There are special terms of sort `Ide` called **meta-identifiers**. These are written as an error or otherwise fail to terminate. That is, they may fail to return with a value.

(`meta-id`  $I$ ), or  $'I$  for short, and they are nullary symbols (and thus terms) of sort `Ide`. Athena also provides the constant terms `true` and `false` of sort `Boolean`.

Variables, symbols, and (well-sorted) terms are all values in Athena. Meta-identifiers are values as well, since they are terms.

A **substitution** is an Athena value that represents a (possibly empty) mapping of variables to terms. The empty substitution is written as  $\{\}$ . Formally, a substitution  $\theta$  is a function mapping finitely many variables  $v_i$  to terms  $t_i$ , and every other variable to itself:

$$\{v_1 \mapsto t_1 = \theta(v_1), \dots, v_n \mapsto t_n = \theta(v_n)\} \quad (v_i \neq t_i)$$

The function  $\bar{\theta}$  (called the lift of  $\theta$ ) is defined with respect to  $\theta$ , mapping terms to terms. For a given variable  $v$ , constant function symbol  $f_c$ , and function symbol  $f$ ,  $\bar{\theta}$  is defined as follows:

$$\begin{aligned} \bar{\theta}(v) &= \theta(v) \\ \bar{\theta}(f_c) &= f_c \\ \bar{\theta}((f \ t_1 \ \dots \ t_n)) &= (f \ \bar{\theta}(t_1) \ \dots \ \bar{\theta}(t_n)) \end{aligned}$$

In Athena, substitutions can be applied to terms and lists of terms; effectively, they act as lifts. Substitutions can also be applied to propositions and lists of propositions; this is discussed in the next section.

### 2.1.3 Propositions

A **proposition** is defined as follows:

- Every term is a proposition. Propositions of this form are also called **atoms**.
- If  $P_1$  and  $P_2$  are propositions, then so are (`not`  $P_1$ ), (`and`  $P_1$   $P_2$ ), (`or`  $P_1$   $P_2$ ), (`if`  $P_1$   $P_2$ ), and (`iff`  $P_1$   $P_2$ ). Propositions of these forms are called, respectively, **negations**, **conjunctions**, **disjunctions**, **conditionals**, and **biconditionals**. As a whole, they are called **logical propositions**.
- If  $P$  is a proposition, then so are (`forall`  $v$   $P$ ) and (`exists`  $v$   $P$ ), for any variable  $v$ . Propositions of these two forms are called, respectively, **universal** and **existential quantifications**. As a whole, they are called **quantified propositions**.  $P$  is also referred to as the **body** of the quantified proposition, and  $v$  is referred to as the **quantified variable**.

The identifiers `not`, `and`, `or`, `if`, and `iff` are called **propositional constructors**, or **prop-cons** for short. The identifiers `forall` and `exists` are called **quantifiers**.

Terms that are atoms must either be of sort `Boolean` or coercible to sort `Boolean`. Thus, it is possible that a proposition in one of the above forms may be constructed out of a term that is not coercible to sort `Boolean`. Or, a proposition might attempt to coerce a variable to be of more than one sort. Such propositions are ill-sorted; only well-sorted propositions are legal in Athena. What constitutes a well-sorted proposition is discussed in Section 2.3.4.

An occurrence of a quantified variable within a quantified proposition is said to be **bound**. Otherwise, the occurrence is **free**. The set of **free variables** for a proposition  $P$  are those variables that have free occurrences within  $P$ ; likewise, the set of **bound variables** are those having a bound occurrence within  $P$ . Clearly, these sets are not necessarily disjoint. A given proposition can be **freshly renamed** by replacing all bound variables and their occurrences with variables that have been newly introduced. (Such newly introduced variables are guaranteed not to be in any other proposition or term, because up until the renaming they did not exist.)

Propositional constructors, quantifiers, and (well-sorted) propositions are all values in Athena. Substitutions, when applied to propositions or a list of propositions, only affect free variables, and all of the results must also be well-sorted.

#### 2.1.4 Equality Testing of Values

The existence of quantified propositions also introduces a blurring in the notion of value equality. Are `(forall ?x (and ?x true))` and `(forall ?y (and ?y true))` the same value? What is different about them? Does that change any possible interpretations about these propositions?

For propositions, there are two notions of equality. A **literal equality** test would determine that the two propositions are different. An **alphabetical equivalence** test, however, would determine that they are the same: for a given proposition, all bound variables (`?x` and `?y` in the example) could be renamed to some canonical variables and the results could be compared for literal equality. An example of a canonical representation of the above propositions would be `(forall ?v1 (and ?v1 true))`. In Athena, propositions are always tested for equality using an alphabetical equivalence test.

The existence of function and method closures also causes a prob-

lem. What does it mean to compare two closures for equality? Does it mean they have the exact same program code, or that they do the same thing? Computation theory proves that deciding whether two programs are equivalent is undecidable. Athena simply designates that it is an error if closures of the same type (both functions or both methods) are compared for equality.

## 2.2 Athena Semantics

Athena accepts phrases, performs some computation depending on the phrase entered, and normally returns a value. Figure 2.1 gives the core Athena syntax and specifies all phrases  $F$  that Athena recognizes. As per standard regular expression syntax,  $V^*$  denotes zero or more occurrences of  $V$ , and  $V^+$  denotes one or more occurrences of  $V$ . An identifier  $I$  is any printable character (ASCII 32–127) that is not a reserved keyword, whose first character is not one of `?!$'[( ,` and which does not contain any of the characters `)]"; .` Additionally `'I` is an abbreviation for `(meta-id I)`, `'I` is a character, `?I` is an abbreviation for `(var I)`, and `(!E F*)` is an abbreviation for `(apply-method E F*)`. A string  $S$  is normally written as text between quotation marks, such as `"This is a list of characters."`

The following sections will describe the semantics of each of the phrases and patterns that Athena uses. Each phrase is evaluated with respect to an environment  $\rho$ , a store  $\sigma$ , and an assumption base  $\beta$ . An environment  $\rho$  represents the current values of all bound identifiers. A store  $\sigma$  represents the values held by cells. An assumption base  $\beta$  represents those propositions that hold during the evaluation of a phrase. The result of evaluating a phrase, assuming termination and non-determinism, will be a value  $V$  and a store  $\sigma'$ . In general,  $\sigma = \sigma'$  unless the phrase uses `cell` or `set!`.

### 2.2.1 Expressions

- **Unit:** The value of the expression `()` is always the unit value, regardless of  $\rho$ ,  $\sigma$ , and  $\beta$ .
- **Identifier:** The value of the expression  $I$  depends only on  $\rho$ . If an identifier  $I$  is bound in the current environment  $\rho$  to a value  $V$ , then the value of the identifier is  $V$ . It is an error if  $I$  is not bound.



- **String:** A string  $S$  is merely a list of characters, regardless of  $\rho$ ,  $\sigma$ , and  $\beta$ .
- **Character:** An expression of the form ‘ $I$  is a character assuming  $I$  is a legal identifier for a character, regardless of  $\rho$ ,  $\sigma$ , and  $\beta$ .  $I$  must be of the form:
  - $\backslash n$ , where  $0 \leq n \leq 127$ .
  - $\backslash^X$ , where  $X$  is one of  $A\dots Z$ ,  $@$ ,  $[$ ,  $]$ ,  $\wedge$ , or  $\_$ , representing the control characters in ASCII 1–26, 0, 27, 29, 30, and 31, respectively.
  - $X$ , where  $X$  is a printable character but not  $\backslash$ .

```

E ::= () | I | S | 'I | (var I) | (meta-id I) | (cell F) |
      (ref E) | (set! E F) | (function (I*) E) | (E F*) |
      [F*] | (method (I*) D) | (check (F E)*) |
      (match F ( $\pi$  E)*) | (let ((I F)*) E) |
      (letrec ((I F)*) E) | (begin F+) | (while F1 F2) |
      (& F+) | (|| F+)

D ::= (apply-method E F*) | (assume F D) |
      (suppose-absurd F D) | (dcheck (F D)*) |
      (dmatch F ( $\pi$  D)*) | (dlet ((I F)*) D) |
      (dletrec ((I F)*) D) | (try D+) | (dbegin F* D) |
      (E BY D) | (generalize-over E D) | (pick-any I D) |
      (with-witness E F D) | (pick-witness I F D)

F ::= E | D

 $\pi$  ::= _ | () | I | S | 'I | [ $\pi^*$ ] | (var I) | (meta-id I) |
      (val-of I) | (list-of  $\pi_1$   $\pi_2$ ) | ( $\pi$   $\pi^+$ ) | (bind I  $\pi$ ) |
      (some-atom I) | (some-function I) | (some-list I) |
      (some-method I) | (some-prop I) | (some-prop-con I) |
      (some-quant I) | (some-sub I) | (some-symbol I) |
      (some-term I) | (some-var I)

```

Figure 2.1: Syntax of Athena Phrases

- `\\`, representing the backslash character.
  - `\space`, representing a space.
  - or
  - `\x`, where `x` is one of `a`, `b`, `t`, `n`, `v`, `f`, or `r`, representing the standard C escape characters.
- **Variable:** The value of `?I` is the variable `?I`, regardless of  $\rho$ ,  $\sigma$ , and  $\beta$ . `?I` is actually shorthand for `(var I)`.
  - **Meta-Identifier:** The value of `'I` is the nullary symbol `'I` of the built-in sort `Ide`, regardless of  $\rho$ ,  $\sigma$ , and  $\beta$ . `'I` is actually shorthand for `(meta-id I)`.
  - **Cell:** The value of `(cell F)` in  $\rho$ ,  $\sigma$ , and  $\beta$  is a cell that initially contains the value obtained by evaluating `F` in  $\rho$ ,  $\sigma$ , and  $\beta$ . The store returned by `(cell F)` is the  $\sigma'$  obtained by evaluating `F` extended to include this new cell.
  - **Cell References:** The value of `(ref E)` is the contents of the cell that `E` evaluates to in  $\rho$ ,  $\sigma$ , and  $\beta$ . It is an error if `E` does not evaluate to a cell. The resulting store is that which was returned by evaluating `E`.
  - **Assignment:** The *effect* of `(set! E F)` is to replace the contents of the cell that is the result of `E` (in  $\rho$ ,  $\sigma$ , and  $\beta$ ) with the result of `F` (within the store  $\sigma'$  as returned by `E`). The value returned by `set!` is the unit value, and the resulting store is the store returned by evaluating `F`.
  - **Function:** The value of `(function (I1 ... In) E)` in  $\rho$ ,  $\sigma$ , and  $\beta$  is a function closure. The closure, when applied, accepts  $n$  argument values  $V_1, \dots, V_n$  and the current store  $\sigma'$  and assumption base  $\beta'$  and returns the value and the store that are obtained by evaluating `E` in the environment  $\rho$  extended with bindings mapping each  $I_j$  to each  $V_j$ , for  $0 \leq j \leq n$ , the store  $\sigma'$ , and the assumption base  $\beta'$ .  
 Note that `E` is evaluated in  $\sigma'$  and  $\beta'$  to take into account any changes that might have occurred in between when the function closure was declared and when it was applied.
  - **Application:** To evaluate an expression of the form `(E F1 ... Fn)`, first evaluate `E` with respect to  $\rho$ ,  $\sigma$ , and  $\beta$  to obtain  $V$  and  $\sigma'$ . The evaluation of the entire expression depends on the type of  $V$ .

- If  $V$  is a function closure, then each  $F_i$  is evaluated, in order from  $0 \leq i \leq n$ , with respect to  $\rho$ ,  $\sigma_{i-1}$ , and  $\beta$  to obtain a value  $V_i$  and  $\sigma_i$ . (Note that  $\sigma_0 = \sigma'$ .) The final result is that of invoking the function closure  $V$  with the values  $V_1, \dots, V_n$ , the store  $\sigma_n$ , and assumption base  $\beta$ .
- If  $V$  is a function symbol  $f$ , then each  $F_1, \dots, F_n$  is evaluated in  $\sigma_{i-1}$  and  $\beta$  in order to obtain values  $V_1, \dots, V_n$  and store  $\sigma_n$ . If each  $V_i$  is a term  $t_i$ , then the result of the expression is the term  $(f t_1 \dots t_n)$  *if* this term is well-sorted. Otherwise, the result is an error. Note that if  $f$  is a nullary function symbol, then  $(f)$  evaluates to  $f$ .
- If  $V$  is a propositional constructor  $con$ , then each  $F_1, \dots, F_n$  is evaluated in  $\sigma_{i-1}$  and  $\beta$  in order to obtain values  $V_1, \dots, V_n$  and store  $\sigma_n$ . If each  $V_i$  is a proposition  $P_i$ , then the result of the expression is the proposition  $(con P_1 \dots P_n)$  *if* this proposition is well-sorted *and* the arity of the prop-con is equal to  $n$ . Otherwise, the result is an error.
- If  $V$  is a quantifier  $q$ , then each  $F_1, \dots, F_n$  is evaluated in  $\sigma_{i-1}$  and  $\beta$  in order to obtain values  $V_1, \dots, V_n$  and store  $\sigma_n$ . If  $n = 2$ ,  $V_1$  is a variable  $v$ , and  $V_2$  is a proposition  $P$ , then the result is the proposition  $(q v P)$ , *if* this proposition is well-sorted. Otherwise, the result is an error.
- If  $V$  is a substitution  $\theta$ , first verify that  $n = 1$ . It is an error if the substitution has more than one argument. Evaluate  $F_1$  in  $\rho$ ,  $\sigma'$ , and  $\beta$  to get  $V_1$  and  $\sigma_1$ .
  - \* If  $V_1$  is a term  $t$ , then the result is the term  $\bar{\theta}(t)$  *if* this term is well-sorted. It is an error if it is ill-sorted.
  - \* If  $V_1$  is a list of terms  $[t_1 \dots t_n]$ , then the result is the list of terms  $[\bar{\theta}(t_1) \dots \bar{\theta}(t_n)]$  *if* each term  $\bar{\theta}(t_i)$  is well-sorted. It is an error if some  $\bar{\theta}(t_i)$  is ill-sorted.
  - \* If  $V_1$  is a proposition  $P$ , then the result is that of applying the  $\bar{\theta}$  to all the atoms of proposition  $P'$ , where  $P'$  is a fresh renaming of  $P$ , *if* the proposition that results is well-sorted. It is an error if it is ill-sorted.
  - \* Likewise, if  $V_1$  is a list of propositions, then the result is that of applying  $\bar{\theta}$  to a fresh renaming of each proposition *if* each resulting proposition is well-sorted. It is an error if some resulting proposition is ill-sorted.
- Otherwise, if  $V$  is not any of the above, the result is an error.

- **List:** The value of  $[F_1 \dots F_n]$ ,  $n \geq 0$ , in environment  $\rho$ , store  $\sigma$ , and assumption base  $\beta$ , is the list of values  $[V_1 \dots V_n]$  and the store  $\sigma_n$ , where each  $V_i$  and  $\sigma_i$  is the result of evaluating  $F_i$  with respect to  $\rho$ ,  $\sigma_{i-1}$ , and  $\beta$ . (Again,  $\sigma_0 = \sigma$ .)
- **Method:** The value of  $(\text{method } (I_1 \dots I_n) D)$  in  $\rho$ ,  $\sigma$ , and  $\beta$  is a method closure. The closure, when applied as part of a method application, accepts  $n$  argument values  $V_1, \dots, V_n$  and the current store  $\sigma'$  and assumption base  $\beta'$  and returns the value and the store that are obtained by evaluating  $D$  in the environment  $\rho$  extended with bindings mapping each  $I_j$  to each  $V_j$ , for  $0 \leq j \leq n$ , store  $\sigma'$ , and assumption base  $\beta'$ .

Note that  $D$  is evaluated in  $\sigma'$  and  $\beta'$  to take into account any changes that might have occurred in between when the method closure was declared and when it was applied.

- **Check:** A check expression is of the form  $(\text{check } (F_1 E_1) \dots (F_n E_n))$ . Setting  $\sigma_0 = \sigma$ , start with  $i = 1$ :  $F_i$  is evaluated in  $\rho$ ,  $\sigma_{i-1}$ , and  $\beta$ , resulting in a value  $V_i$  and store  $\sigma_i$ . If  $V_i$  is the symbol `true`, then the result of the check expression is the value of  $E_i$  in  $\rho$  and  $\sigma_i$ . If  $V_i$  is not the symbol `true`, then increment  $i$  and repeat, as long it remains  $\leq n$ . If  $V_n$  is also not the symbol `true`, then the result of the check expression is an error.

$F_n$  is allowed to be the reserved keyword `else`; if this case is reached, the result of the check expression is  $E_n$  evaluated in  $\rho$ ,  $\sigma_{n-1}$ , and  $\beta$ . If  $n = 0$ , then the result of the check expression is an error.

- **Match:** A match expression is of the form  $(\text{match } F (\pi_1 E_1) \dots (\pi_n E_n))$ . The semantics of pattern matching is detailed in Section 2.2.3; however, the semantics for the match expression itself is straightforward.  $F$  is evaluated in  $\rho$ ,  $\sigma$ , and  $\beta$ , to get value  $V$  and  $\sigma'$ .  $V$  is matched against each  $\pi_i$  with respect to  $\rho$  in turn. For the first  $\pi_i$  that  $V$  is successfully matched against, the result of the match expression is  $E_i$  evaluated in  $\sigma'$ ,  $\beta$ , and  $\rho'$ , where  $\rho'$  is  $\rho$  extended with pattern variables from  $\pi_i$ . If no pattern  $\pi_i$  matches successfully (or if  $n = 0$ ), then the result is an error.

- **Let:** A let expression is of the form  $(\text{let } ((I_1 F_1) \dots (I_n F_n)) E)$ . With  $\rho_0 = \rho$  and  $\sigma_0 = \sigma$ , each  $F_i$  is evaluated in  $\rho_{i-1}$ ,  $\sigma_{i-1}$ , and  $\beta$  to get a value  $V_i$  and

store  $\sigma_i$ . In addition,  $\rho_i$  is defined to be  $\rho_{i-1}$  extended with the binding of identifier  $I_i$  to the value  $V_i$ . The value returned by the let expression is the result of the evaluation of  $E$  in environment  $\rho_n$ , store  $\sigma_n$ , and assumption base  $\beta$ .

- **Letrec:** A letrec expression is of the form  $(\text{letrec } ((I_1 F_1) \dots (I_n F_n)) E)$ . Effectively, this is equivalent to

```
(let ((I1 (cell ())) ... (In (cell ())))
  (begin
    (set! I1 F'1)
    ⋮
    (set! In F'n)
    E')).
```

where  $E'$  and  $F'_j$  are the original  $E$  and  $F_j$  in which free occurrences of  $I_j$  have been replaced with  $(\text{ref } I_j)$ . Free occurrences of an identifier are those occurrences that are not under the scope of another binding phrase, e.g. a closure application.

- **Begin:** A begin expression is of the form  $(\text{begin } F_1 \dots F_n)$ , for some  $n > 0$ . Setting  $\sigma_0 = \sigma$ , each  $F_i$  is evaluated in order with respect  $\rho$ ,  $\sigma_{i-1}$ , and  $\beta$ , resulting in value  $V_i$  and  $\sigma_i$ . The result of the begin expression is value  $V_n$  and store  $\sigma_n$ .
- **While loop:** A while expression is of the form  $(\text{while } F_1 F_2)$ . Initially,  $F_1$  is evaluated in  $\rho$ ,  $\sigma$ , and  $\beta$  to obtain  $V_1$  and  $\sigma_1$ . The expression then enters the while loop. If  $V_1$  is the symbol **false**, then the result of the while expression is the unit value and the store  $\sigma_1$ . If  $V_1$  is the symbol **true**, then  $F_2$  is evaluated in  $\rho$ ,  $\sigma_1$ , and  $\beta$ , resulting in  $V_2$  and  $\sigma_2$ ;  $F_1$  is then re-evaluated, but in  $\rho$ ,  $\sigma_2$ , and  $\beta$ , and the loop continues by testing the new  $V_1$ . It is an error if  $V_1$  is neither **false** nor **true**.
- **Short-circuit, logical And:** A logical and is of the form  $(\& F_1 \dots F_n)$ ,  $n > 0$ . Starting with  $i = 1$ ,  $F_i$  is evaluated in  $\rho$ ,  $\sigma_{i-1}$ , and  $\beta$ , to get value  $V_i$  and  $\sigma_i$ . If  $V_i$  is the symbol **false**, the result of the logical and expression is the value **false** and the store  $\sigma_i$ . If  $V_i$  is the symbol **true**, increment  $i$  and evaluate the next phrase; if  $V_n$  is **true**, then the result of the logical and expression is the value **true** and the store  $\sigma_n$ . It is an error if  $V_i$  is neither **true** nor **false**.

- **Short-circuit, Logical Or:** A logical or is of the form  $(|| F_1 \dots F_n)$ ,  $n > 0$ . Starting with  $i = 1$ ,  $F_i$  is evaluated in  $\rho$ ,  $\sigma_{i-1}$ , and  $\beta$ , to get value  $V_i$  and  $\sigma_i$ . If  $V_i$  is the symbol **true**, the result of the logical or expression is the value **true** and the store  $\sigma_i$ . If  $V_i$  is the symbol **false**, increment  $i$  and evaluate the next phrase; if  $V_n$  is **false**, then the result of the logical or expression is the value **false** and the store  $\sigma_n$ . It is an error if  $V_i$  is neither **true** nor **false**.

## 2.2.2 Deductions

- **Method Application:** A method application is of the form  $(!E F_1 \dots F_n)$ ; the use of **!** is actually shorthand for **apply-method**. First,  $E$  is evaluated in environment  $\rho$ , store  $\sigma$ , and assumption base  $\beta$ , resulting in  $V$  and  $\sigma_0$ . It is an error if  $V$  is not a method closure or an Athena primitive method (Section 4.2). Setting  $\beta_0 = \beta$ , each  $F_i$  is evaluated in  $\rho$ ,  $\sigma_{i-1}$ , and  $\beta_{i-1}$  to yield value  $V_i$ , and store  $\sigma_i$ . If  $F_i$  is a deduction, then the assumption base  $\beta_i$  is set to be  $\beta_{i-1} \cup \{V_i\}$ ; otherwise, set  $\beta_i = \beta_{i-1}$ .<sup>2</sup> Apply the method  $V$  with the arguments  $V_1, \dots, V_n$ , store  $\sigma_n$  and assumption base  $\beta_n$  to get the conclusion  $V_c$  and store  $\sigma_c$ ; it is an error if the arity of the method is not equal to  $n$ . The result of the method application is the value  $V_c$  and the store  $\sigma_c$ .
- **Assume:** For  $(\text{assume } F D)$ ,  $F$  is evaluated in environment  $\rho$ , store  $\sigma$ , and assumption base  $\beta$  to yield value  $V_F$  and store  $\sigma_F$ . It is an error if  $V_F$  is not a proposition. Next,  $D$  is evaluated in  $\rho$ ,  $\sigma_F$ , and  $\beta \cup \{V_F\}$  to yield  $V_D$  and  $\sigma_D$ . The result of the assume deduction is the proposition  $(\text{if } V_F V_D)$  and store  $\sigma_D$ .
- **Suppose Absurd:** For  $(\text{suppose-absurd } F D)$ ,  $F$  is evaluated in environment  $\rho$ , store  $\sigma$ , and assumption base  $\beta$  to yield value  $V_F$  and store  $\sigma_F$ . It is an error if  $V_F$  is not a proposition. Next,  $D$  is evaluated in  $\rho$ ,  $\sigma_F$ , and  $\beta \cup \{V_F\}$  to yield  $V_D$  and  $\sigma_D$ . It is an error if  $V_D$  is not the proposition **false**. The result of the

<sup>2</sup>Technically, this is incorrect. In Athena, each  $F_i$  should be evaluated with respect to  $\rho$ ,  $\sigma_{i-1}$ , and  $\beta$ . Only after each phrase  $F_i$  has been evaluated are the conclusions of any deductions among  $F_1 \dots F_n$  added to the assumption base for the method application. As specified here, the ordering of deductions and the existence of intermediate conclusions may affect the result of the method application. This inconsistency remains because it was discovered late into the project, but it would not be difficult to fix.

suppose-absurd deduction is the proposition (**not**  $V_F$ ) and store  $\sigma_D$ .

- **Dcheck:** A dcheck deduction is of the form (**dcheck**  $(F_1 D_1) \dots (F_n D_n)$ ). Setting  $\sigma_0 = \sigma$ , start with  $i = 1$ :  $F_i$  is evaluated in  $\rho$ ,  $\sigma_{i-1}$ , and  $\beta$ , resulting in a value  $V_i$  and store  $\sigma_i$ . If  $V_i$  is the symbol **true**, then the result of the dcheck deduction is the value of  $D_i$  in  $\rho$ ,  $\sigma_i$ , and  $\beta$ . If  $V_i$  is not the symbol **true**, then increment  $i$  and repeat, as long it remains  $\leq n$ . If  $V_n$  is also not the symbol **true**, then the result of the dcheck deduction is an error.

$F_n$  is allowed to be the reserved keyword **else**; if this case is reached, the result of the dcheck deduction is  $D_n$  evaluated in  $\rho$ ,  $\sigma_{n-1}$ , and  $\beta$ . If  $n = 0$ , then the result of the dcheck deduction is an error.

- **Dmatch:** This deduction is of the form (**dmatch**  $F (\pi_1 D_1) \dots (\pi_n D_n)$ ). The semantics of pattern matching is detailed in Section 2.2.3; however, the semantics for the dmatch deduction itself is straightforward.  $F$  is evaluated in  $\rho$ ,  $\sigma$ , and  $\beta$  to get value  $V$  and store  $\sigma'$ . If  $F$  was a deduction that produced proposition  $P$ , then set  $\beta' = \beta \cup \{P\}$ ; otherwise, let  $\beta' = \beta$ .  $V$  is matched against each  $\pi_i$  with respect to  $\rho$  in turn. For the first  $\pi_i$  that  $V$  is successfully matched against, the result of the dmatch deduction is  $D_i$  evaluated in  $\sigma'$ ,  $\beta'$ , and  $\rho'$ , where  $\rho'$  is  $\rho$  extended with pattern variables from  $\pi_i$ . If no pattern  $\pi_i$  matches successfully (or if  $n = 0$ ), then the result is an error.

- **Dlet:** A dlet deduction is of the form (**dlet**  $((I_1 F_1) \dots (I_n F_n)) D$ ). With  $\rho_0 = \rho$ ,  $\sigma_0 = \sigma$ , and  $\beta_0 = \beta$ , each  $F_i$  is evaluated in  $\rho_{i-1}$ ,  $\sigma_{i-1}$ , and  $\beta_{i-1}$  to get a value  $V_i$  and store  $\sigma_i$ . If  $F_i$  was a deduction, then set  $\beta_i = \beta_{i-1} \cup \{V_i\}$ ; otherwise, set  $\beta_i = \beta_{i-1}$ . In addition,  $\rho_i$  is defined to be  $\rho_{i-1}$  extended with the binding of identifier  $I_i$  to the value  $V_i$ . The value returned by the dlet deduction is the result of the evaluation of  $D$  in environment  $\rho_n$ , store  $\sigma_n$ , and assumption base  $\beta_n$ .

- **Dletrec:** This deduction is of the form (**dletrec**  $((I_1 F_1) \dots (I_n F_n)) D$ ). Effectively, this is equivalent to

(**dlet**  $((I_1 (\text{cell } ())) \dots (I_n (\text{cell } ())))$ )

```

(dbegin
  (set! I1 F'1)
  ⋮
  (set! In F'n)
  D').

```

where  $D'$  and  $F'_j$  are the original  $D$  and  $F_j$  in which free occurrences of  $I_j$  have been replaced with  $(\text{ref } I_j)$ . Free occurrences of an identifier are those occurrences that are not under the scope of another binding construct, e.g. **function**, **let**, or a pattern variable.

- **Try:** A try deduction is of the form  $(\text{try } D_1 \dots D_n)$ ,  $n > 0$ . Setting  $\sigma_0 = \sigma$  and starting with  $i = 1$ ,  $D_i$  is evaluated in environment  $\rho$ , store  $\sigma_{i-1}$ , and assumption base  $\beta$ . If  $D_i$  concludes with  $V_i$  and  $\sigma_i$ , then that is the result of the try deduction. If, however, there was an error during the evaluation of  $D_i$ , then  $i$  is incremented and the next deduction is tried. It is an error if all  $n$  deductions fail.
- **Dbegin:** A **dbegin** deduction is of the form  $(\text{dbegin } F_1 \dots F_n D)$ . Setting  $\sigma_0 = \sigma$  and  $\beta_0 = \beta$ , each  $F_i$  is evaluated in order with respect  $\rho$ ,  $\sigma_{i-1}$ , and  $\beta_{i-1}$ , resulting in value  $V_i$  and store  $\sigma_i$ . If  $F_i$  is a deduction, set  $\beta_i = \beta_{i-1} \cup \{V_i\}$ ; otherwise,  $\beta_i = \beta_{i-1}$ . The result of the **dbegin** deduction is result of evaluating  $D$  in  $\rho$ ,  $\sigma_n$ , and  $\beta_n$ .
- **By:** A **by** deduction is of the form  $(E \text{ BY } D)$ .  $E$  is evaluated in environment  $\rho$ , store  $\sigma$ , and assumption base  $\beta$  to get value  $V_E$  and store  $\sigma_E$ .  $D$  is then evaluated in  $\rho$ ,  $\sigma_E$ , and  $\beta$ , to yield  $V_D$  and  $\sigma_D$ . If  $V_E$  and  $V_D$  are alphabetically equivalent, then the result of the **by** deduction is  $V_D$  and  $\sigma_D$ . It is an error if the two values are not equal.
- **Generalize Over:** This deduction is of the form  $(\text{generalize-over } E D)$ .  $E$  is evaluated in environment  $\rho$ , store  $\sigma$ , and assumption base  $\beta$ , resulting in value  $V_E$  and  $\sigma_E$ . It is an error if  $V_E$  is not a variable. Also, it is an error if the variable  $V_E$  is free in the assumption base  $\beta$  — that is, if the variable is free in any proposition  $P \in \beta$ . Next,  $D$  is evaluated in  $\rho$ ,  $\sigma_E$ , and  $\beta$ , resulting in  $V_D$  and  $\sigma_D$ . The result of the **generalize-over** deduction is  $(\text{forall } V_E V_D)$  and  $\sigma_D$ .



- **Pick Any:** This deduction is of the form (**pick-any**  $I D$ ). It is similar to generalize-over, except that it provides the variable via the identifier  $I$ . Define  $\rho'$  to be  $\rho$  extended with a binding of  $I$  to a fresh variable  $v$  — a variable that has never been before introduced, and thus does not exist in any proposition.  $D$  is then evaluated with respect to  $\rho'$ ,  $\sigma$ , and  $\beta$ , resulting in  $V_D$  and  $\sigma_D$ . The result of the pick-any deduction is (**forall**  $v V_D$ ) and store  $\sigma_D$ .

- **With Witness:** This deduction is of the form (**with-witness**  $E F D$ ). First,  $E$  is evaluated in environment  $\rho$ , store  $\sigma$ , and assumption base  $\beta$  to yield  $V_E$  and  $\sigma_E$ . It is an error if  $V_E$  is not a variable. Also, it is an error if the variable  $V_E$  is free in the assumption base  $\beta$ .

Next, evaluate  $F$  in  $\rho$ ,  $\sigma_E$ , and  $\beta$  to yield  $V_F$ ,  $\sigma_F$ . If  $F$  is a deduction, set  $\beta' = \beta \cup \{V_F\}$ ; otherwise, set  $\beta' = \beta$ . It is an error if  $V_F$  is not an existentially qualified proposition, e.g. of the form (**exists**  $x P$ ) for some  $x$  and  $P$ . Also, it is an error if  $V_F \notin \beta'$ .

Finally, evaluate  $D$  in  $\rho$ ,  $\sigma_F$ , and  $\beta' \cup \{P'\}$ , where  $P'$  is the proposition  $P$  from  $V_D$  in which free occurrences of  $x$  have been replaced by  $V_E$ , to yield  $V_D$  and  $\sigma_D$ . The result of the with-witness deduction is  $V_D$  and store  $\sigma_D$ , provided that the variable  $V_E$  does not occur free in  $V_D$ ; it is an error if the variable occurs free.

- **Pick Witness:** This deduction is of the form (**pick-witness**  $I F D$ ). It is similar to with-witness, except that it provides the variable via the identifier  $I$ . Define  $\rho'$  to be  $\rho$  extended with a binding of  $I$  to a fresh variable  $v$ .  $F$  is then evaluated with respect to  $\rho'$ ,  $\sigma$ , and  $\beta$ , resulting in  $V_F$  and  $\sigma'$ . If  $F$  is a deduction, set  $\beta' = \beta \cup \{V_F\}$ ; otherwise, set  $\beta' = \beta$ . It is an error if  $V_F$  is not an existentially qualified proposition, e.g. of the form (**exists**  $x P$ ) for some  $x$  and  $P$ . Also, it is an error if  $V_F \notin \beta'$ .

Finally, evaluate  $D$  in  $\rho'$ ,  $\sigma_F$ , and  $\beta' \cup \{P'\}$ , where  $P'$  is the proposition  $P$  from  $V_F$  in which free occurrences of  $x$  have been replaced by  $v$ , to yield  $V_D$  and  $\sigma_D$ . The result of the pick-witness deduction is  $V_D$  and store  $\sigma_D$ , provided that the variable  $v$  does not occur free in  $V_D$ ; it is an error if the variable occurs free.

### 2.2.3 Patterns

A pattern  $\pi$  has three basic forms, all of which are shown in Figure 2.1 (page 16). These are:

1. A **bracket pattern**, of the form  $[\pi_1 \dots \pi_n]$ . Bracket patterns match lists.
2. A **compound pattern**, of the form  $(\pi_1 \dots \pi_n)$ ,  $n > 1$ . Compound patterns are used to decompose terms and propositions.
3. A **simple pattern**, which includes all of the rest. Simple patterns are:
  - (a) Basic values such as the unit value, individual characters, variables, and meta-identifiers.
  - (b) Patterns using the keywords `_`, `val-of`, `list-of`, `bind`, and those of the form *some-type*.
  - (c) Function symbols, prop-cons, and quantifiers. These are called **pattern constants**.
  - (d) Identifiers that are neither keywords, symbols, prop-cons, or quantifiers. These are called **pattern variables**.

In addition, there are these pattern types:<sup>3</sup>

- A **list pattern** is a bracket pattern,  $(\text{list-of } \pi_1 \ \pi_2)$ ,  $(\text{some-list } I)$ , or  $(\text{bind } I \ \pi)$  where  $\pi$  is a list pattern.
- A **quantifier pattern** is the pattern constant `forall`, the pattern constant `exists`, or a pattern of the form  $(\text{some-quant } I)$ .

During the course of pattern matching, a value  $V$ , called the **discriminant**, is compared to a pattern  $\pi$  in the environment  $\rho$  with respect to a mapping  $\mu$ . The mapping  $\mu$  holds bindings from pattern variables earlier in the pattern to values;  $\mu$  initially starts out empty. The overall result will either be failure or a mapping  $\mu$  of all the pattern variables of  $\pi$  to values. If a given identifier is used multiple times in a pattern, it adds the constraint that both values are equal — unless the identifier is rebound later in the pattern via a `bind` or *some-type* pattern. When comparing propositions during the course of pattern matching, an alphabetical equivalence test is used.

---

<sup>3</sup>The full Athena language also provides a split pattern,  $(\text{split } \pi_1 \ \pi_2)$ , used to match lists, which this monomorphic Athena variant does not provide.

## Simple Patterns

- If  $\pi$  is `_`, the match is successful. The pattern `_` matches anything.
- If  $\pi$  is the unit value, a character, a variable, or a meta-identifier, then the match is successful if the discriminant  $V$  is the same value. Otherwise, the match fails.
- If  $\pi$  is of the form `(val-of I)`, then  $V$  is compared with the value that is bound to  $I$  in  $\rho$ . If the values are equal, the match succeeds; otherwise, the match fails. It is an error if  $I$  is not bound.
- If  $\pi$  is of the form `(list-of  $\pi_1$   $\pi_2$ )`, the match fails if  $V$  is not a list or if  $V$  is the empty list. Setting  $V$  to be the list of values  $[V_1 \dots V_n]$ ,  $n > 0$ , the match continues by matching  $[V_1 [V_2 \dots V_n]]$  against the bracket pattern  $[\pi_1 \pi_2]$  with respect to  $\rho$  and  $\mu$ .
- If  $\pi$  is of the form `(bind I  $\pi'$ )`, the discriminant  $V$  is matched against  $\pi'$ , with respect to  $\rho$  and  $\mu$ . If this match succeeds with the mapping  $\mu'$ , then the result of the bind pattern is the mapping  $\mu'$  extended with the binding of  $I$  to  $V$ . Otherwise, the match fails.
- If  $\pi$  is of the form `(some-type I)`, the discriminant  $V$  is tested to be of the specified type. If it is, the match succeeds with the mapping  $\mu$  extended with a binding of  $I$  to  $V$ ; otherwise, the match fails.
- If  $\pi$  is symbol, prop-con, or quantifier, the match succeeds if  $V$  is exactly that symbol, prop-con, or quantifier. Otherwise, the match fails.
- Finally,  $\pi$  could be an identifier  $I$ . If  $I$  has a binding to  $V_I$  in  $\mu$ , then the match is successful if the discriminant  $V$  is equal to  $V_I$ . If  $I$  does not have a binding in  $\mu$ , then the match succeeds with the mapping  $\mu$  extended with the binding of  $I$  to  $V$ . Otherwise, the match fails.

## Bracket Patterns

- If  $\pi$  is a bracket pattern, the match fails if  $V$  is not a list. For  $\pi$  of the form  $[\pi_1 \dots \pi_n]$ , and discriminant  $V$  of the form  $[V_1 \dots V_n]$ , the match continues by sequentially matching each  $\pi_i$  and  $V_i$ ; the

match fails if they of different length. Setting  $\mu_0 = \mu$ , start with  $i = 1$ : attempt to match  $V_i$  with  $\pi_i$  in environment  $\rho$  with respect to mapping  $\mu_{i-1}$ . If the match succeeds with mapping  $\mu_i$ , then  $i$  is incremented and the next value/pattern pair is tried. If  $\pi_n$  succeeds, the bracket pattern succeeds with the mapping  $\mu_n$ . The match fails if any  $\pi_i$  fails.

### Compound Patterns

- If  $\pi$  is of the form  $(\pi_1 \pi_2 \pi_3)$ , where  $\pi_1$  is a quantifier pattern and  $\pi_2$  is a list pattern, then the compound pattern will attempt to decompose the proposition  $V$ . The match fails if  $V$  is not a proposition. First,  $V$  is expressed in the form

$$(q v_1 (q v_2 \dots (q v_j B) \dots))$$

for some quantifier  $q$ , variables  $v_1, \dots, v_j$ , and proposition  $B$  not of the form  $(q v_{j+1} B')$ . Note that all propositions can be expressed in this form; however, if  $j = 0$  then  $q$  is undetermined. The following are then done in order; if any step fails, the match fails.

- If  $j > 0$ , then quantifier  $q$  is matched against the quantifier pattern  $\pi_1$  with respect to  $\rho$  and  $\mu$ , yielding  $\mu'$  if successful. If  $j = 0$  and  $\pi_1$  is a **(some-quant  $I$ )** pattern, the match fails.
- Starting with  $i = j$  and working down to  $i = 0$ , match the list  $[v_1 \dots v_i]$  against the list pattern  $\pi_2$  with respect to  $\rho$  and  $\mu'$ . The first successful match yields a mapping  $\mu''$  and the largest  $i$  such that  $[v_1 \dots v_i]$  is the longest variable list that matches  $\pi_2$ . The match fails if all such lists for  $0 \leq i \leq j$  fail.
- Finally, the result of the compound pattern is the proposition

$$(q v_{i+1} \dots (q v_j B) \dots)$$

matched against pattern  $\pi_3$  with respect to  $\rho$  and  $\mu''$ .

- If  $\pi$  is of the form  $(\pi_1 \pi_2)$ , where  $\pi_2$  is a list pattern, then the compound pattern will attempt to decompose the term  $V$ . The match fails if  $V$  is not a term of the form  $(f t_1 \dots t_i)$ .<sup>4</sup> The result

---

<sup>4</sup>For nullary function symbols,  $i = 0$ . Thus, the term is of the form  $(f)$ , and the subsequent pattern is  $[f []]$ .

of the compound pattern is the result of matching  $[f [t_1 \dots t_n]]$  against the bracket pattern  $[\pi_1 \pi_2]$  with respect to  $\rho$  and  $\mu$ .

- Otherwise, for  $\pi$  of the form  $(\pi_1 \dots \pi_n)$ :
  - If  $V$  is a term (or atom) of the form  $(f t_1 \dots t_i)$ , then the result is that of matching  $[f t_1 \dots t_i]$  against  $[\pi_1 \dots \pi_n]$  in  $\rho$  and  $\mu$ . The match fails if  $V$  is a term not of this form.
  - If  $V$  is logical proposition of the form  $(con P_1 \dots P_i)$ , for some prop-con  $con$  and  $i = 1$  or  $2$ , then the result is that of matching  $[con P_1 \dots P_i]$  against  $[\pi_1 \dots \pi_n]$  in  $\rho$  and  $\mu$ .
  - If  $V$  is a quantified proposition of the form  $(q v P)$ , then the result is that of matching  $[q v P]$  against  $[\pi_1 \dots \pi_n]$  in  $\rho$  and  $\mu$ .
  - Finally, if  $V$  is neither a term nor a proposition, the compound match fails.

## 2.3 Athena Directives

In addition to accepting phrases to produce values, Athena also accepts top-level **directives**. Directives do not produce a value but instead affect Athena in fundamental ways. Directives can introduce new function symbols and sorts, change the assumption base, create new bindings in the top-level environment, and bypass the read-eval-print loop to instead read directly from a file. The top-level directives recognized by monomorphic Athena are listed in Figure 2.2.

### 2.3.1 Introducing New Sorts and Symbols

In Athena, terms have a **sort**. Sorts are the objects within a universe of discourse that Athena can operate on. An Athena sort is either a **domain** or a **structure**, and directives are used to introduce new sorts. Athena starts with the predefined domain `Ide` (used for meta-identifiers) and the predefined structure `Boolean` (used for propositions).

#### Domains

New domains are introduced via the top-level directive `(domain I)`, where  $I$  specifies the name of the new domain. The namespace for sorts is distinct from that of symbols and bindings in the top-level

constructor	::=	$I \mid (I \ I^*)$
define-block	::=	$(I \ I^*) \ F$
directive	::=	$(\text{domain } I) \mid$ $(\text{declare } I \ I) \mid (\text{declare } I \ (-> (I^*) \ I)) \mid$ $(\text{structure } I \ \text{constructor}^+) \mid$ $(\text{structures } (I \ \text{constructor}^+)^+) \mid$ $(\text{use-numerals } (I \ \dots) \ I) \mid$ $(\text{define-numeric-operations}) \mid$ $(\text{define-numeric-operations } I) \mid$ $(\text{assert } F^+) \mid$ $\text{clear-assumption-base} \mid$ $(\text{define } I \ F) \mid (\text{define } \text{define-block}^+) \mid$ $(\text{load-file } \text{"filename"})$

Figure 2.2: Syntax of Athena Directives

environment; conceivably, one could declare a function symbol named `Ide` without a problem, except perhaps confusion on the part of the user. It is common to restrict names with a capitalized first letter to sorts. It is illegal (but harmless) to introduce a sort that already exists.

### Declaring New Function Symbols

The `declare` directive is used to create new function symbols by specifying what sorts each symbol accepts as arguments and which sort the symbol returns as a result. The directive `(declare  $I_n \ I_s$ )` creates a new constant symbol named  $I_n$  of sort  $I_s$ , assuming  $I_s$  is the name of an existing sort. It is an error to introduce a symbol that already exists.

A directive of the form `(declare  $I_n \ (-> (I_1 \dots I_k) \ I_s)$ )` is used to introduce a function symbol of the name  $I_n$ , of arity  $k$ , which accepts terms of the sorts  $I_1 \dots I_k$  and produces a term of sort  $I_s$ , assuming that the sorts  $I_1, \dots, I_k, I_s$  exist. The set of argument sorts and the return sort are sometimes called the **signature** of the function symbol.

For example, below are introduced a domain representing integers, a symbol representing an integer, another symbol representing addition upon integers, and finally a symbol representing a test of inequality.

```

(domain Integer)

(declare six Integer)

(declare + (-> (Integer Integer) Integer))

(declare not-equal (-> (Integer Integer) Boolean))

```

Function symbols whose return sort is Boolean, such as `not-equal` above, are also called **relations** or **predicates**.

After these directives, `(+ six six)` is recognized as a term of sort Integer, and `(not-equal six (+ six six))` would be recognized as a term of sort Boolean.

## Structures

A structure is an inductively generated sort. When a structure is declared, it includes a list of function symbols which can be used to inductively create terms of the structure; these function symbols are called **constructors**. The constructor listings are similar to the `declare` directive, except that the return sort need not be specified — the `structure` directive makes it clear what the return sort is.

For example, the code below includes a structure for Boolean (if it were not already predefined by Athena) and a possible structure for the natural numbers.

```

(structure Boolean
  true
  false)

(structure Natural
  zero
  (successor Natural))

```

Thus the sort Natural has two constructors: one nullary which represents a single term of sort Natural, and another which takes a Natural as input and returns a term of sort Natural. Because `successor` takes as argument a term of the structure being defined, it is a *reflexive* constructor. Constructors that are not reflexive are called *irreflexive*; for example, `true`, `false`, and `zero` are irreflexive. Besides requiring that the names of the constructors be distinct, Athena also requires that a structure be *inductive*: at least one of the constructors must be irreflexive.

There is also the `structures` directive which allows for the creation of mutually recursive structures. For mutually recursive structures, the notion of inductive is extended: a structure is inductive if and only if it has an irreflexive constructor or if it has a constructor one of whose argument sorts is inductive.

For example, the first structure listed below is not inductive, and would be rejected by Athena. The second structure has a “base case” for one of its sorts, which is sufficient to show that they are inductive.

```
(structures
  (First
    (is-before Second))
  (Second
    (is-after First)))
```

```
(structures
  (Foo
    (fop Bar Baz))
  (Bar
    (bop Baz))
  (Baz
    buzz
    (biz Bar)))
```

### 2.3.2 Numerals

Athena also supports numerals and mathematical operations, but only after the user has specifically introduced each. The directive `(use-numerals ( $I_{num}$  ...)  $I_{domain}$ )` expects either 0 or 1 and a domain. After introduction, identifiers of the specified number and greater are recognized to be terms of the domain. (The full Athena language also allows 0.0 and 1.0 to allow real numbers; however, integers are sufficient for this subset of Athena.) In particular, if `use-numerals` is given 1, the identifier 0 is not recognized as a valid term of the domain. Attempts to reintroduce numerals are ignored, as are attempts to introduce numerals with a structure.

The directive `define-numeric-operations` is used to introduce Athena’s mathematical, primitive functions after `use-numerals` has been successfully invoked. In its nullary form, the identifiers `plus`, `minus`, `times`, `div`, `exp`, `mod`, `less?`, and `greater?` are introduced to operate on numerals. The primitive `minus`, however, is partial, in that



it will not return a number lower than the start number specified in `use-numerals`.

The alternative form (`define-numeric-operations I`) takes an argument that specifies the inverse operation for numeric terms. The identifier *I* must be a unary function symbol with both the argument sort and the return sort being the numerical domain. This function symbol can then be used to negate terms of the numerical domain, and these negated terms can be used and returned by Athena's mathematical functions.

### 2.3.3 Other Directives

The directive `assert` accepts phrases. Each phrase is evaluated to produce a value, and if each value is a proposition then that proposition is added to the assumption base. The directive `clear-assumption-base` resets the assumption base to its initial value: the lone proposition `true`.

The (`define I F`) directive creates a new binding in the top-level environment from the given identifier *I* to the value of phrase *F*. The phrase *F* is evaluated with respect to an environment similar to `letrec` or `dletrec`, in that the identifier *I* can be referenced in *F*, but not used to obtain a value. Because of the environment, this form of the `define` directive can be used to make recursive closures.

The alternative form of the `define` directive provides syntactic sugar for the creation of closures, similar to that of other Lisp-like languages. In a given `define` block, (*I I\**) denotes that the name of the new closure is *I* and that the arguments to the closure are *I\**. The type of closure, function or method, is determined by the phrase *F*: if *F* is an expression, a function closure; if *F* is a deduction, a method closure. In addition, this form of `define` permits the declaration of mutually-recursive closures: each closure can reference the other closures by the identifier *I* listed in each `define` block.

Finally, the directive `load-file` bypasses the read-eval-print loop directly and reads Athena input from the specified file. Athena acts upon the phrases and directives within the file as if they were presented to the repl itself.

### 2.3.4 Sort Checking

Athena verifies that all terms and propositions are well-sorted. For monomorphic Athena, sort checking is quite straightforward. For polymorphic Athena, however, sort checking is more sophisticated; the full

details are provided as an appendix in [Ark99b]. This section will provide an algorithm for determining if a term or proposition is well-sorted, as well as the sorts imposed on its variables.

A context  $\Delta$  is a mapping of variables to sorts. For any term or proposition, each variable will have exactly one sort in the context  $\Delta$ . The only exception is if the term is a lone variable, in which case the variable (and the resulting term) does not have a sort.<sup>5</sup> In checking the well-sortedness of a term, it is verified that each subterm has the sort required by the signatures of its component function symbols, and that each variable in the term has at most one sort in the context  $\Delta$ .

The algorithm accepts a term  $t$  and a context  $\Delta$  and either returns a context or fails. The algorithm is called initially with an empty context and proceeds by structural recursion on the term  $t$ :

- If  $t$  is a variable, return the context  $\Delta$ .
- If  $t$  is a nullary function symbol, return the context  $\Delta$ .
- If  $t$  is a function symbol application of arity  $n > 0$ , of the form  $(f\ t_1 \dots t_n)$ , the algorithm checks that each subterm  $t_i$  is of the sort specified by the signature of  $f$ . Denote the signature of  $f$  as  $S_1 \times \dots \times S_n \rightarrow S$ , where each  $S_i$  is an argument sort and  $S$  is the return sort of the function symbol. Setting  $\Delta_0 = \Delta$ , for  $i = 1, \dots, n$ :
  - If  $t_i$  is a variable  $v$ , then extend the context  $\Delta_{i-1}$  with the mapping  $v \mapsto S_i$  to obtain the new context  $\Delta_i$ , where  $S_i$  is the sort specified by the signature of the function symbol  $f$ . The algorithm fails if there is already a mapping  $v \mapsto S_v$  in  $\Delta_{i-1}$ , where  $S_v \neq S_i$ .
  - Otherwise, the algorithm recurses with arguments  $t_i$  and  $\Delta_{i-1}$  to obtain  $\Delta_i$ . If the recursive call fails, the algorithm fails.

If no recursive calls failed, then the result of the algorithm is the context  $\Delta_n$ .

For example, the algorithm applied to term  $(+ \text{?x six})$  (and the initially empty context) would result in the mapping  $\text{?x} \mapsto \text{Integer}$ , as specified by the signature of  $+$ . However, the algorithm would fail for  $(\text{successor six})$ , as  $\text{six}$  is of sort  $\text{Integer}$  but  $\text{successor}$  requires a  $\text{Natural}$ .

---

<sup>5</sup>In polymorphic Athena, the sort of a lone variable is actually a Sort Variable, but monomorphic Athena has no Sort Variables, which is why it is specified to have no sort at all.

Sort checking for propositions is similar, but with a few additional details. Well-sorted propositions must have their atoms be of sort Boolean, and must also deal with quantified variables properly. The algorithm is extended to check propositions; for a given proposition  $P$  and an (initially empty) context  $\Delta$ :

- If  $P$  is an atom  $t$ :
  - If  $t$  is a variable  $v$ , then return the context  $\Delta$  extended with the binding  $v \mapsto \text{Boolean}$ ; fail if  $\Delta$  has a binding to a non-Boolean sort.
  - If  $t$  is a nullary function symbol, fail if it is not of sort Boolean. Otherwise, return the context  $\Delta$ .
  - If  $t$  is a function symbol application, fail if the return sort of the function symbol is not of sort Boolean. The algorithm otherwise proceeds by sort checking the term  $t$  with respect to  $\Delta$ .
- If  $P$  is a logical proposition, the algorithm recurses on the first subproposition  $P_1$  with  $\Delta$  and (if the logical proposition is binary) uses the returned context to recurse on the second subproposition  $P_2$ . The algorithm fails if either recursive call fails.
- If  $P$  is a quantified proposition, the algorithm recurses on the body of the proposition and the context  $\Delta$  with any binding for the quantified variable *removed*.

Thus for example, following proposition is legal, even though `?x` apparently has more than one sort:

```
(and (forall ?x
      (not-equal ?x (+ ?x six)))
     (or ?x false))
```

The `forall` provides a separate scope for `?x` in which it is well-sorted, and the sort that `?x` takes within the quantified proposition is lost with respect to the rest of the proposition. This concludes the discussion of sort checking in monomorphic Athena.

The next chapter details the layout of the Athena virtual machine and its operations and transitions. When properly assembled, the virtual machine will be able to execute a large subset of monomorphic Athena. Features missing include support for the `split` pattern, the `by-induction-on` deduction used for induction upon structures, and certain automatically generated methods for structures that the full Athena language provides.

## Chapter 3

# The Virtual Machine

The Athena virtual machine is a stack-based machine designed to support functional and deductive programming, sophisticated pattern matching, and fast error handling. It can be considered to be a heavily modified SECD machine [Lan64].

The initial design of the virtual machine was based on Henderson’s Lispkit Lisp interpreter, which uses an SECD machine with twenty-one operations. Lispkit is a small, purely functional language discussed extensively in [Hen80]. Lispkit supports such important features as higher-order function closures, function application, and `letrec`. Lispkit lacks features such as a top-level environment, state, and error handling. Most of the operations in Henderson’s SECD machine are still available in the Athena machine, as are its four stacks: *S*, *E*, *C*, and *D*. The Athena VM has also inherited the SECD machine’s non-flat code list — that is, instead of compiled code being merely a list of operations and operation arguments, it is a list of operations, operation arguments, and *other lists* which themselves include compiled code.

Some additional design considerations come from Cardelli’s Functional Abstract Machine [Car83], which is itself a heavily modified SECD machine. In particular, the Fam uses a stack *M* as “memory” to hold state in the machine and provides operations `Ref`, `At`, and `DestRef` to access *M*. These have been incorporated into the Athena VM as `REF`, `AT`, and `REFASSIGN` to provide the functionality of Athena cells. The Fam also has an explicit operation for generating the unit value, `Triv`, which has been borrowed as well by the Athena VM.

## 3.1 The Stacks

The Athena virtual machine consists of seven stacks or lists.

- S: The evaluation Stack** stores intermediate results during the evaluation of expressions and deductions. The objects on top of *S* are usually the ones manipulated by each state transition.
- E: The Environment stack** is used to hold the values that are bound to identifiers. Effectively, it is a list of lists, and each internal list represents a “layer” of values that have been bound by a single closure.
- C: The Control list** contains the machine code that is being executed. At the top level, it is a non-flat list of machine operations and arguments. Most steps of the virtual machine involve transitions based on what operation is on top of the *C* stack.
- D: The Dump** holds machine state in between certain expression-specific operations. For example, the *S*, *E*, and *C* are dumped onto this stack before function application and are restored upon return of the function. Usually, objects are pushed on to the *D* stack during the evaluation of expressions.
- M: The Memory stack** stores the values contained in Athena cells.
- A: The Assumption base stack** holds all propositions that are “true,” as well as any intermediate conclusions produced during the evaluation of a deduction.
- B: The Backup stack** holds machine state in between certain evaluations, such as method applications. During the evaluation of some phrases, the assumption base stack may need to be saved and restored. The backup stack is where the assumption base stack is saved to and restored from.

### 3.1.1 The Initial Environment Stack

When Athena starts, five of the seven stacks are initially empty. The first exception is *A*: the assumption base stack initially contains the proposition `true` by convention. The environment stack *E* contains the initial top-level environment. Both of these can change as Athena evaluates phrases and directives: successful deductions will add their conclusion to the *A* stack, and successful `define` directives will update

the top-level environment  $E$ . Each phrase entered is parsed and compiled into a code list; the details of this in Chapter 4. The stack  $C$  is set to be the new code list, which the machine proceeds to execute.

The environment stack  $E$  at any point is a list of lists. The top-level environment stack — the one in which all phrases are executed with respect to — is a list of exactly two lists. As  $E$  is a stack, the top element is referenced by the index 0, and lists that are deeper in the stack are referenced by 1, 2, and so on. (Thus a reference to a *value* in an environment  $E$  is a pair of nonnegative integers; the first being which list in  $E$  to look in, and the second being an index into that list to get the value. This is done by the LD operation.) The top-most list of the top-level environment stack contains all values that have been **defined** by the user; it initially starts out as an empty list. The deeper list contains all of the Athena primitives. It never changes except for if and when a **define-numeric-operations** directive is successfully issued, at which point it will extend to include the numerical primitive function closures. The full details of what primitives Athena provides is discussed starting in Section 4.2.

### 3.1.2 Sorts and Symbols

There is an additional collection of information that is important in Athena: the set of defined sorts and the set of declared function symbols, along with their arity and arguments. However, this information is not directly used by the virtual machine itself. It is the *compiler* that needs this information, so that it can generate the proper Athena code. The machine itself does not worry about whether a given identifier is a function symbol, because the compiler would have made that distinction — if it happened to be the name of a function symbol, then the code list generated by the compiler will reference a function symbol.

In between the time when the machine starts executing a code list and when (if) the machine completes, no new function symbols or sorts are created. They are created only by top-level directives in the read-eval-print loop. After such a directive, then the compiler can make use of the new sorts and symbols in later phrases. The distinction is made because the virtual machine does not deal with their creation, only the fact that they exist. And the fact that they exist is not even noticed by the machine, it is noted by the compiler which generated the code for the machine. Thus, the sort and symbol information is not available as a stack because is not manipulated by the machine, and adding it to the machine would complicate it needlessly.

## 3.2 Operations and Transitions

With each machine step, the machine attempts to match the current state of its stacks against the transition rules. On finding a match, it proceeds to manipulate the stacks according to the rule.

In the state transitions below, the letters  $x$  and  $y$  denote arbitrary values, whereas  $t$  denotes a term,  $v$  a variable,  $f$  a function symbol,  $\theta$  a substitution, and  $p$  a proposition. Lists are in brackets,  $[\ ]$ . Function and method closures appear as parenthesized lists with the leading tag *function* or *method*. For each transition, each stack is either labeled as a letter or as parenthesized list of the top elements of the stack, a dot, and then the remainder of the stack labeled as a letter. Not all transitions include all seven stacks. Most transitions either have the four basic expression stacks (*SECD*) or the six core expression and deduction stacks (*SECDAB*); there are only three transitions that use the *M* stack. An explicitly empty stack will be denoted by *NIL*. Finally, operations are denoted by *OPNAME*.

The machine halts if the code list  $C$  is empty. If  $C$  is not empty and there is no legal transition, the machine halts with an irrecoverable error. This is distinct from the *ERROR* pseudo-value, as *ERROR* has its own special transitions, as listed in Section 3.2.9.

### 3.2.1 Value Loading

$$1. \quad \frac{S \quad E \quad (\text{TRIV} . C) \quad D}{((\ ) . S) \quad E \quad C \quad D}$$

Generate the unit value.

$$2. \quad \frac{(x . S) \quad E \quad (\text{DUP} . C) \quad D}{(x x . S) \quad E \quad C \quad D}$$

Duplicate the top value on  $S$ .

$$3. \quad \frac{\text{NIL} \quad E \quad (\text{DUP} . C) \quad D}{(\text{ERROR}) \quad E \quad C \quad D}$$

$$4. \quad \frac{(x y . S) \quad E \quad (\text{SWAP} . C) \quad D}{(y x . S) \quad E \quad C \quad D}$$

Swap the top two stack elements.

$$5. \quad \frac{S \quad E \quad (\text{SWAP} . C) \quad D}{(\text{ERROR}) \quad E \quad C \quad D}$$

If  $S$  has less than two elements.

$$6. \quad \frac{S \quad E \quad (\text{LD } m \ n . C) \quad D}{(x . S) \quad E \quad C \quad D}$$

If  $m, n$  is a proper offset into the environment  $E$ .

$$7. \quad \frac{S \quad E \quad (\text{LD } m \ n . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

If  $m, n$  is not a proper offset into the environment  $E$ .

$$8. \quad \frac{S \quad E \quad (\text{FRESHVAR} . C) \quad D}{(v_{\text{fresh}} . S) \quad E \quad C \quad D}$$

Produce a variable that has never before been introduced.

$$9. \quad \frac{S \quad E \quad (\text{LDV} \text{ "v"} . C) \quad D}{(v . S) \quad E \quad C \quad D}$$

Load a variable by name.

$$10. \quad \frac{S \quad E \quad (\text{LDPRIM } x . C) \quad D}{(x . S) \quad E \quad C \quad D}$$

Load a primitive value onto the  $S$  stack. Normally used for props, cons, quantifiers, symbols, and characters.

$$11. \quad \frac{S \quad E \quad (\text{LDTRUE} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

Generate the Boolean function symbol `true`.

$$12. \quad \frac{S \quad E \quad (\text{LDFALSE} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Generate the Boolean function symbol `false`.

$$13. \quad \frac{S \quad E \quad (\text{LDERROR} \text{ "string"} . C) \quad D}{(\text{ERROR} \text{ "string"} . S) \quad E \quad C \quad D}$$

Produce the error pseudo-value containing the user visible string.



### 3.2.2 Lists and List Manipulation

$$14. \quad \frac{S \quad E \quad (\text{NIL} . C) \quad D}{([\ ] . S) \quad E \quad C \quad D}$$

Create an empty list.

$$15. \quad \frac{(x [x_1 \dots x_n] . S) \quad E \quad (\text{PRE} . C) \quad D}{([x \ x_1 \dots x_n] . S) \quad E \quad C \quad D}$$

Prepend a value onto a list

$$16. \quad \frac{(x [x_1 \dots x_n] . S) \quad E \quad (\text{POST} . C) \quad D}{([x_1 \dots x_n \ x] . S) \quad E \quad C \quad D}$$

Append a value onto a list.

$$17. \quad \frac{([x_0 \dots x_{n-1}] . S) \quad E \quad (\text{ELEM } m . C) \quad D}{(x_m . S) \quad E \quad C \quad D}$$

Get the  $m$ th element of a list, if  $0 \leq m < n$ .

$$18. \quad \frac{([x_0 \dots x_{n-1}] . S) \quad E \quad (\text{ELEM } m . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $m < 0$  or  $m \geq n$ .

$$19. \quad \frac{([x_1 \dots x_n] . S) \quad E \quad (\text{CDR} . C) \quad D}{([x_2 \dots x_n] . S) \quad E \quad C \quad D}$$

Return the list of all but the first element of a list.

$$20. \quad \frac{([\ ] . S) \quad E \quad (\text{CDR} . C) \quad D}{([\ ] . S) \quad E \quad C \quad D}$$

This transition is explicitly here to act like Athena's `tail` primitive function. (Section 4.3.)

$$21. \quad \frac{(x . S) \quad E \quad (\text{CDR} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a list.

$$22. \quad \frac{(x . S) \quad E \quad (\text{ASLIST} . C) \quad D}{([x_0 \ x_1 \ \dots \ x_n] . S) \quad E \quad C \quad D}$$

If  $x$  is a (non-variable) term, returns the list of the function symbol  $x_0$  and the subterms  $x_1 \dots x_n$ .

If  $x$  is a logical proposition, returns the list of the prop-con  $x_0$  and propositions  $x_1 \dots x_n$ .

If  $x$  is a quantified proposition, returns the list of the quantifier  $x_0$ , variable  $x_1$ , and proposition  $x_2$ .

$$23. \quad \frac{(x . S) \quad E \quad (\text{ASLIST} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a non-variable term or proposition.

### 3.2.3 Closure Creation

$$24. \quad \frac{S \quad E \quad (\text{CLOSURE } n \ c' . C) \quad D}{((\text{function } n \ c' \ E) . S) \quad E \quad C \quad D}$$

Build a function closure expecting  $n$  arguments from the code list  $c'$ .

$$25. \quad \frac{S \quad E \quad (\text{DCLOSURE } n \ c' . C) \quad D}{((\text{method } n \ c' \ E) . S) \quad E \quad C \quad D}$$

Build a method closure expecting  $n$  arguments from the code list  $c'$ .

### 3.2.4 Application

$$26. \quad \frac{\frac{([x_1 \dots x_n] (\text{function } n \ c' \ e') . S) \quad E}{\text{NIL} \quad ([x_1 \dots x_n] . e')}}{(\text{APPLY} . C) \quad D}{c' \quad (S \ E \ C . D)}$$

Where  $[x_1 \dots x_n]$  is the list of arguments, of arity  $n$ , to the function closure.

Applying a function closure consists of saving the current  $S$ ,  $E$ , and  $C$  stacks to the dump  $D$ , loading the environment of the closure, adding the arguments to the new environment, and then setting the code list to be the closure code.

$$27. \quad \frac{([x_1 \dots x_m] (\text{function } n \text{ } c' \text{ } e') . S) \quad E \quad (\text{APPLY} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $[x_1 \dots x_m]$  is a list of arguments of length  $m \neq n$ , where  $n$  is the arity of the function closure.

$$28. \quad \frac{([t_1 \dots t_n] f . S) \quad E \quad (\text{APPLY} . C) \quad D}{(t . S) \quad E \quad C \quad D}$$

Where  $f$  is a function symbol, and  $[t_1 \dots t_n]$  is a list of terms, with the arity of the symbol equal to the length of the term list. Also, the resulting term  $t$  is well-sorted.

$$29. \quad \frac{([t_1 \dots t_m] f . S) \quad E \quad (\text{APPLY} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where the arity of  $f$  is not equal to  $m$ , or if the resulting term would have been ill-sorted.

$$30. \quad \frac{(x \theta . S) \quad E \quad (\text{APPLY} . C) \quad D}{(\theta(x) . S) \quad E \quad C \quad D}$$

Where  $x$  is a term, list of terms, proposition, or list of propositions.

$$31. \quad \frac{(x \theta . S) \quad E \quad (\text{APPLY} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x$  is neither a term, a list of terms, a proposition, nor a list of propositions.

$$32. \quad \frac{([p_1 \dots p_n] \text{con} . S) \quad E \quad (\text{APPLY} . C) \quad D}{((\text{con } p_1 \dots p_n) . S) \quad E \quad C \quad D}$$

Where  $\text{con}$  is a prop-con, where the arity of  $\text{con}$  equals  $n$ .

$$33. \quad \frac{(x \text{con} . S) \quad E \quad (\text{APPLY} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $\text{con}$  is a prop-con, but  $x$  is either not a list of propositions of the correct length or if the resulting proposition would have been ill-sorted.

$$34. \quad \frac{([v p] q . S) \quad E \quad (\text{APPLY} . C) \quad D}{((q v p) . S) \quad E \quad C \quad D}$$

Where  $q$  is a quantifier.

$$35. \quad \frac{(x q . S) \quad E \quad (\text{APPLY} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $q$  is a quantifier, but  $x$  either is not a list of a variable and a proposition or if the resulting proposition would have been ill-sorted.

$$36. \quad \frac{(x . s') \quad e' \quad (\text{RTN}) \quad (S E C . D)}{(x . S) \quad E \quad C \quad D}$$

The return from a closure. Note that the top (presumably, only) value of  $s'$  is “returned” when the closure ends.

$$37. \quad \frac{\text{NIL} \quad e' \quad (\text{RTN}) \quad (S E C . D)}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

### 3.2.5 Deductions and Method Application

$$38. \quad \frac{S \quad E \quad (\text{DSTART} . C) \quad D \quad A \quad B}{S \quad E \quad C \quad D \quad A \quad (A . B)}$$

Keep a copy of the assumption base safe in  $B$ .

$$39. \quad \frac{(p . S) \quad E \quad (\text{DEND} . C) \quad D \quad a' \quad (A . B)}{(p . S) \quad E \quad C \quad D \quad (p . A) \quad B}$$

Add the proposition  $p$  (now a theorem) at the top of the  $S$  stack to the restored assumption base. Note that it is also left on the top of  $S$ .

$$40. \quad \frac{S \quad E \quad (\text{DRESTORE} . C) \quad D \quad a' \quad (A . B)}{S \quad E \quad C \quad D \quad A \quad B}$$

Restore the old assumption base unmodified.

$$41. \quad \frac{\frac{([x_1 \dots x_n] \text{ (method } n \ c' \ e') . S}) \quad E}{\text{NIL}}}{\frac{\text{(DAPPLY . } C) \quad D}{c' \quad (S \ E \ C \ . \ D)}}$$

Where the arity of the method closure equals the length of the value list.

$$42. \quad \frac{\frac{(y \ x \ . \ S) \quad E \quad \text{(DAPPLY . } C) \quad D}{(\text{ERROR . } S) \quad E} \quad C \quad D}{D}$$

If  $x$  is not a method closure or  $y$  is not a list, or if  $x$  is a method closure whose arity is not equal to the length of list  $y$ .

$$43. \quad \frac{\frac{(p \ . \ S) \quad E \quad \text{(ASSERT . } C) \quad D \quad A \quad B}{(p \ . \ S) \quad E} \quad C \quad D \quad (p \ . \ A) \quad B}{D \quad A \quad B}$$

Add proposition  $p$  to the assumption base. Note that the proposition remains on  $S$ .

$$44. \quad \frac{\frac{(x \ . \ S) \quad E \quad \text{(ASSERT . } C) \quad D \quad A \quad B}{(\text{ERROR . } S) \quad E} \quad C \quad D \quad A \quad B}{D \quad A \quad B}$$

If  $x$  is not a proposition.

### 3.2.6 Recursive Closures and Applications

$$45. \quad \frac{\frac{S \quad E \quad \text{(DUMMY . } C) \quad D}{S \quad \text{(pending . } E)} \quad C \quad D}{D}$$

Add a dummy environment layer to  $E$ . It will be replaced by RAPPPLY or RDAPPLY later.

$$46. \quad \frac{\frac{([x_1 \dots x_n] \text{ (function } n \ c' \ \text{(pending . } e') \ . \ S)})}{\text{NIL}}}{\frac{\text{(pending . } E) \quad \text{(RAPPPLY . } C) \quad D}{(v \ . \ e') \quad c' \quad (S \ E \ C \ . \ D)}}$$

The effective replacement of **pending** with the argument list happens both on top of the  $E$  stack, *but also within the environments of any closures in the argument list.*

$$47. \quad \frac{(y \ x \ . \ S) \quad E \quad (\text{RAPPLY} \ . \ C) \quad D}{(\text{ERROR} \ . \ S) \quad E \quad C \quad D}$$

If  $x$  is not a function closure of the proper form,  $y$  is not a list, or if  $x$  is a function closure whose arity does not match the length of list  $y$ .

$$48. \quad \frac{([x_1 \dots x_n] \ (\text{method } n \ c' \ (\text{pending} \ . \ e')) \ . \ S)}{\text{NIL}} \\ \frac{(\text{pending} \ . \ E) \quad (\text{RDAPPLY} \ . \ C) \quad D}{([x_1 \dots x_n] \ . \ e') \quad c' \quad (S \ E \ C \ . \ D)}$$

The effective replacement of **pending** with the argument list happens both on top of the  $E$  stack, *but also within the environments of any closures in the argument list.*

$$49. \quad \frac{(y \ x \ . \ S) \quad E \quad (\text{RDAPPLY} \ . \ C) \quad D}{(\text{ERROR} \ . \ S) \quad E \quad C \quad D}$$

If  $x$  is not a method closure of the proper form,  $y$  is not a list, or if  $x$  is a method closure whose arity does not match the length of list  $y$ .

### 3.2.7 Select

SEL is the most basic and generic switching operation. SEL chooses the next code list depending on the value on top of  $S$ . Each selection code list “returns” via the JOIN operation.

$$50. \quad \frac{(\text{true} \ . \ S) \quad E \quad (\text{BOOLNOT} \ . \ C) \quad D}{(\text{false} \ . \ S) \quad E \quad C \quad D}$$

$$51. \quad \frac{(\text{false} \ . \ S) \quad E \quad (\text{BOOLNOT} \ . \ C) \quad D}{(\text{true} \ . \ S) \quad E \quad C \quad D}$$

$$52. \quad \frac{(x \ . \ S) \quad E \quad (\text{BOOLNOT} \ . \ C) \quad D}{(\text{ERROR} \ . \ S) \quad E \quad C \quad D}$$

Where  $x$  is neither true nor false.

BOOLNOT is used to make the SEL operation an explicit test for false.

$$53. \quad \frac{(\text{true} . S) \quad E \quad (\text{SEL } c_{\text{true}} \ c_{\text{not-true}} . C) \quad D}{S \quad E \quad c_{\text{true}} \quad (C . D)}$$

Where “true” is the boolean function symbol **true**.

$$54. \quad \frac{(x . S) \quad E \quad (\text{SEL } c_{\text{true}} \ c_{\text{not-true}} . C) \quad D}{S \quad E \quad c_{\text{not-true}} \quad (C . D)}$$

Where  $x$  is any value other than **true**.

For an explicit test for **false** use **BOOLNOT** before **SEL**.

$$55. \quad \frac{S \quad E \quad (\text{JOIN}) \quad (C . D)}{S \quad E \quad C \quad D}$$

$$56. \quad \frac{(x . S) \quad E \quad (\text{POP} . C) \quad D}{S \quad E \quad C \quad D}$$

Remove the top-most element from  $S$ .

$$57. \quad \frac{\text{NIL} \quad E \quad (\text{POP} . C) \quad D}{(\text{ERROR}) \quad E \quad C \quad D}$$

If  $S$  is empty, return **ERROR**.

### 3.2.8 While

$$58. \quad \frac{S \quad E \quad (\text{WHILE } c_{\text{loop}} . C) \quad D}{S \quad E \quad c_{\text{loop}} \quad (c_{\text{loop}} \ C . D)}$$

$$59. \quad \frac{(\text{true} . S) \quad E \quad (\text{WHILETEST} . C) \quad (c_{\text{loop}} \ c_{\text{cont}} . D)}{S \quad E \quad C \quad (c_{\text{loop}} \ c_{\text{cont}} . D)}$$

$$60. \quad \frac{(\text{false} . S) \quad E \quad (\text{WHILETEST} . C) \quad (c_{\text{loop}} \ c_{\text{cont}} . D)}{(() . S) \quad E \quad c_{\text{cont}} \quad D}$$

$$61. \quad \frac{(x . S) \quad E \quad (\text{WHILETEST} . C) \quad (c_{\text{loop}} \ c_{\text{cont}} . D)}{(\text{ERROR} . S) \quad E \quad C \quad (c_{\text{loop}} \ c_{\text{cont}} . D)}$$

Where  $x$  is neither **true** nor **false**.

$$62. \quad \frac{(x . S) \quad E \quad (\text{WHILELOOP} . C) \quad (c_{\text{loop}} \ c_{\text{cont}} . D)}{S \quad E \quad c_{\text{loop}} \quad (c_{\text{loop}} \ c_{\text{cont}} . D)}$$

### 3.2.9 Try and the Error Transitions

The use of “...” in these transitions denotes a possibly empty list of stack objects that do not contain the pseudo-value “try-token”. At any given time, there may be multiple try-tokens on  $D$  and  $B$ , and these transitions only deal with respect to the outermost try-token. When the special pseudo-value `ERROR` is on top of the  $S$  stack, the machine reverts to these transitions until the error has been dealt with — either by moving to the next deduction in a `try` block or reporting the error to the user. The machine acts similarly for the pseudo-value `match-fail`, as discussed in Section 3.3.4.

$$63. \quad \frac{\frac{S \quad E \quad (\text{TRY} . C)}{S \quad E} \quad \frac{D}{(c_{\text{retry}} \text{ try-token } S E . D)}}{A \quad B} \quad \frac{A \quad B}{A \quad (\text{try-token } A . B)}$$

Where  $c_{\text{retry}}$  is actually taking  $C$ , removing everything up until `RETRY` or `FAIL`. Thus  $c_{\text{retry}}$  is a code list starting with either `RETRY` or `FAIL`.

$$64. \quad \frac{\frac{(\text{ERROR} . s') \quad e' \quad (\text{RETRY} . C)}{S \quad E \quad C}}{\frac{(\dots c'_{\text{retry}} \text{ try-token } S E . D) \quad a' \quad (\dots \text{try-token } A . B)}{(c_{\text{retry}} \text{ try-token } S E . D) \quad A \quad (\text{try-token } A . B)}}$$

Where  $c_{\text{retry}}$  is freshly generated from  $C$ .

$$65. \quad \frac{\frac{(p . S) \quad E \quad (\text{RETRY} . c)}{(p . S) \quad E} \quad C}{\frac{(\dots c_{\text{retry}} \text{ try-token } s' e' . D) \quad A \quad (\dots \text{try-token } a' . B)}{D \quad A \quad B}}$$

Success!  $A$  should already have  $p$  as the conclusion, so just return  $D$  and  $B$  to a try-token-less clean state.

Where  $C$  is actually  $c$ , removing everything up to and including `FAIL`.

$$66. \quad \frac{\frac{(\text{ERROR} . s') \quad e' \quad (\text{FAIL} . C)}{(\text{ERROR} \text{“try-failed”} . S) \quad E} \quad C}{\frac{(\dots c_{\text{retry}} \text{ try-token } S E . D) \quad a' \quad (\dots \text{try-token } A . B)}{D \quad A \quad B}}$$



$$67. \quad \frac{\frac{(p . S) \quad E \quad (\text{FAIL} . C)}{(p . S) \quad E \quad C}}{(\dots c_{\text{retry}} \text{ try-token } s' e' . D) \quad A \quad (\dots \text{ try-token } a' . B)} \quad \frac{D \quad A \quad B}{A}$$

$$68. \quad \frac{\frac{(\text{ERROR} . S) \quad E \quad (x . C)}{(\text{ERROR} . S) \quad E \quad c_{\text{retry}}}}{(\dots c_{\text{retry}} \text{ try-token } s' e') \quad A \quad (\dots \text{ try-token } A . B)} \quad \frac{A \quad (\dots \text{ try-token } A . B)}{(\dots c_{\text{retry}} \text{ try-token } s' e') \quad A \quad (\dots \text{ try-token } A . B)}$$

Where  $x$  is neither RETRY nor FAIL.

Upon any error, search  $D$  until a try-token has been found, and set  $C$  to be the code list right before the try-token.

$$69. \quad \frac{\frac{(\text{ERROR} . S) \quad E \quad \text{NIL} \quad (\dots c_{\text{retry}} \text{ try-token } s' e')}{(\text{ERROR} . S) \quad E \quad c_{\text{retry}} \quad (\dots c_{\text{retry}} \text{ try-token } s' e')}}{\frac{A \quad (\dots \text{ try-token } A . B)}{A \quad (\dots \text{ try-token } A . B)}}$$

### 3.2.10 Substitutions

$$70. \quad \frac{S \quad E \quad (\text{NULLSUB} . C) \quad D}{(\theta_{\text{empty}} . S) \quad E \quad C \quad D} \quad \frac{D}{D}$$

Introduce an empty substitution.

$$71. \quad \frac{(t v \theta . S) \quad E \quad (\text{EXTENDSUB} . C) \quad D}{(\theta[v \mapsto t] . S) \quad E \quad C \quad D} \quad \frac{D}{D}$$

Extend the given substitution. Note that if  $v$  is already mapped in  $\theta$ , that mapping is lost.

$$72. \quad \frac{(\theta_2 \theta_1 . S) \quad E \quad (\text{COMPOSESUB} . C) \quad D}{(\theta_1 \circ \theta_2 . S) \quad E \quad C \quad D} \quad \frac{D}{D}$$

Compose two substitutions.

### 3.2.11 Propositions

While it is true that propositions can be created using APPLY and the proper prop-con or quantifier, these are more direct. In particular, these are used by the primitive methods of Athena.

$$73. \quad \frac{(p . S) \quad E \quad (\text{NOT} . C) \quad D}{((\text{not } p) . S) \quad E \quad C \quad D}$$

$$74. \quad \frac{(x . S) \quad E \quad (\text{NOT} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a proposition.

$$75. \quad \frac{(p_2 p_1 . S) \quad E \quad (\text{AND} . C) \quad D}{((\text{and } p_1 p_2) . S) \quad E \quad C \quad D}$$

$$76. \quad \frac{(y x . S) \quad E \quad (\text{AND} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x$  or  $y$  is not a proposition.

$$77. \quad \frac{(p_2 p_1 . S) \quad E \quad (\text{OR} . C) \quad D}{((\text{or } p_1 p_2) . S) \quad E \quad C \quad D}$$

$$78. \quad \frac{(y x . S) \quad E \quad (\text{OR} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x$  or  $y$  is not a proposition.

$$79. \quad \frac{(p_2 p_1 . S) \quad E \quad (\text{IF} . C) \quad D}{((\text{if } p_1 p_2) . S) \quad E \quad C \quad D}$$

$$80. \quad \frac{(y x . S) \quad E \quad (\text{IF} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x$  or  $y$  is not a proposition.

$$81. \quad \frac{(p_2 p_1 . S) \quad E \quad (\text{IFF} . C) \quad D}{((\text{iff } p_1 p_2) . S) \quad E \quad C \quad D}$$

$$82. \quad \frac{(y \ x . S) \quad E \quad (\text{IFF} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x$  or  $y$  is not a proposition.

$$83. \quad \frac{(p \ v . S) \quad E \quad (\text{EXISTS} . C) \quad D}{((\text{exists } v \ p) . S) \quad E \quad C \quad D}$$

$$84. \quad \frac{(y \ x . S) \quad E \quad (\text{EXISTS} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a variable or  $y$  is not a proposition.

$$85. \quad \frac{(p \ v \ t . S) \quad E \quad (\text{FORALL} . C) \quad D}{((\text{forall } v \ p) . S) \quad E \quad C \quad D}$$

$$86. \quad \frac{(y \ x . S) \quad E \quad (\text{FORALL} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a variable or  $y$  is not a proposition.

$$87. \quad \frac{(p \ v \ t . S) \quad E \quad (\text{REPLACE} . C) \quad D}{(p' . S) \quad E \quad C \quad D}$$

Perform the “safe replacement” of variable  $v$  with  $t$  in proposition  $p$ . This involves the renaming of bound variables in  $p$ .

### 3.2.12 Store Access and Manipulation

$$88. \quad \frac{(x . S) \quad E \quad (\text{REF} . C) \quad D \quad M}{(\text{cell}_i . S) \quad E \quad C \quad D \quad M[i \mapsto x]}$$

Creates a cell containing the value  $x$ .

$$89. \quad \frac{(\text{cell}_i . S) \quad E \quad (\text{AT} . C) \quad D \quad M}{(M[i] . S) \quad E \quad C \quad D \quad M}$$

Retrieve the value in the cell.

$$90. \quad \frac{(x \ \text{cell}_i . S) \quad E \quad (\text{REFASSIGN} . C) \quad D \quad M}{S \quad E \quad C \quad D \quad M[i \mapsto x]}$$

Replace the value stored in a cell with a new value.

### 3.2.13 Assumption Base Access

The use of  $ae(p)$  in the following transitions denotes whether a proposition alphabetically equivalent to  $p$  is or is not contained within  $A$ .

$$91. \quad \frac{(p . S) \quad E \quad (\text{KNOWN} . C) \quad D \quad ae(p) \in A \quad B}{(\text{true} . S) \quad E \quad C \quad D \quad A \quad B}$$

$$92. \quad \frac{(p . S) \quad E \quad (\text{KNOWN} . C) \quad D \quad ae(p) \notin A \quad B}{(\text{false} . S) \quad E \quad C \quad D \quad A \quad B}$$

Explicitly when  $p$  is a proposition yet it is not in the assumption base.

$$93. \quad \frac{(x . S) \quad E \quad (\text{KNOWN} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a proposition.

$$94. \quad \frac{(v . S) \quad E \quad (\text{FREEAB} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

If the variable  $v$  is free within  $A$ .

$$95. \quad \frac{(v . S) \quad E \quad (\text{FREEAB} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

If the variable  $v$  is not free within  $A$ .

$$96. \quad \frac{(v p . S) \quad E \quad (\text{FREEVAR} . C) \quad D}{(\text{true } v p . S) \quad E \quad C \quad D}$$

If the variable  $v$  is free within the proposition  $p$ .

$$97. \quad \frac{(v p . S) \quad E \quad (\text{FREEVAR} . C) \quad D}{(\text{false } v p . S) \quad E \quad C \quad D}$$

If the variable  $v$  is not free within the proposition  $p$ .

$$98. \quad \frac{(v p . S) \quad E \quad (\text{FREEVAR} . C) \quad D}{(\text{ERROR } v p . S) \quad E \quad C \quad D}$$

If  $v$  is not a variable or  $p$  is not a proposition.

### 3.2.14 Tests

$$99. \quad \frac{([\ ] . S) \quad E \quad (\text{TESTNIL} . C) \quad D}{(\text{true} [\ ] . S) \quad E \quad C \quad D}$$

$$100. \quad \frac{(x . S) \quad E \quad (\text{TESTNIL} . C) \quad D}{(\text{false } x . S) \quad E \quad C \quad D}$$

Where  $x$  is not an empty list.

$$101. \quad \frac{([x_1 \dots x_n] . S) \quad E \quad (\text{TESTLIST} . C) \quad D}{(\text{true } [x_1 \dots x_n] . S) \quad E \quad C \quad D}$$

$$102. \quad \frac{(x . S) \quad E \quad (\text{TESTLIST} . C) \quad D}{(\text{false } x . S) \quad E \quad C \quad D}$$

Where  $x$  is not a list.

$$103. \quad \frac{(\text{true} . S) \quad E \quad (\text{TESTBOOL} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

$$104. \quad \frac{(\text{false} . S) \quad E \quad (\text{TESTBOOL} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

$$105. \quad \frac{(x . S) \quad E \quad (\text{TESTBOOL} . C) \quad D}{(\text{ERROR } x . S) \quad E \quad C \quad D}$$

Where  $x$  is neither true nor false.

$$106. \quad \frac{(v . S) \quad E \quad (\text{TESTVAR} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

$$107. \quad \frac{(x . S) \quad E \quad (\text{TESTVAR} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a variable.

$$108. \quad \frac{(t . S) \quad E \quad (\text{TESTTERM} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

$$109. \quad \frac{(x . S) \quad E \quad (\text{TESTTERM} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Where,  $x$  is not a term.

$$110. \quad \frac{((\text{not } p) . S) \quad E \quad (\text{TESTNOT} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

$$111. \quad \frac{(x . S) \quad E \quad (\text{TESTNOT} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Where,  $x$  is not a proposition of the form  $(\text{not } p)$ .

$$112. \quad \frac{((\text{and } p_1 p_2) . S) \quad E \quad (\text{TESTAND} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

$$113. \quad \frac{(x . S) \quad E \quad (\text{TESTAND} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a proposition of the form  $(\text{and } p_1 p_2)$ .

$$114. \quad \frac{((\text{if } p_1 p_2) . S) \quad E \quad (\text{TESTIF} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

$$115. \quad \frac{(x . S) \quad E \quad (\text{TESTIF} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a proposition of the form  $(\text{if } p_1 p_2)$ .

$$116. \quad \frac{((\text{iff } p_1 p_2) . S) \quad E \quad (\text{TESTIFF} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

$$117. \quad \frac{(x . S) \quad E \quad (\text{TESTIFF} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Where,  $x$  is not a proposition of the form  $(\text{iff } p_1 p_2)$ .

$$118. \quad \frac{((\text{or } p_1 p_2) . S) \quad E \quad (\text{TESTOR} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

$$119. \quad \frac{(x . S) \quad E \quad (\text{TESTOR} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a proposition of the form  $(\text{or } p_1 \ p_2)$ .

$$120. \quad \frac{((\text{forall } v \ p) . S) \quad E \quad (\text{TESTFORALL} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

$$121. \quad \frac{(x . S) \quad E \quad (\text{TESTFORALL} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a proposition of the form  $(\text{forall } v \ p)$ .

$$122. \quad \frac{((\text{exists } v \ p) . S) \quad E \quad (\text{TESTEXISTS} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

$$123. \quad \frac{(x . S) \quad E \quad (\text{TESTEXISTS} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Where  $x$  is not a proposition of the form  $(\text{exists } v \ p)$ .

$$124. \quad \frac{(x_T . S) \quad E \quad (\text{TESTTYPE } T . C) \quad D}{(\text{true } x_T . S) \quad E \quad C \quad D}$$

$$125. \quad \frac{(x . S) \quad E \quad (\text{TESTTYPE } T . C) \quad D}{(\text{false } x . S) \quad E \quad C \quad D}$$

Test that  $x$  is an Athena value of type  $T$ , where  $T$  is one of: A for atom, F for function, M for method, P for proposition, N for prop-con, Q for quantifier, S for substitution, Y for symbol, C for cell, R for char, and U for unit.

Note that lists, terms, and variables have their own testing operations.

$$126. \quad \frac{(p_2 \ p_1 . S) \quad E \quad (\text{ALPHAEQ} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

Where  $p_1$  and  $p_2$  are alphabetically equal propositions.

$$127. \quad \frac{(p_2 \ p_1 . S) \quad E \quad (\text{ALPHAEQ} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Where  $p_1$  and  $p_2$  are not alphabetically equal propositions.

$$128. \quad \frac{(y \ x . S) \quad E \quad (\text{ALPHAEQ} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where either  $x$  or  $y$  is not a proposition.

$$129. \quad \frac{(x_2 \ x_1 . S) \quad E \quad (\text{VALEQ} . C) \quad D}{(\text{true} . S) \quad E \quad C \quad D}$$

Where  $x_1$  and  $x_2$  are “value equal”. Note that function closure and method closures cannot be compared for equality.

$$130. \quad \frac{(x_2 \ x_1 . S) \quad E \quad (\text{VALEQ} . C) \quad D}{(\text{false} . S) \quad E \quad C \quad D}$$

Where  $x_1$  and  $x_2$  are not “value equal”.

$$131. \quad \frac{(x_2 \ x_1 . S) \quad E \quad (\text{VALEQ} . C) \quad D}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

Where  $x_1$  and  $x_2$  contain closures that are compared for equality.

### 3.3 Pattern Matching Operations and Transitions

The pattern matching operations are complex in and of themselves. Just as there are special transitions for handling the pseudo-value `ERROR`, there are special transitions for when pattern matching fails.

The discriminant value is denoted *dis*. Angle brackets  $\langle \rangle$  denote a special map-stack object used only in pattern matching. It denotes both a mapping of identifiers (pattern variables) to values and a stack of values that are pending because of a `bind` pattern. A  $\langle \rangle$  specifically denotes both an empty map-stack, whereas  $\langle * \rangle$  denotes *any* map-stack.

$$132. \quad \frac{(dis \ . S) \quad E \quad (\text{MATCH} . C) \quad D}{(dis \ \langle \rangle . S) \quad E \quad C \quad (dis \ \text{match-token} . D)}$$

$$133. \quad \frac{(\text{match-fail} \ \langle * \rangle . S) \quad E \quad (\text{MATCHNEXT} . C)}{(dis \ \langle \rangle . S) \quad E \quad C} \\ \frac{(\dots \ dis \ \text{match-token} . D)}{(dis \ \text{match-token} . D)}$$



$$134. \frac{(x \langle * \rangle . S) \quad E \quad (\text{MATCHEND} . C) \quad (\dots \text{dis match-token} . D)}{(\text{ERROR} . S) \quad E \quad C \quad D}$$

The ERROR here is that the matched failed.

$$135. \frac{\frac{(\langle * \rangle . S) \quad E \quad (\text{MAPCLOSURE } n \ c' . C)}{\text{NIL} \quad ([v_1 \dots v_n] . E) \quad c'}}{(\text{dis match-token} . D)} \\ \frac{}{(S \ E \ c_{\text{clean}} . D)}$$

Where  $c_{\text{clean}}$  is the code list  $C$ , but with everything up to and including

**MATCHEND** removed. Here, a function closure of arity  $n$  is created and applied with the  $n$  values in the map-stack  $\langle * \rangle$ .

$$136. \frac{\frac{(\langle * \rangle . S) \quad E \quad (\text{MAPDCLOSURE } n \ c' . C)}{\text{NIL} \quad ([v_1 \dots v_n] . E) \quad c'}}{(\text{dis match-token} . D)} \\ \frac{}{(S \ E \ c_{\text{clean}} . D)}$$

Where  $c_{\text{clean}}$  is the code list  $C$ , but with everything up to and including

**MATCHEND** removed. Here, a method closure of arity  $n$  is created and applied with the  $n$  values in the map-stack  $\langle * \rangle$ .

### 3.3.1 Simple Patterns

$$137. \frac{(\text{dis } \langle * \rangle . S) \quad E \quad (\text{MATCHANY} . C) \quad D}{(\langle * \rangle . S) \quad E \quad C \quad D}$$

$$138. \frac{(\text{dis } \langle * \rangle . S) \quad E \quad (\text{MATCHI } i . C) \quad D}{(\langle * , i \mapsto \text{dis} \rangle . S) \quad E \quad C \quad D}$$

If  $\langle * \rangle$  did not have a mapping for identifier  $i$ , then add such a mapping.

If there is a mapping from identifier  $i$  to some value  $V$  in the map-stack  $\langle * \rangle$ , this transition happens if  $V$  is value equal to  $\text{dis}$ . If closures are compared as a result of the value equality test, ERROR is returned.

$$139. \quad \frac{(dis \langle *, i \mapsto x \rangle . S) \quad E \quad (MATCHI \ i . C) \quad D}{(\text{match-fail } \langle *, i \mapsto x \rangle . S) \quad E \quad C \quad D}$$

Where  $i$  is mapped to value  $x$  not equal to value  $dis$ .

$$140. \quad \frac{(\text{true } dis \langle * \rangle . S) \quad E \quad (MATCHSOME \ i . C) \quad D}{(\langle *, i \mapsto dis \rangle . S) \quad E \quad C \quad D}$$

Map  $i$  to  $dis$ ; lose any previous mapping of  $i$  that may have been in  $\langle * \rangle$ .

$$141. \quad \frac{(x \ dis \langle * \rangle . S) \quad E \quad (MATCHSOME \ i . C) \quad D}{(\text{match-fail } \langle * \rangle . S) \quad E \quad C \quad D}$$

Where  $x$  is not true.

$$142. \quad \frac{(dis \langle *, stack \rangle . S) \quad E \quad (MATCHBIND \ i . C) \quad D}{(\langle *, i.dis.stack \rangle . S) \quad E \quad C \quad D}$$

Save the identifier  $i$  and the discriminant  $dis$  to the stack in the map-stack  $\langle * \rangle$ .

$$143. \quad \frac{(\langle *, i.dis.stack \rangle . S) \quad E \quad (MATCHBINDCOMMIT \ i . C) \quad D}{(\langle *, i \mapsto dis, stack \rangle . S) \quad E \quad C \quad D}$$

Map  $i$  to the saved  $dis$ ; lose any previous mapping of  $i$  that may have been in  $\langle * \rangle$ .

$$144. \quad \frac{(x \ dis \langle * \rangle . S) \quad E \quad (MATCHEQUAL . C) \quad D}{(\langle * \rangle . S) \quad E \quad C \quad D}$$

Where  $x$  and  $dis$  are equal values. (Alphabetically equivalent, if  $x$  and  $dis$  are propositions.)

$$145. \quad \frac{(x \ dis \langle * \rangle . S) \quad E \quad (MATCHEQUAL . C) \quad D}{(\text{match-fail } \langle * \rangle . S) \quad E \quad C \quad D}$$

Where  $x$  and  $dis$  are not equal values.

$$146. \quad \frac{(x \ dis \langle * \rangle . S) \quad E \quad (MATCHEQUAL . C) \quad D}{(\text{ERROR } \langle * \rangle . S) \quad E \quad C \quad D}$$

Where  $x$  and  $dis$  either both function closures or both method closures.

147. 
$$\frac{([x_1] \langle * \rangle . S) \quad E \quad (\text{MATCHLISTOF} . C) \quad D}{([x_1 []] \langle * \rangle . S) \quad E \quad C \quad D}$$
148. 
$$\frac{([x_1 x_2 \dots x_n] \langle * \rangle . S) \quad E \quad (\text{MATCHLISTOF} . C) \quad D}{([x_1 [x_2 \dots x_n]] \langle * \rangle . S) \quad E \quad C \quad D}$$
149. 
$$\frac{(x \langle * \rangle . S) \quad E \quad (\text{MATCHLISTOF} . C) \quad D}{(\text{match-fail} \langle * \rangle . S) \quad E \quad C \quad D}$$

If  $x$  is not a list, or if  $x$  is an empty list.

### 3.3.2 Bracket Patterns

150. 
$$\frac{([ ] \langle * \rangle . S) \quad E \quad (\text{MATCHLIST } 0 . C) \quad D}{(\langle * \rangle . S) \quad E \quad C \quad ([ ] \text{ match-list-token } . D)}$$
151. 
$$\frac{\frac{(list \langle * \rangle . S) \quad E \quad (\text{MATCHLIST } n . C)}{(list[0] \langle * \rangle . S) \quad E \quad C} \quad D}{(list \text{ match-list-token } . D)}$$

Where  $n > 0$  and is equal to the length of the discriminant list.

152. 
$$\frac{(dis \langle * \rangle . S) \quad E \quad (\text{MATCHLIST } n . C) \quad D}{(\text{match-fail} \langle * \rangle . S) \quad E \quad C \quad D}$$

Where  $dis$  is either not a list, or a list whose length is not equal to  $n$ .

153. 
$$\frac{\frac{(\langle * \rangle . S) \quad E \quad (\text{MATCHLISTNEXT } m . C)}{(list[m] \langle * \rangle . S) \quad E \quad C} \quad D}{(list \text{ match-list-token } . D)}$$

Where  $0 \leq m <$  the length of  $list$ .

154. 
$$\frac{(\langle * \rangle . S) \quad E \quad (\text{MATCHLISTEND} . C) \quad (list \text{ match-list-token } . D)}{(\langle * \rangle . S) \quad E \quad C \quad D}$$

Clean up the dump after the list has been matched.

### 3.3.3 Compound Patterns

$$155. \frac{(t \langle * \rangle . S) \quad E \quad (\text{MATCHASLIST nested} . C) \quad D}{([f [t_1 \dots t_n] \langle * \rangle] . S) \quad E \quad C \quad D}$$

Where term (or atom)  $t$  is not a variable. If  $t$  is nullary, then  $n = 0$  and  $[t_1 \dots t_n]$  is merely the empty list  $[]$ .

$$156. \frac{(x \langle * \rangle . S) \quad E \quad (\text{MATCHASLIST nested} . C) \quad D}{(\text{match-fail} \langle * \rangle . S) \quad E \quad C \quad D}$$

Fail for any other value.

$$157. \frac{(t \langle * \rangle . S) \quad E \quad (\text{MATCHASLIST flatten} . C) \quad D}{([f t_1 \dots t_n] \langle * \rangle . S) \quad E \quad C \quad D}$$

Where term (or atom)  $t$  is not a variable. If  $t$  is nullary, then  $n = 0$  and  $[f t_1 \dots t_n]$  is the list  $[f]$ .

$$158. \frac{(x \langle * \rangle . S) \quad E \quad (\text{MATCHASLIST flatten} . C) \quad D}{(\text{match-fail} \langle * \rangle . S) \quad E \quad C \quad D}$$

Fail for any other value.

$$159. \frac{(p \langle * \rangle . S) \quad E \quad (\text{MATCHQPROP} . C) \quad D}{(q \langle * \rangle . S) \quad E \quad C \quad ([v_1 \dots v_k] q b \text{ q-token} . D)}$$

The operation `MATCHQPROP` decomposes the proposition  $p$  into  $q$ , the variable list  $[v_1 \dots v_k]$ , and the proposition  $b$  as noted in Section 3. In particular, the value  $q$  is either the quantifier `forall`, the quantifier `exists`, or the unit value — indicating that the quantifier is undetermined.

$$160. \frac{(x \langle * \rangle . S) \quad E \quad (\text{MATCHQPROP} . C) \quad D}{(\text{match-fail} \langle * \rangle . S) \quad E \quad C \quad (\text{q-token} . D)}$$

If  $x$  is not a proposition, or if  $x$  is the pseudo-value “match-fail”.

$$161. \quad \frac{(\text{forall } \langle * \rangle . S) \quad E \quad (\text{MATCHQ1S } i . C) \quad D}{(\langle *, i \mapsto \text{forall} \rangle . S) \quad E \quad C \quad D}$$

$$162. \quad \frac{(\text{forall } \langle *, i \mapsto x \rangle . S) \quad E \quad (\text{MATCHQ1S } i . C) \quad D}{(\text{match-fail } \langle *, i \mapsto x \rangle . S) \quad E \quad C \quad D}$$

If identifier  $i$  is already bound to  $x$ , and  $x$  is not **forall**.

$$163. \quad \frac{(\text{exists } \langle * \rangle . S) \quad E \quad (\text{MATCHQ1S } i . C) \quad D}{(\langle *, i \mapsto \text{exists} \rangle . S) \quad E \quad C \quad D}$$

$$164. \quad \frac{(\text{exists } \langle *, i \mapsto x \rangle . S) \quad E \quad (\text{MATCHQ1S } i . C) \quad D}{(\text{match-fail } \langle *, i \mapsto x \rangle . S) \quad E \quad C \quad D}$$

If identifier  $i$  is already bound to  $x$ , and  $x$  is not **exists**.

$$165. \quad \frac{(\langle \rangle \langle * \rangle . S) \quad E \quad (\text{MATCHQ1S } i . C) \quad D}{(\text{match-fail } \langle * \rangle . S) \quad E \quad C \quad D}$$

Fail if the expected quantifier to be bound to identifier  $i$  is instead the unit value.

$$166. \quad \frac{(\text{forall } \langle * \rangle . S) \quad E \quad (\text{MATCHQ1 forall } . C) \quad D}{(\langle * \rangle . S) \quad E \quad C \quad D}$$

$$167. \quad \frac{(\langle \rangle \langle * \rangle . S) \quad E \quad (\text{MATCHQ1 forall } . C) \quad D}{(\langle * \rangle . S) \quad E \quad C \quad D}$$

$$168. \quad \frac{(\text{exists } \langle * \rangle . S) \quad E \quad (\text{MATCHQ1 forall } . C) \quad D}{(\text{match-fail } . S) \quad E \quad C \quad D}$$

$$169. \quad \frac{(\text{exists } \langle * \rangle . S) \quad E \quad (\text{MATCHQ1 exists } . C) \quad D}{(\langle * \rangle . S) \quad E \quad C \quad D}$$

$$170. \quad \frac{(\langle \rangle \langle * \rangle . S) \quad E \quad (\text{MATCHQ1 exists } . C) \quad D}{(\langle * \rangle . S) \quad E \quad C \quad D}$$

$$171. \quad \frac{(\text{forall } \langle * \rangle . S) \quad E \quad (\text{MATCHQ1 exists } . C) \quad D}{(\text{match-fail } . S) \quad E \quad C \quad D}$$

$$172. \quad \frac{\frac{(\langle * \rangle . S) \quad E \quad (\text{MATCHQ2INIT } c_{Q2} . C)}{S \quad E \quad c_{Q2}}}{\frac{([v_1 \dots v_k] \ q \ b \ \text{q-token} . D)}{(k \ [v_1 \dots v_k] \ \langle * \rangle \ c_{Q2} \ \text{q2-token} \ C \ q \ b \ \text{q-token} . D)}}$$

Set up the dump  $D$ , save the old code list  $C$  and current map-stack  $\langle * \rangle$ , and load the code list  $c_{Q2}$ .

$$173. \quad \frac{\frac{S \quad E \quad (\text{MATCHQ2START} . C)}{([v_1 \dots v_i] \ \langle * \rangle . S) \quad E \quad C}}{\frac{(i \ [v_1 \dots v_k] \ \langle * \rangle \ c_{Q2} \ \text{q2-token} \ c_{cont} \ q \ b \ \text{q-token} . D)}{(i \ [v_1 \dots v_k] \ \langle * \rangle \ c_{Q2} \ \text{q2-token} \ c_{cont} \ q \ b \ \text{q-token} . D)}}$$

Load the new variable list and saved mapping.

$$174. \quad \frac{\frac{(\langle * \rangle . S) \quad E \quad (\text{MATCHQ2END} . C)}{(\langle * \rangle . S) \quad E \quad c_{cont}}}{\frac{(i \ [v_1 \dots v_k] \ \langle * \rangle_{saved} \ c_{Q2} \ \text{q2-token} \ c_{cont} \ q \ b \ \text{q-token} . D)}{(i \ [v_1 \dots v_k] \ q \ b \ \text{q-token} . D)}}$$

Clean up the dump  $D$  and restore the continue code list.

$$175. \quad \frac{\frac{(\text{match-fail } \langle * \rangle . S) \quad E \quad (\text{MATCHQ2END} . C)}{(\text{match-fail } \langle * \rangle . S) \quad E \quad c_{cont}}}{\frac{(\dots \ 0 \ [v_1 \dots v_k] \ \langle * \rangle_{saved} \ c_{Q2} \ \text{q2-token} \ c_{cont} \ q \ b \ \text{q-token} . D)}{(\text{q-token} . D)}}$$

Where “...” does not contain q2-token. Here, all attempts at matching the variable list has failed, and so the match-fail gets propagated outside of the Q2 matching loop.

$$176. \quad \frac{\frac{(\text{match-fail } \langle * \rangle . S) \quad E \quad (\text{MATCHQ2END} . C)}{S \quad E \quad c_{Q2}}}{\frac{(\dots \ i \ [v_1 \dots v_k] \ \langle * \rangle_{saved} \ c_{Q2} \ \text{q2-token} \ c_{cont} \ q \ b \ \text{q-token} . D)}{(i - 1 \ [v_1 \dots v_k] \ \langle * \rangle_{saved} \ c_{Q2} \ \text{q2-token} \ c_{cont} \ q \ b \ \text{q-token} . D)}}$$

Where “...” does not contain q2-token and  $i > 0$ . Here,  $i$  is decremented and the Q2 matching loop is re-entered.

$$177. \quad \frac{\frac{(\langle * \rangle . S) \quad E \quad (\text{MATCHQ3} . C)}{(p \langle * \rangle . S) \quad E \quad C} \quad \frac{(i [v_1 \dots v_k] q b \text{ q-token} . D)}{(\text{q-token} . D)}}{(\text{q-token} . D)}$$

Where proposition  $p$  is created by requantifying proposition  $b$  with  $q$  over the remaining variables  $[x_{i+1} \dots x_k]$ , as per Section 3. Note that if  $q$  is the unit value, then it had better also be the case that  $i = k = 0$ .

$$178. \quad \frac{(\langle * \rangle . S) \quad E \quad (\text{MATCHQEND} . C) \quad (\text{q-token} . D)}{(\langle * \rangle . S) \quad E \quad C \quad D}$$

$$179. \quad \frac{(\text{match-fail} \langle * \rangle . S) \quad E \quad (\text{MATCHQEND} . C) \quad (\dots \text{ q-token} . D)}{(\text{match-fail} \langle * \rangle . S) \quad E \quad C \quad D}$$

Where “...” does not contain q-token.

### 3.3.4 Match-Fail Transitions

Transitions for match-fail are special in the same way that transitions for ERROR are special. When the pseudo-value match-fail is on top of the  $S$  stack, the machine exclusively deals with this transition until an operation that handles match-fail is found.

$$180. \quad \frac{(\text{match-fail} \langle * \rangle . S) \quad E \quad (\dots \text{ TERMINATOR} . C) \quad D}{(\text{match-fail} \langle * \rangle . S) \quad E \quad (\text{TERMINATOR} . C) \quad D}$$

Where *TERMINATOR* is one of MATCHNEXT, MATCHEND, MATCHQPROP, MATCHQ2END, or MATCHQEND, and “...” does not contain any of these operations. It is the job of each of these match operations to clean up the dump  $D$  if match-fail is on top of  $S$ .

The next chapter discusses how Athena phrases are compiled into these virtual machine operations and provides code for all of Athena’s primitive closures.

## Chapter 4

# Compilation and Code

The compiler for the Athena virtual machine accepts an abstract syntax tree (AST) generated by a parser of Athena phrases. The AST has forms for each of the phrases as listed in the syntax given in Figure 2.1 (page 16). This chapter specifies how each phrase compiles into a code list of virtual machine operations. It also provides the code lists of all of the primitive functions and methods of Athena.

For the Lispkit Lisp interpreter, Henderson states one of the key ideas for implementing a functional programming language on an SECD machine, what he calls the net-effect property [Hen80]. The idea is quite simple and powerful: a series of state transitions should have the net-effect of placing exactly one value on top of the  $S$  stack. Simply loading a value via the LD operation obviously satisfies this property. However, so should the application of a function closure on a list of arguments. During the application of the closure, there is certainly activity on other stacks, but the net-effect after the application has completed will be the pushing of the return value on top of  $S$ . This notion has been extended for Athena's virtual machine. All of the phrases of Athena are compiled with the net-effect property in mind, including such constructs as while loops, try blocks, pattern matching, and both function and method closure application.

Both Henderson's machine and the Athena virtual machine have the same mechanism for adding new bindings to the environment, and that is via the application of a closure. For Lispkit, the bindings done by `let` and `letrec` are added by creating a closure and almost immediately applying a closure. This notion has also been borrowed by the Athena virtual machine for `let`, `letrec`, `dlet`, `dletrec`, the primitive deductive forms `pick-any` and `pick-witness`, and the binding of the



pattern variables upon successful matching.

## 4.1 The Compiler

Compilation of an Athena phrase happens with respect to the namespace of an environment. The namespace is structurally similar to the environmental stack  $E$ , except instead of being a list-of-lists of values, it is a list-of-lists of identifiers. Over the course of compiling a given phrase, if an identifier is referenced, the namespace is checked in order to provide the offset into the environment with the identifier's value. This offset, a pair of integers, is used by the LD operation to obtain the value when the code list is executed. The value associated by an identifier is not known by the compiler; it will only be computed while the VM is running. However, the compiler knows *where* in the environment the value will be, because the namespace during compilation structurally mirrors the environment stack during execution.

The compiler detects whether a given identifier is unbound by consulting the namespace. If the identifier is not in the namespace, it may still be a legal identifier if it is the name of a prop-con, quantifier, the empty substitution, or function symbol. There are various identifiers that are bound in the namespace (and thus environment) when Athena starts, and additional identifiers might be later bound depending on what top-level directives have been issued. The Athena virtual machine handles both of these by setting the initial namespace and environment to be a list of two lists. The top-level namespace contains all of the identifiers in the top-level environment. Just like the top-level environment, it too is extended upon successful use of the `define-numeric-operations` and `define` directives.

During execution of a code list, the environment stack changes. The compiler directs how these changes are made by which operations are generated for the code list. However, the compiler must also keep track of these changes in its namespace so as to accurately produce code that can reference identifiers. This is why compilation is with respect to both a particular phrase being compiled *and* a namespace — the namespace reflects the environment that the phrase will be in when it is evaluated.

For the compilation of each phrase, the namespace must be explicitly designated. In general, the compilation will be with respect to the current namespace. The compilation of a phrase  $F$  with respect to the current namespace is denoted  $[[F]]$ . A phrase might also be compiled with respect to an extended version of the namespace. These exten-

sions are always to the front of the namespace — the new identifiers will be at offset  $(0, n)$  and the older ones that were at  $(m, n)$  will afterwards be at  $(m + 1, n)$ . The compilation of a phrase  $F$  in an namespace extended with the list of identifiers  $(I_1 \dots I_n)$  is denoted  $[[F^{(I_1 \dots I_n)}]]$ .

Semantically, there is the distinction between evaluating a deduction for its conclusion, and that of adding the conclusion to the assumption base. The Athena virtual machine does not have this distinction. To the machine, in order to return the correct value for a deduction, that deduction might have to call other deductions, and these further calls will require that the assumption base be extended to hold their conclusions; all of this, just to obtain the *value* of the outer-most conclusion. Therefore, the machine does not attempt to determine for what effect a deduction is being evaluated. The compilation of a deduction returns code that *always* adds the conclusion as a side-effect of evaluating the deduction. If this were done naïvely, then the semantics of Athena would be violated.

The compiler determines which deductions are being evaluated merely for their value and which deductions are being evaluated for both their value and their addition to the assumption base. Those deductions being evaluated merely for their value — or more generally, phrases that might be deductions — are wrapped around a DSTART/DRESTORE pair, which saves and then restores the assumption base after the phrase has returned a value.

Thus, during the compilation of expressions, a phrase that is being compiled for its value will be wrapped around a DSTART/DRESTORE pair, so that only the resulting value remains; effectively, it protects the assumption base from whatever the phrase might otherwise do to it. The compilation of deductions, since to the machine they will always add their conclusion to the assumption base (upon success), will be wrapped around a DSTART/DEND pair, and it is the DEND that adds the conclusion to the assumption base. A failed deduction will instead end with a DRESTORE, load an error, or both.

#### 4.1.1 Expressions

For a given expression  $E$ , this list specifies the compilation  $[[E]]$ :

( )

TRIV

The unit value is generated via the TRIV operation.

*I*

- If *I* is bound in the current namespace,  
LD.*m.n*
- If *I* is a prop-con, quantifier, or function symbol,  
LDPRIM.*I*
- If *I* is the empty substitution,  
NULLSUB
- Otherwise, a compile-time error is produced.

The namespace is searched, and offset of the first occurrence of the identifier *I* is returned. These are taken as arguments to the LD operation, which will retrieve the value during execution. The first occurrence is important in order to have the proper semantics for a lexically-scoped environment.

As mentioned earlier, if the identifier is not in the namespace the compiler checks if the identifier is a prop-con, quantifier, the empty substitution, or a function symbol. For the cases, the code LDPRIM.*I* is generated instead.

*S*

A string *S* is decomposed by the parser into a list of characters; it does not exist as a distinct object in the AST.

'*I*

LDPRIM.*c*

A character '*I* becomes a LDPRIM of the character that the identifier represents.

?*I*

LDV.*I*

A variable ?*I* is loaded by name.

'*I*

LDPRIM.'*I*

The meta-identifier '*I* is just a function symbol.

(cell *F*)

[[*F*]].REF

To create a cell, REF expects the cell's value to be on top of the *S* stack. By having the code for *F* precede the REF, the net-effect

property guarantees that the value of  $F$  will be on  $S$  for the REF operation.

(ref  $E$ )

[[ $E$ ]].AT

Likewise, to reference a cell,  $E$  is evaluated and then the AT operation is called to obtain the value from whatever cell  $E$  happens to evaluate to.

(set!  $E F$ )

[[ $E$ ]].[[ $F$ ]].REFASSIGN.TRIV

The semantics specifies that `set!` return the unit value, hence the TRIV in the compilation.

(function ( $I^*$ )  $E$ )

CLOSURE. $n$ .([[ $E^{(I^*)}$ ]].RTN)

After checking that each  $I$  is unique, a code list is generated for  $E$  in a namespace extended by closure's  $n$  arguments. The RTN is required to “return” from the closure.

( $E F^*$ )

[[ $E$ ]].NIL.  
DSTART.[[ $F_1$ ]].DRESTORE.POST.  
DSTART.[[ $F_2$ ]].DRESTORE.POST ...  
DSTART.[[ $F_n$ ]].DRESTORE.POST.APPLY

After generating the value that will be applied to, the result of each argument phrase is computed and then added to the argument list. Note the use of DSTART/DRESTORE so that each  $F_i$  is only being generated only for its value; if any were deductions, their results are not added to the assumption base when the value resulting from  $E$  gets applied. Finally, the APPLY starts the application, as the remaining two elements on the stack are the argument list and the value being applied.

(method ( $I^*$ )  $D$ )

DCLOSURE. $n$ .([[ $D^{(I^*)}$ ]].RTN)

After checking that each  $I$  is unique, a code list is generated for  $D$  in a namespace extended by closure's  $n$  arguments. The RTN is required to “return” from the closure.

(**check** ( $F$   $E$ )\*)

By induction on the number of phrase-expression pairs:

0.  
LDERROR.“check”
1.  
DSTART.[[ $F_1$ ]].DRESTORE.SEL.  
  ([[ $E_1$ ]].JOIN).  
  (LDERROR.“check”.JOIN)
2.  
DSTART.[[ $F_1$ ]].DRESTORE.SEL.  
  ([[ $E_1$ ]].JOIN).  
  (DSTART.[[ $F_2$ ]].DRESTORE.SEL.  
    ([[ $E_2$ ]].JOIN).  
    (LDERROR.“check”.JOIN).JOIN)
3. ...

Thus as each phrase  $F_i$  is evaluated, the code list will proceed to either  $E_i$  if the result was true or to the next phrase otherwise. And if all phrases fail, an error with the text string “check” is returned. The reserved identifier **else**, which may appear as  $F_n$ , compiles to LDTRUE.

(**match**  $F$  ( $\pi$   $E$ )\*)

The compilation of match expressions is discussed in Section 4.1.3.

(**let** (( $I$   $F$ )\*  $E$ )

By induction on the number of identifier-phrase pairs:

0.  
[[ $E$ ]]
1.  
CLOSURE.1.([[ $E^{(I_1)}$ ]].RTN).  
NIL.DSTART.[[ $F_1$ ]].DRESTORE.POST.APPLY
2.  
CLOSURE.1.(  
  CLOSURE.1.([[ $E^{(I_2)(I_1)}$ ]].RTN).  
  NIL.DSTART.[[ $F_2^{(I_1)}$ ]].DRESTORE.POST.APPLY.RTN)  
NIL.DSTART.[[ $F_1$ ]].DRESTORE.POST.APPLY

### 3. ...

This is basically the generation of a function closure followed by its application. Note that the compilation of  $E$  happens in an environment extended with each  $I$ . This compilation for `let` is quite different from Henderson's SECD machine, because Athena's semantics are different than those of Lispkit.

```
(letrec ((I F)* ) E)
```

```
DUMMY.CLOSURE.n.([[E(I1...In)]].RTN).NIL.
  DSTART.([[F1(I1...In)]].DRESTORE.POST.
  DSTART.([[F2(I1...In)]].DRESTORE.POST. ...
  DSTART.([[Fn(I1...In)]].DRESTORE.POST.RAPPLY
```

Each phrase and expression is compiled in an environment extended with all of the identifiers  $I_1 \dots I_n$ . The DUMMY generates a dummy environment during the evaluation of each  $F_i$ , and the RAPPLY replaces the dummy environment with the argument list at runtime. The DUMMY/RAPPLY pair together “tie the knot” of the `letrec`.

```
(begin F+)
```

```
DSTART.([[F1]].DRESTORE.POP. ...
DSTART.([[Fn-1]].DRESTORE.POP.
DSTART.([[Fn]].DRESTORE
```

The value of each phrase  $F_i$  is computed and then discarded, except for the last phrase  $F_n$ , which is left as the return value for the `begin` expression.

```
(while F1 F2)
```

```
WHILE.(DSTART.([[F1]].DRESTORE.WHILETEST.
  DSTART.([[F2]].DRESTORE.WHILELOOP)
```

The WHILE sets up the loop, WHILETEST exits the loop (both on success, or in case of a non-boolean being returned by  $F_1$ ), and WHILELOOP reloads the loop code.

```
(& F+)
```

By induction on the number of phrases:

1.
  - DSTART.[[ $F_1$ ]].DRESTORE.TESTBOOL.SEL.
  - (LDTRUE.JOIN).
  - (LDFALSE.JOIN)
2.
  - DSTART.[[ $F_1$ ]].DRESTORE.TESTBOOL.SEL.
  - (DSTART.[[ $F_2$ ]].DRESTORE.TESTBOOL.SEL.
  - (LDTRUE.JOIN).
  - (LDFALSE.JOIN).JOIN).
  - (LDFALSE.JOIN)
3. ...

As long as each phrase results in the value **true**, then the LDTRUE operation will be reached and **true** will be returned as the value of the logical and expression. If any one phrase has the value **false**, then LDFALSE is executed and **false** is returned. The TESTBOOL operation will generate an error if a reached phrase generates a non-Boolean value, as required by the semantics of logical and.

(||  $F^+$ )

By induction on the number of phrases:

1.
  - DSTART.[[ $F_1$ ]].DRESTORE.TESTBOOL.SEL.
  - (LDTRUE.JOIN).
  - (LDFALSE.JOIN)
2.
  - DSTART.[[ $F_1$ ]].DRESTORE.TESTBOOL.SEL.
  - (LDTRUE.JOIN).
  - (DSTART.[[ $F_2$ ]].DRESTORE.TESTBOOL.SEL.
  - (LDTRUE.JOIN).
  - (LDFALSE.JOIN).JOIN)
3. ...

Structurally and semantically, the logical or expression is similar to the logical and, above. Instead, if any phrase evaluates to **true**, then a LDTRUE is reached, and all phrases must evaluate to **false** for a LDFALSE to be reached.

### 4.1.2 Deductions

For a given deduction  $D$ , this list specifies the compilation  $[[D]]$ :

(apply-method  $E F^*$ )

```
DSTART. [[E]].NIL.  
  [[F1]].POST. ...  
  [[Fn]].POST.DAPPLY.DEND
```

Structurally, this is quite similar to the application of expressions, but without the `DSTART/DRESTORE` pairs as there is no need to protect the assumption base during the evaluation of each phrase  $F_i$ . If any  $F_i$  is a deduction, then its conclusion will be added to the assumption base. The `DAPPLY` operations enforces that  $E$  be a method closure. Finally, the outer `DSTART/DEND` pair will add the result of this deduction to the assumption base.

(assume  $F D$ )

```
DSTART. [[F]].DRESTORE.  
DSTART.ASSERT. [[D]].DRESTORE.IF.ASSERT
```

The result of  $F$ , either expression or deduction, is added to the assumption base explicitly (the `ASSERT` will fail it if is not a proposition). Then,  $D$  is executed with respect to this extended assumption base. The `IF` generates the conclusion to the assume deduction, which is explicitly added via `ASSERT`.

(suppose-absurd  $F D$ )

```
DSTART. [[F]].DRESTORE.  
DSTART.ASSERT. [[D]].BOOLNOT.SEL.  
  (NOT.DEND.JOIN).  
  (POP.LDERROR. "badsupposeabsurd".DRESTORE.JOIN)
```

The result of  $F$  is explicitly added to the assumption base.  $D$  is evaluated in this extended assumption base. The `BOOLNOT.SEL` makes an explicit test for `false`, in which the negation of  $F$  is returned as a conclusion; otherwise, the error is produced.

(dcheck ( $F D$ )\*)

By induction on the number of phrase-deduction pairs:



0. LDERROR.“dcheck”
1.  $[[F_1]].\text{SEL}.$   
 $(([[D_1]].\text{JOIN}).$   
 $(\text{LDERROR.“dcheck”}.\text{JOIN})$
2.  $[[F_1]].\text{SEL}.$   
 $(([[D_1]].\text{JOIN}).$   
 $(([[F_2]].\text{SEL}.$   
 $(([[D_2]].\text{JOIN}).$   
 $(\text{LDERROR.“dcheck”}.\text{JOIN}).\text{JOIN})$
3. ...

The dcheck deduction is structurally and semantically similar to the check expression. The reserved identifier `else`, which may appear as  $F_n$ , compiles to LDTRUE.

(dmatch  $F (\pi D)^*$ )

The compilation of dmatch deductions is discussed in Section 4.1.3.

(dlet  $((I F)^*) D$ )

By induction on the number of identifier-phrase pairs:

0.  $[[D]]$
1. DCLOSURE.1.  $([[D^{(I_1)}}]].\text{RTN}.$   
 $\text{NIL}.[[F_1]].\text{POST}.\text{DAPPLY}$
2. DCLOSURE.1.(  
DCLOSURE.1.  $([[D^{(I_2)(I_1)}}]].\text{RTN}.$   
 $\text{NIL}.[[F_2^{(I_1)}}]].\text{POST}.\text{DAPPLY}.\text{RTN}$   
 $\text{NIL}.[[F_1]].\text{POST}.\text{DAPPLY}$
3. ...

This is basically the generation of a method closure followed by its application. Note that is structurally similar to `let`, except that there is no need to protect the assumption base with DSTART/DRESTORE pairs.

```

(dletrec ((I F*) D)
  DSTART.DUMMY.DCLOSURE.n.([[D(I1...In)]].RTN).NIL.
  [[F1(I1...In)]].POST.
  [[F2(I1...In)]].POST. ...
  [[Fn(I1...In)]].POST.RDAPPLY.DEND

```

Each phrase and deduction is compiled in an environment extended with all of the identifiers  $I_1 \dots I_n$ . The DUMMY generates a dummy environment during the evaluation of each  $F_i$ , and the RDAPPLY replaced the dummy environment with the argument list at runtime. The DUMMY/RDAPPLY pair together “tie the knot” of the dletrec. Compared to letrec, the dletrec is quite similar, except that there is no need to protect the assumption base. (In fact, RAPPLY and RDAPPLY are exactly the same, except that the former expects a function closure and the latter expects a method closure.)

```

(try D+)
  TRY.([[D1]].RETRY ... RETRY.([[Dn-1]].RETRY.([[Dn]].FAIL

```

Compilation for try blocks is very straightforward. All of the details of try semantics are built into the operations TRY, RETRY, and FAIL. It is the job of these operations to move past the FAIL operation upon successful completion of a deduction, and to reset the machine for the next deduction if the current one fails by producing an ERROR.

```

(dbegin F* D)
  DSTART.([[F1]].POP.([[F2]].POP ... [[D]].DEND

```

A dbegin deduction is also quite straightforward, each phrase  $F$  is evaluated and its return value discarded, and  $D$  is evaluated in an assumption base extended with any deductions that  $F^*$  may have added.

```

(E BY D)
  DSTART.([[E]].DUP.([[D]].ALPHAEQ.SEL.
  (DEND.JOIN).
  (POP.DRESTORE.LDERROR.“by”.JOIN)

```

First expression  $E$  is generated, and then deduction  $D$ . If the results are the same, then the deduction ends with DEND; otherwise, an ERROR is generated.

(generalize-over  $E D$ )

```
DSTART.[[ $E$ ]].DUP.TESTVAR.SEL.  
  (DUP.FREEAB.BOOLNOT.SEL.  
    ([[ $D$ ]].FORALL.DEND.JOIN)  
    (POP.DRESTORE.LDERROR.“generalize-over”.  
      JOIN).JOIN)  
  (POP.DRESTORE.LDERROR.“generalize-overnotvar”.JOIN)
```

The expression  $E$  is evaluated and tested to make sure it is a variable; if it is not, an error with the text “generalize-over not var” is returned. The variable is then tested as to whether it is free in the assumption base; an error with the text “generalize-over” is generated if the variable is free. Finally, the deduction  $D$  is evaluated to obtain an proposition, the FORALL operations generates a forall proposition out of the variable obtained from  $E$  and the conclusion of  $D$ , and concludes with it via DEND.

(pick-any  $I D$ )

```
DSTART.FRESHVAR.DUP.NIL.SWAP.POST.  
DCLOSURE.1.([[ $D^{(I)}$ ]].RTN).SWAP.DAPPLY.FORALL.DEND
```

The pick-any deduction is similar to generalize-over, except that the virtual machine provides the variable (generated by FRESHVAR) via the binding  $I$ . To actually generate the binding, a closure is created and then applied.

(with-witness  $E F D$ )

```
DSTART.[[ $E$ ]].DUP.TESTVAR.SEL.  
  (DUP.DUP.FREEAB.BOOLNOT.SEL.  
    (DSTART.[[ $F$ ]].DUP.TESTEXISTS.SEL.  
      (DUP.KNOWN.SEL.  
        (DUP.prop1.SWAP.prop2.REPLACE.  
          ASSERT.POP.[[ $D$ ]].  
        SWAP.FREEVAR.BOOLNOT.SEL.  
          (POP.DRESTORE.DEND.JOIN)  
          (POP.POP.DRESTORE.DRESTORE.  
            LDERROR.“with-witnesswitnessfreeinconclusion”.  
              JOIN).JOIN)  
          (POP.POP.POP.DRESTORE.DRESTORE.  
            LDERROR.“with-witnessphrasenotinab”.  
              JOIN).JOIN)
```

```

      (POP.POP.POP.DRESTORE.DRESTORE.
      LDERROR.“with-witnessexpectedexists”.JOIN).JOIN)
    (POP.POP.DRESTORE.
      LDERROR.“with-witnessvarfreeinab”.JOIN).JOIN)
  (POP.DRESTORE.LDERROR.“with-witnessnotvar”.JOIN)

```

Half of the code for with-witness is merely to generate the proper error message when and if it fails. Also note that the notation for “prop1” and “prop2” is borrowed from Section 4.2 — they amount to ASLIST.ELEM.1 and ASLIST.ELEM.2, respectively.

First  $E$  is evaluated. It is verified the result is a variable, and that the variable is not in the assumption base. (An extra copy of the variable is left on the stack for use later with REPLACE.) Second,  $F$  is evaluated, tested to be an existential proposition and tested that it is in the assumption base, via KNOWN. Third, the body of the existential proposition from  $F$  has its quantified variable replaced with the variable from  $E$ , and the resulting proposition is added to the assumption base via ASSERT.

Finally, the deduction  $D$  is evaluated in this newly extended assumption base. Assuming that the variable from  $E$  is not free within the resulting conclusion, then the result of  $D$  becomes the return value and sole extension from the assumption base — the DRESTORE restores the old assumption base and then the DEND that follows adds the correct conclusion.

```

(pick-witness  $I$   $F$   $D$ )
DSTART.FRESHVAR.DUP.[[ $F$ ]].DUP.TESTEXISTS.SEL.
  (DUP.KNOWN.SEL.
    (DUP.prop1.SWAP.prop2.REPLACE.ASSERT.POP.
    DUP.NIL.SWAP.POST.DCLOSURE.1.([[ $D^I$ ]].RTN)
    SWAP.DAPPLY.SWAP.FREEVAR.BOOLNOT.SEL.
    (POP.DEND.JOIN)
    (POP.POP.DRESTORE.
      LDERROR.“pick-witnesswitnessfreeinconclusion”.
      JOIN).JOIN)
    (POP.POP.POP.DRESTORE.
      LDERROR.“pick-witnessphrasenotinab”.JOIN).JOIN)
  (POP.POP.POP.DRESTORE.
    LDERROR.“pick-witnessexpectedexists”.JOIN)

```

The pick-witness deduction is similar to with-witness, except that

the virtual machine provides the variable via the binding  $I$ ; again, a closure is used to actually making the binding.

### 4.1.3 Patterns

(match  $F$  ( $\pi$   $E$ )\*)

```
DSTART.[[F]].DEND.MATCH.
  [[ $\pi_1$ ]].MAPCLOSURE. $v_1$ .([[ $E^{(I_1 \dots I_{v_1})}$ ]].RTN).
  MATCHNEXT ...
  [[ $\pi_n$ ]].MAPCLOSURE. $v_n$ .([[ $E^{(I_1 \dots I_{v_n})}$ ]].RTN).
MATCHEND
```

There are  $n$  pattern-expression pairs, and  $v_i$  refers to the number of pattern variables that are within pattern  $\pi_i$ . At compile time, the compiler can distinguish between which identifiers are symbols, prop-cons, or quantifiers, and which identifiers are actually pattern variables.

(dmatch  $F$  ( $\pi$   $D$ )\*)

```
[[F]].MATCH.
  [[ $\pi_1$ ]].MAPDCLOSURE. $v_1$ .([[ $D^{(I_1 \dots I_{v_1})}$ ]].RTN).
  MATCHNEXT ...
  [[ $\pi_n$ ]].MAPDCLOSURE. $v_n$ .([[ $D^{(I_1 \dots I_{v_n})}$ ]].RTN).
MATCHEND
```

There are  $n$  pattern-deduction pairs, and  $v_i$  refers to the number of pattern variables that are within pattern  $\pi_i$ . Note that `match` and `dmatch` compilation are almost identical, except for `MAPCLOSURE`/`MAPDCLOSURE` and no need to protect the assumption base for `dmatch`.

Effectively, the `MAPCLOSURE`/`MAPDCLOSURE` operation both creates and applies a function or method closure with arguments from the mappings of the map-stack, but making sure to have the closure return to the code list *after* the `MATCHEND` operation. It is also important that the closure generated expects the identifiers in the same order that the compiler did — this compiler sorts the identifiers lexicographically.

It should be noted that the `MATCH` operation leaves the discriminant value on top of  $S$ , but places a map-stack — holding the mapping of pattern variables (identifiers) to values — just beneath it on  $S$ . Over

the course of pattern matching, the equivalent of the net-effect property is, upon a successful match, to leave the map-stack on top of  $S$ ; upon failure, the pseudo-value “match-fail” is left on top of  $S$  with the map-stack underneath it.

For a given pattern  $\pi$ , this list specifies the compilation  $[[\pi]]$ :

### Simple Patterns

–

#### MATCHANY

Since the pattern  $_$  matches anything, the MATCHANY operation basically just pops the discriminant off of  $S$ , leaving the map-stack.

( )

#### TRIV.MATCHEQUAL

A unit value is generated, and MATCHEQUAL compares them for equality. As specified in the transitions, MATCHEQUAL will leave the map-stack if the values are equal, or generate match-fail if they are not.

$I$

- If  $I$  is a prop-con, quantifier, or function symbol,  
LDPRIM.I.MATCHEQUAL
- Otherwise,  $I$  is a pattern variable,  
MATCHI.I

In the first case,  $I$  is a particular value so MATCHEQUAL is used. In the second,  $I$  is a pattern variable, and so MATCHI attempts to bind the identifier  $I$  to the discriminant in the map-stack; it may fail if  $I$  is bound already to a different value.

‘ $I$

#### LDPRIM.c.MATCHEQUAL

Load the character designated by  $I$ , and compare against the discriminant for equality.

? $I$

#### LDV.I.MATCHEQUAL

Load the variable designated by  $I$ , and compare against the discriminant for equality.

'*I*

LDPRIM.'*I*.MATCHEQUAL

Load the meta-identifier designated and compare for equality.

(val-of *I*)

[[*I*]].MATCHEQUAL

Effectively, the identifier's value will be loaded via a LD and then compared for equality with the discriminant.

(list-of  $\pi_1$   $\pi_2$ )

MATCHLISTOF.[[ [ $\pi_1$   $\pi_2$ ] ]]

A list-of pattern compiles into a single operation and a slightly more complicated bracket pattern. The MATCHLISTOF enforces that the discriminant is a non-empty list.

(bind *I*  $\pi$ )

MATCHBIND.I.[[ $\pi$ ]].MATCHBINDCOMMIT.I

The MATCHBIND places the identifier and the discriminant into the map-stack. Later, after  $\pi$  is matched successfully and the MATCHBINDCOMMIT is reached, the identifier gets bound, as per the semantics of the bind pattern.

(some-list *I*)

TESTLIST.MATCHSOME.I

(some-term *I*)

DUP.TESTTERM.MATCHSOME.I

(some-var *I*)

DUP.TESTVAR.MATCHSOME.I

(some-*type* *I*) where *type* is not list, term, or var

TESTTYPE.*T*.MATCHSOME.I

The some patterns all use the MATCHSOME operation, which binds *I* to the discriminant if the type testing was successful.

The character *T* here is that of the requested type, as listed with transition 125 for TESTTYPE, Section 3.2.14.

## Bracket Patterns

```
[ $\pi_1 \dots \pi_n$ ]  
  MATCHLIST. $n$ .[[ $\pi_1$ ]].  
    MATCHLISTNEXT.1. [[ $\pi_2$ ]]. ...  
    MATCHLISTNEXT. $n - 1$ . [[ $\pi_n$ ]].  
  MATCHLISTEND
```

The compilation of the empty list pattern is MATCHLIST.0. MATCHLISTEND. Note that the code indexes the discriminant list starting at zero, but the pattern as listed indexes from one.

## Compound Patterns

```
(forall  $\pi_2 \pi_3$ )  where  $\pi_2$  is a list pattern  
  MATCHQPROP.MATCHQ1.forall.MATCHQ2INIT.  
  (MATCHQ2START. [[ $\pi_2$ ]].MATCHQ2END).  
  MATCHQ3. [[ $\pi_3$ ]].MATCHQEND
```

```
(exists  $\pi_2 \pi_3$ )  where  $\pi_2$  is a list pattern  
  MATCHQPROP.MATCHQ1.exists.MATCHQ2INIT.  
  (MATCHQ2START. [[ $\pi_2$ ]].MATCHQ2END).  
  MATCHQ3. [[ $\pi_3$ ]].MATCHQEND
```

```
((some-quant  $I$ )  $\pi_2 \pi_3$ )  where  $\pi_2$  is a list pattern  
  MATCHQPROP.MATCHQ1S.I.MATCHQ2INIT.  
  (MATCHQ2START. [[ $\pi_2$ ]].MATCHQ2END).  
  MATCHQ3. [[ $\pi_3$ ]].MATCHQEND
```

These three forms of the same compound pattern use several special operations. MATCHQPROP matches only propositions, and also collapses them into the form specified in Section 3. The resulting quantifier (if any, the quantifier may be undetermined) is tested with either MATCHQ1 or MATCHQ1S. Most notably, the match fails if the pattern requested a binding  $I$  for the quantifier and the quantifier is undetermined; this is noted in the transitions of MATCHQ1S. MATCHQ1 takes as an argument what quantifier it expects, and succeeds if the discriminant's quantifier matches or is undetermined.

MATCHQ2INIT sets up the code list for matching the list of variables generated by collapsing the proposition against the list pattern  $\pi_2$ . If a match-fail occurs during this code list, however, then



the MATCHQ2START and MATCHQ2END operations will retry on a smaller list of variables, and the compound match will fail only after all variable lists have failed, as per Section 3. If the match is successful when it reaches MATCHQ2END, then it escapes to MATCHQ3, which reassembles the proposition out of the quantifier, the list of unmatched variables, and the base proposition. This is then matched against  $\pi_3$ . MATCHQEND cleans up the dump stack  $D$ .

$(\pi_1 \ \pi_2)$  where  $\pi_2$  is a list pattern  
MATCHASLIST.“nested”.[[  $[\pi_1 \ \pi_2]$  ]]

$(\pi_1 \dots \pi_n)$  for  $n \geq 2$   
MATCHASLIST.“flatten”.[[  $[\pi_1 \dots \pi_n]$  ]]

For these forms of compound patterns, MATCHASLIST verifies that the discriminant is a term or proposition, and then lets the match continue using only bracket patterns.

The transitions of pattern matching have been exactly and painstakingly designed to work properly. When a given pattern fails, the transitions unwind such that the next pattern is attempted. Depending on where in a compound pattern a failure occurs, the machine can determine if it needs to retry or move on to the next pattern. In particular, the matching of nested lists or of lists of compound patterns works with this scheme because of how the transitions have been designed and how the operations are assembled by the compiler.

## 4.2 Primitive Methods

During the code summaries, there are a few macros that are used. This makes them slightly easier to read. The macros “prop1” and “prop2” expand to ASLIST.ELEM.1 and ASLIST.ELEM.2, respectively. They are used to disassemble the proposition on top of the  $S$  stack into their first and second “arguments”; the zeroth argument is the propcon or quantifier. The macros “arg1”, “arg2”, up to “arg $n$ ” expand to LD.0. $n - 1$ . They are used to load the arguments of the closure onto  $S$ , but are slightly easier to read than the LD operation, because the macros start counting from one.

claim P

arg1.KNOWN.SEL.

```
(arg1.JOIN).
(LDERROR.“claimfailed”.JOIN)
```

Check that the argument is in the assumption base, via KNOWN.  
If so, return it as the conclusion; otherwise, generate an error.

**mp** (if P Q) P

```
arg1.TESTIF.SEL.
  (arg1.KNOWN.SEL.
    (arg2.KNOWN.SEL.
      (arg1.prop1.arg2.ALPHAEQ.SEL.
        (DSTART.arg1.prop2.DEND.JOIN)
        (LDERROR.“mpequalityfailed”.JOIN).JOIN)
      (LDERROR.“mp2”.JOIN).JOIN)
    (LDERROR.“mp1”.JOIN).JOIN)
  (LDERROR.“mpnotif”.JOIN)
```

*modus ponens*: Test that (1) the first argument is an **if** proposition, (2) it is in the assumption base, (3) the second argument is in the assumption base, (4) that the first proposition in the first argument is alphabetically equivalent to the second argument, at which point conclude with the second proposition of the first argument.

**absurd** P (not P)

```
arg1.KNOWN.SEL.
  (arg2.KNOWN.SEL.
    (arg2.TESTNOT.SEL.
      (arg1.arg2.prop1.ALPHAEQ.SEL.
        (DSTART.LDFALSE.DEND.JOIN)
        (LDERROR.“absurdfailed”.JOIN).JOIN)
      (LDERROR.“absurdfailed”.JOIN).JOIN)
    (LDERROR.“absurdfailed”.JOIN).JOIN)
  (LDERROR.“absurd1”.JOIN)
```

Generate the conclusion **false** when: (1) the first argument is in the assumption base, (2) the second argument is in the assumption base, (3) the second argument is a negation, and (4) the interior of the negation is alphabetically equivalent to first argument.

**dn** (not (not P))

```

arg1.KNOWN.SEL.
  (arg1.TESTNOT.SEL.
    (arg1.prop1.DUP.TESTNOT.SEL.
      (DSTART.prop1.DEND.JOIN)
      (POP.LDERROR.“dnnotneg” .JOIN).JOIN)
    (LDERROR.“dnnotneg” .JOIN).JOIN)
  (LDERROR.“dn1” .JOIN)

```

Double negation elimination: Test that (1) the argument is in the assumption base, (2) that the argument is a negation, and (3) that the negation’s interior is also a negation, at which point conclude the inner proposition.

**both** P Q

```

arg1.KNOWN.SEL.
  (arg2.KNOWN.SEL.
    (DSTART.arg1.arg2.AND.DEND.JOIN)
    (LDERROR.“both2” .JOIN).JOIN)
  (LDERROR.“both1” .JOIN)

```

Conclude the conjunction after both arguments are tested to be in the assumption base.

**left-and** (and P Q)

```

arg1.KNOWN.SEL.
  (arg1.TESTAND.SEL.
    (DSTART.arg1.prop1.DEND.JOIN)
    (LDERROR.“left-andnotand” .JOIN).JOIN)
  (LDERROR.“left-and1” .JOIN)

```

Return the left-hand side of an **and** proposition, if that proposition is in the assumption base.

**right-and** (and P Q)

```

arg1.KNOWN.SEL.
  (arg1.TESTAND.SEL.
    (DSTART.arg1.prop2.DEND.JOIN)
    (LDERROR.“right-andnotand” .JOIN).JOIN)
  (LDERROR.“right-and1” .JOIN)

```

Return the right-hand side of an **and** proposition, if that proposition is in the assumption base.

**equiv** (if P Q) (if Q P)

```
arg1.KNOWN.SEL.  
  (arg1.TESTIF.SEL.  
    (arg2.KNOWN.SEL.  
      (arg2.TESTIF.SEL.  
        (arg1.prop1.arg2.prop2.ALPHAEQ.SEL.  
          (arg1.prop2.arg2.prop1.ALPHAEQ.SEL.  
            (DSTART.arg1.prop1.arg1.prop2.IFF.  
              DEND.JOIN)  
            (LDERROR.“equivnoteq2”.JOIN).JOIN)  
          (LDERROR.“equivnoteq1”.JOIN).JOIN)  
        (LDERROR.“equivnotif2”.JOIN).JOIN)  
      (LDERROR.“equiv2”.JOIN).JOIN)  
    (LDERROR.“equivnotif1”.JOIN).JOIN)  
  (LDERROR.“equiv1”.JOIN)
```

Generate the biconditional if (1) the first argument is in the assumption base, (2) and it is an **if** proposition, (3) second argument in assumption base, (4) and it too is an **if**, (5) the first part of the first proposition is alphabetically equivalent to the second part of the second, and (6) the second part of the first is equivalent to the first part of the second.

**left-iff** (iff P Q)

```
arg1.KNOWN.SEL.  
  (arg1.TESTIFFSEL.  
    (DSTART.arg1.prop1.arg1.prop2.IF.DEND.JOIN)  
    (LDERROR“left-iffnotiff” JOIN).JOIN)  
  (LDERROR“left-iff1”.JOIN)
```

Generate the left conditional, assuming the argument is in the assumption base and of the appropriate form.

**right-iff** (iff P Q)

```
arg1.KNOWN.SEL.  
  (arg1.TESTIFF.SEL.  
    (DSTART.arg1.prop2.arg1.prop1.IF.DEND.JOIN)  
    (LDERROR.“right-iffnotiff”.JOIN).JOIN)  
  (LDERROR.“right-iff1”.JOIN)
```

Generate the right conditional, assuming the argument is in the assumption base and of the appropriate form.

either P Q

```
arg1.KNOWN.SEL.
  (DSTART.arg1.arg2.OR.DEND.JOIN)
  (arg2.KNOWN.SEL.
    (DSTART.arg1.arg2.OR.DEND.JOIN)
    (LDERROR."eitherfailed".JOIN).JOIN)
```

If the first argument is in the assumption base, generate the disjunction. Otherwise, if the second argument is in the assumption base, generate the disjunction. Otherwise, generate an error.

cd (or P P') (if P Q) (if P' Q)

```
arg1.KNOWN.SEL.
  (arg2.KNOWN.SEL.
    (arg3.KNOWN.SEL.
      (arg1.TESTOR.SEL.
        (arg2.TESTIF.SEL.
          (arg3.TESTIF.SEL.
            (arg1.prop1.arg2.prop1.ALPHAEQ.SEL.
              (arg1.prop2.arg3.prop1.ALPHAEQ.SEL.
                (arg2.prop2.arg3.prop2.ALPHAEQ.SEL.
                  (DSTART.arg2.prop2.DEND.JOIN)
                  (LDERROR."cdfail3".JOIN).JOIN)
                  (LDERROR."cdfail2".JOIN).JOIN)
                  (LDERROR."cdfail1".JOIN).JOIN)
                  (LDERROR."cdnotif2".JOIN).JOIN)
                  (LDERROR."cdnotif1".JOIN).JOIN)
                  (LDERROR."cdnotor".JOIN).JOIN)
                  (LDERROR."cd3".JOIN).JOIN)
                  (LDERROR."cd2".JOIN).JOIN)
                  (LDERROR."cd1".JOIN)
```

Generate the conclusion Q if: (1) all three arguments are in the assumption base; (2) they are an or, if, and if, respectively; (3) the first part of the first argument is alphabetically equivalent with the first part of the second; (4) the second part of the first argument is alphabetically equivalent with the first part of

the third argument; and (5) the second part of the second argument is alphabetically equivalent with the second part of the third argument.

```
uspec (forall ?x P) t
  arg1.KNOWN.SEL.
  (arg1.TESTFORALL.SEL.
   (arg2.TESTTERM.SEL.
    (DSTART.arg2.arg1.prop1.arg1.prop2.REPLACE.
     DEND.JOIN)
    (LDERROR.“uspecnotterm”.JOIN).JOIN)
   (LDERROR.“uspecnotforall”.JOIN).JOIN)
  (LDERROR.“uspec1”.JOIN)
```

Universal specialization: If (1) the first argument is in the assumption base and (2) it is a `forall` proposition, and (3) the second argument is a term, then specialize the `forall` by replacing its variable with the term within the body of the `forall` proposition.

```
egen (exists ?x P) t
  arg1.TESTEXISTS.SEL.
  (arg2.TESTTERM.SEL.
   (arg2.arg1.prop1.arg1.prop2.REPLACE.KNOWN.SEL.
    (DSTART.arg1.DEND.JOIN)
    (LDERROR.“egen1”.JOIN).JOIN)
   (LDERROR.“egennotterm”.JOIN).JOIN)
  (LDERROR.“egennotexists”.JOIN)
```

Existential generalization: If (1) the first argument is an `exists` proposition, and (2) the second argument is a term, and (3) the result of replacing the quantified variable of the `exists` with the term is in the assumption base, then generalize by concluding with the first argument.

## 4.3 Primitive Functions

```
null? list
  arg1.TESTLIST.SEL.
  (TESTNIL.SWAP.POP.JOIN).
  (POP.LDERROR.“null?notlist”.JOIN)
```

Test whether the argument is an empty list.

**add** value list

```
arg2.TESTLIST.SEL.  
  (arg1.PRE.JOIN).  
  (POP.LDERROR.“addnotlist”.JOIN)
```

Add a value to the front of an existing list.

**head** list

```
arg1.TESTLIST.SEL.  
  (TESTNIL.BOOLNOT.SEL.  
   (ELEM.0.JOIN)  
   (POP.LDERROR.“headlistisnil”.JOIN).JOIN)  
  (POP.LDERROR.“headnotlist”.JOIN)
```

Remove the first value of a list.

**tail** list

```
arg1.TESTLIST.SEL.  
  (TESTNIL.SEL.  
   (JOIN).  
   (CDR.JOIN).JOIN)  
  (POP.LDERROR.“tailnotlist”.JOIN)
```

Return the list with its first value removed.

**equal?** value1 value2

```
arg1.arg2.VALEQ
```

Test if two values are equal.

**char?** value

```
arg1.TESTTYPE.R.SWAP.POP
```

**function?** value

```
arg1.TESTTYPE.F.SWAP.POP
```

**list?** value

```

    arg1.TESTLIST.SWAP.POP
method? value
    arg1.TESTTYPE.M.SWAP.POP
prop? value
    arg1.TESTTYPE.P.SWAP.POP
substitution? value
    arg1.TESTTYPE.S.SWAP.POP
symbol? value
    arg1.TESTTYPE.Y.SWAP.POP
term? value
    arg1.TESTTERM
unit? value
    arg1.TESTTYPE.U.SWAP.POP
alpha-equiv prop1 prop2
    arg1.arg2.ALPHAEQ
    Test if two propositions are alphabetically equivalent.
make-term symbol list-of-terms
    arg1.TESTTYPE.Y.SEL.
      (arg2.TESTLIST.SEL.
        (APPLY.JOIN)
        (POP.POP.LDERROR.“make-term2”.JOIN).JOIN).
      (POP.LDERROR.“make-term1”.JOIN)

```

Given a function symbol  $f$  and a list of terms  $[t_1 \dots t_n]$ , generate the term  $(f t_1 \dots t_n)$ .

```
fresh-var
```



## FRESHVAR

Generate a variable that has never existed during the machine's existence, one that is therefore guaranteed not to exist in any term or proposition.

`root` term-or-prop

```
arg1.DUP.TESTVAR.SEL.  
  (JOIN)  
  (ASLIST.ELEM.0.JOIN)
```

Return the “root” of a term or proposition — the symbol, prop-con, or quantifier. A variable is a special case; a variable is its own root.

`children` term-or-prop

```
arg1.DUP.TESTVAR.SEL.  
  (POP.NIL.JOIN)  
  (ASLIST.CDR.JOIN)
```

Return the “children” of a term or proposition — the list of arguments to the symbol, prop-con, or quantifier. A variable is a special case, the list of its children is an empty list.

`extend-sub`  $\theta$   $v$   $t$

```
arg1.arg2.arg3.EXTENDSUB
```

Extend the given substitution  $\theta$  with the binding of variable  $v$  to term  $t$ .

`compose-subs`  $\theta_1$   $\theta_2$

```
arg1.arg2.COMPOSESUB
```

Return the composition substitution  $\overline{\theta_1} \circ \theta_2$ .

### 4.3.1 Math and Internal

There are two operations that were not given in Chapter 3, in that they are not truly relevant to the core of the machine but are relevant for implementing Athena efficiently. The first operation is `MATH`, which is used to implement all of the mathematical, primitive functions that are introduced by directive `define-numeric-operations`. The `MATH` operation takes an argument from the code list specifying what operation to do exactly, reads two arguments from the *S* stack, and pushes the answer onto *S*. Via the `MATH` operations, the primitive functions `plus`, `minus`, `times`, `div`, `exp`, `mod`, `less?`, and `greater?` are available.

Henderson's SECD machine also has most of these primitive math operations, but they are each given their own operation. This could have been done for the Athena virtual machine, but it would have merely padded the number of operations and transitions that the machine can do. And unlike Henderson's machine, this machine can perform a great deal more than just numerical math and higher-order functions.

The other operation not listed is `INTERNAL`. This operation exists for two reasons. First, it provides a means by which the results of certain primitive functions can efficiently computed. `INTERNAL` provides hooks into the implementation of the virtual machine — in this case, the objects that represent Athena values — and allows them to do the computation and return the result, rather than slow down the operation by loading virtual machine code to compute the result. An example of this would be `unify`, which attempts to return a substitution that represents the unification of its two argument terms. Although it would be quite possible to have a native closure to do this (especially with the powerful pattern matching capabilities that Athena provides), it would be slower than having the value objects perform the unification themselves without the virtual machine's intervention.

The second reason for `INTERNAL` is that it provides functionality that is outside of the bounds of the virtual machine itself. For example, the `read-file` primitive function reads a file from the filesystem and returns the result as a string. It would be obscene to provide a VM-specific operation to do this, but it makes sense to ask the internal, VM interpreter to perform the task. `INTERNAL` is also used for special “value morphing” primitive closures, such as `id->string`, `string->var`, `symbol->string` and so on.

The following primitive functions are defined via the `INTERNAL` operation:

`var->string` Convert a variable argument into an Athena string.

**string->var** Convert a list of characters into an Athena variable.

**id->string** Convert a meta-identifier argument into an Athena string.

**string->id** Convert a list of characters into a meta-identifier.

**symbol->string** Convert a function symbol argument into an Athena string.

**string->symbol** Convert a list of characters into a function symbol, assuming the function symbol exists; otherwise, return an error.

**rev** Reverse the order of the elements of list. This is much more efficient internally than using a function closure.

**join** Concatenate the a variable number of lists into one list. This primitive is special in that it can take a variable number of arguments. In particular, the transition for APPLY does not perform the arity check when this primitive function is applied; the primitive has a special, negative arity internally which APPLY knows to ignore.

**supp** Return a list of the support of a substitution. The support consists of those variables that have mappings in the substitution. Internally, the object that represents substitutions has code to generate this list.

**range-vars** Return a list of variables that are in the range of a substitution, those variables being mapped to in the substitution.

**unify** Given terms  $t_1$  and  $t_2$ , return a substitution  $\theta$  such that  $\bar{\theta}(t_1) = \bar{\theta}(t_2)$ , or **false** if no such substitution exists. Unification is performed more efficiently internally than would be possible as native virtual machine code.

**occurs** Given a variable  $v$  and a proposition  $P$ , returns **true** or **false** depending on if  $v$  occurs in  $P$ . The object representing propositions has code explicitly for this test.

**replace-var** Given a variable  $v$ , term  $t$ , and proposition  $P$ , replace free occurrences of  $v$  in  $P$  with  $t$ .

**rename** Given a proposition, replace all quantified variables with fresh variables. The old and new propositions are alphabetically equivalent.

**read-file** Given a list of characters representing a filename, return a string containing the file's contents; return an error if there was an error in reading the file.

**print** Print to the user the argument string.

**write** Print to the user an Athena value.

This ends the listing of the primitive functions of Athena.

The next and final chapter will provide details into the design and implementation of the machine, provide a few examples of the machine working, and discuss possible optimizations.

## Chapter 5

# Design and Implementation

### 5.1 Design

In the design of the Athena virtual machine, there are many different issues to deal with. The VM uses Henderson's SECD machine as a basis [Hen80], but with many extensions. It seems clear that a structure is needed to hold the assumption base and that a stack or list would be sufficient. Cardelli's Functional Abstract Machine (Fam) is itself an extension of SECD, but optimized for fast function application [Car83]. The largest offering from the Fam is a memory structure  $M$  for mapping addresses, "the formal characterization of ref cells" [Car83, pp. 4], to values. Although the Athena VM has no notion of addresses for values in general, it adopts the notion of addresses for cells and provides a stack  $M$  to act as memory for the address space to reside.

#### 5.1.1 Errors and Error Handling

The Fam has an explicit notion of when operations can fail. In contrast, Henderson's machine merely notes that results of certain operations are undefined if they are used improperly. It is clear that Athena needs to have errors be available, and that they sometimes need to act like values in that they are able to be generated and pushed onto stacks. An alternative would be to have a specialized structure to note when an error occurred and to direct the machine to handle it; the VM as implemented has this built into the transitions without additional

structures. The use of try blocks and the possibility of nested try blocks suggested that errors during execution act as values. By careful use of the stacks themselves to keep track of errors and their resolution, neither the machine nor the compiler has to special-case the nesting of try blocks.

Errors that occur as the result of Fam operations also contain a text string, usually the name of the operation that failed. This is also used by the Athena VM, except that most VM errors will attempt to clarify what went wrong. In this implementation, an operation will fail with a verbose message. If `ASSERT` fails because it was not given a proposition, it will include the bad value it was given. Likewise, the primitive method for *modus ponens*, `mp`, has four different error messages depending on how its arguments may have been inappropriate, and similarly for the rest of the primitive methods.

### 5.1.2 Closure Application

In Henderson's SECD machine, the placement of closures and their argument lists is reversed when compared to that of the Athena virtual machine. The semantics of Athena requires, in the evaluation of an expression of the form  $(E F^*)$ , that  $E$  be computed first. Therefore, Athena's `APPLY` operation expects its arguments in the reverse order, and this change has to be propagated to the recursive apply operation `RAPPLY` as well. Additionally, arity checking has been added to these operations for Athena; arity checking was not required for Lispkit. Since Athena also provides method closures for the handling of deductive processes, it has additional, analogous operations for their creation and application.

### 5.1.3 Sufficiently Powerful Operations

An effort was made in the design to provide just enough operations so that all the core Athena functionality is actually available as virtual machine code, and not just by having several "super operations" that do all of the work. It is clear that some of what the machine does during the course of executing Athena code is making full use of the stacks available and the provided operations. One example is the compilation of `with-witness`. Over the course of operation, values are pushed onto the  $S$  stack up to four levels deep for possible use later, and these are `SWAPped` in place "just in time" for their use by other operations.

Another good example is the compilation and execution of patterns, which makes extensive use of the dump stack  $D$  to perform the neces-

sary work. Each piece is compiled into matching primitives operations, which, when properly compiled, can handle all sorts of seemingly special cases: lists that contain other lists, lists of compound matches, compound matches nested within other compound matches, and so on. If a match-fail occurs, the flow of the code is directed to an operation than can deal with it, either by restarting the matching of the variable list inside a compound pattern or proceeding to the next `MATCHNEXT` or `MATCHEND`. As it proceeds, it passes operations whose transitions clean up the dump stack  $D$  into the form expected by the next pattern.

#### 5.1.4 The Compiler

The Jam compiler makes no attempt at optimization. It does not look-ahead, it does not rearrange phrases, and it does not analyze when and if it could remove `DSTART/DRESTORE` pairs from the code list. On purpose, it is merely a straightforward translation from Athena phrases to VM code. Bugs in any attempt to optimize the compiler could introduce potentially difficult-to-find bugs within the code. Discovering, isolating, and fixing these bugs would be irrelevant to the correctness of the operations and VM implementation. To avoid wasting time in debugging the compiler — time that could be better spent elsewhere — a conscious decision was made for the compiler to be simple. A benevolent side-effect of this is that it produces code that is not entirely difficult to inspect and follow visually during debugging.

#### 5.1.5 The Backup Stack $B$

When comparing the Lispkit to Athena, it makes sense to add an  $M$  stack for dealing with cells and an  $A$  stack for holding the assumption base. Why is a  $B$  stack necessary? There are several answers.

The compiler (as currently written) does not know or notice when a `DRESTORE` or `DEND` will occur. To properly handle these operations, the machine needs to know where the backed-up assumption base resides, so that it can be restored or extended. The machine could instead define these operations to dig through the dump stack  $D$  (similar to what the machine does when dealing with an error) to find the correct assumption base to restore; there would be some slight slowdown for this search, as well as a slight increase in overhead in performing a `DSTART` (such as by leaving a labeling next to the backup assumption base, similar to `TRY`). The details for these are slightly messy, but not much more so than the transitions involved with a try block or pattern matching.

However, the most compelling reason is that having a separate stack is both easy to specify, easy to implement, and much easier to debug than the above constructions. Working around having a *B* stack is also discussed as an optimization in Section 5.4.1. Aesthetically, there is also something pleasing in having “the *S* and *D* stacks for functions” and the analogous “*A* and *B* stacks for methods.”

## 5.2 Implementation

The virtual machine, compiler, and read-eval-print loop as designated in this thesis have been implemented in Java, as the Java Athena Machine (Jam). Jam runs in Sun’s Java™ 2 Runtime Environment (build 1.3.1.01) with the Java HotSpot™ Client VM and was built with the 1.3 java compiler. The bulk of the code for Jam was developed across two different Debian GNU/Linux x86 machines. For parsing Athena code, Jam uses a combination of a lexical analyzer generator for java called JLex (version 1.2.5) and the Constructor of Useful Parsers (CUP for short, version 0.10). The combination of JLex and CUP were extremely useful in incrementally adding all of the features of Athena.

### 5.2.1 Class Hierarchy

Jam has a hierarchy of classes, representing everything from the stacks and operations, to Athena values, to structures used internally for sort checking. The classes fall into these java packages.

**Jam.parser** contains all the JLex and CUP details and their associated debugging machinery. As a whole, it produces a class, `Jam.parser.parser`, that accepts syntactically legal Athena code and produces an abstract syntax tree.

**Jam.ast** contains classes for all of Athena’s phrases. In particular, it contains the interface `Jam.ast.Phrase`, and the two sub-interfaces `Jam.ast.Expression` and `Jam.ast.Deduction`. All of the classes in `Jam.ast` implement one of the two sub-interfaces, except the `Namespace` class. The `Phrase` interface requires the existence of a `compile` method, which accepts a `Namespace` to compile with respect to and a `java.util.ArrayList`, to which the compiled code should be prepended.

**Jam.ast.pat** contains the abstract syntax tree for patterns; there were sufficiently many of them to warrant their own package. All



classes in `Jam.ast.pat` implement the `Jam.ast.pat.Pattern` interface. There are additional interfaces to classify Patterns as being quantifier patterns or list patterns. Patterns, although not Phrases, do have a `compile` method just like Phrases.

**Jam.ast.dir** contains the abstract syntax tree for Directives. The `Directive` class has several subclasses — mainly those directives that required enough code to warrant it. Instead of a `compile` method, Directives have a `handle` method which accepts a `Jam.vm.VM` object.

**Jam.value** contains the interfaces for all Athena values. `Jam.value.Value` is implemented by all Athena values — except for lists, which are discussed below. The `Value` interface requires two methods: `valEquals` which takes another `Object` as argument and returns a boolean (or throws an exception if closures of the same type are compared), and `athString` which returns a `String` representing a “nice” version of the value. Also in `Jam.value` is the `Applicable` interface; this designates values that can be handled by the `APPLY` operation. It also includes classes for those Athena values that do not belong in `Jam.term` or `Jam.prop`.

**Jam.term** contains the interface `Term` and classes for function symbols, variables, function symbol applications, and substitutions. There are separate distinctions for constructors, nullary function symbols, and nullary constructors — in particular nullary symbols implement `Term` whereas normal, non-nullary symbols do not. This makes the hierarchy a little complicated, but makes coding the VM easier in many ways. `Jam.term` also holds the classes `Domain` and `Structure`, and code for checking the legality of terms and sort checking.

`Jam.term` has three instances of the Factory design pattern: `SortFactory`, `FSymbolFactory`, and `VarFactory`. These keep track of what sorts, symbols, and variables exist, and maintain that there is exactly one instance of each. This is done for efficiency: there is only one instance of any given sort, symbol, or variable, and it is shared among all terms or propositions that use them. Also, the existence of a `Factory` for variables makes it easy to enforce that fresh variables are indeed fresh.

**Jam.prop** contains the abstract class `Prop` for propositions. It also defines the propositional constructors and the quantifiers, as well as all the classes for the logical and quantified propositions. The

class `Atom` also exists, basically as a wrapper for `Term`. The `Prop` class uses code from `Jam.term` to perform sort checking and some legality tests.

**Jam.vm** contains the all-important VM class, as well as structures used to represent some stacks efficiently. VM operations are specified in the `Op` class. Each `Op` is statically generated such that all uses of a given operation are actually references to the same object. This package also has the pseudocode class, which contains java methods for generating the code of Athena’s primitive functions and methods.

**Jam** contains the `repl` class, which uses and controls the other packages. Thus the read-eval-print loop for Jam is started via “java `Jam.repl`”.

### 5.2.2 Lists

Lists in the current Jam implementation are instances of `java.util.List`. Usually they are instances of `java.util.ArrayList` unless the value has been acted upon by `PRE`, in which case it is a `java.util.LinkedList` — prepending to an `ArrayList` is an  $O(n)$  operation whereas it is  $O(1)$  for a `LinkedList`. Thus, within the VM class, those operations that can accept either Athena values or Athena lists have to determine if their arguments are of the instance `Jam.value.Value` or of the instance `java.util.List`. This is slightly complicated by the fact the `Jam.value.Closure` class is a subclass of `ArrayList`. In hindsight, it may be simpler to provide a wrapper class for `java.util.List` that implements `Jam.value.Value` and use it whenever lists are generated.

### 5.2.3 Blurring of Types

There is also the issue of the blurring of types in Athena. For example, the symbol `true` is a constant function symbol, a term, and a proposition. A given variable `?v` is a term, but it could also be a proposition. As implemented, `true` is a `Jam.term.NullaryConstructor` of `Structure Boolean`. `NullaryConstructors` are instances of `Term`. However, there is an explicit conversion process for making a `Term` (of sort `Boolean`) into a `Prop`. Thus, those operations that explicitly require propositions have to attempt to convert their arguments into `Prop` objects, and those operations that expect terms have to make an explicit check to see if they were given atomic propositions, and if so grab the inner `Term` object from the `Atom`.

One solution might be to make `Atom` an instance of `Term` — trivially so, in that the methods fall through to the internal `Term`. However, then `Variables` would have to be special-cased, as they can be `Atoms` if they are coerced into the `Boolean` sort, but they are not `Atoms` by default. These issues about the class hierarchy for terms and propositions are complex, and possible future changes should be studied carefully before implementation is attempted.

#### 5.2.4 Numbers and Meta-Identifiers

Both numbers and meta-identifiers are nullary function symbols, and therefore of the instance `Jam.term.NullaryFSymbol`. However, because of the factory for function symbols, their introduction is slightly tricky — if the given number or meta-id did not previously exist, it needs to be dynamically created and added to the `FSymbolFactory`.

Numbers, after a successful `define-numeric-operations`, are handled as identifiers that are special-cased by the `Namespace` during compilation. If the number is within the defined range, then the code produced references a function symbol. If the function symbol doesn't exist for that numeral, it is dynamically created with the proper signature and added to the `FSymbolFactory`. Meta-identifiers are also created dynamically, but they are slightly more simple since their sort is known in advance by the `Namespace`.

#### 5.2.5 Recursive Closures and Circular Environments

One particular bit of cleverness in Henderson's SECD machine is the creation of the environment for `letrec`. In [Hen80], the `DUM` operation adds to the `E` stack the value  $\Omega$ , which is called *pending*. Later, the recursive apply operation `RAP` generates the proper environment for the closure by using the lisp pseudo-function `rplaca`. The effect of `rplaca(x,y)` is to replace the *car* of `x` with `y`; the value returned is the resulting `x`. Thus for `letrec` in Lispkit, first a `DUM` will create a dummy environment, then the list of arguments to the recursive closure will be created with respect to this environment, and finally `RAP` will replace the  $\Omega$  in the environment with the list of values.

However, the resulting list of values will be circular, in that any closures defined there will contain an environment which contains the closures themselves as values. Jam uses the same trick to implement `letrec` and `dletrec`, although the operations are now `DUMMY` and `RAPPLY/RDAPPLY`, the special value  $\Omega$  is replaced with the string

“pending”, and there is no underlying lisp medium. The argument list that is given to `RAPPLY/RDAPPLY` is an `ArrayList`, and from it a circular `ArrayList` is created in which any environment in any closure of the argument list has “pending” replaced with the circular `ArrayList`.

### 5.3 An Example Deduction

Below is a method that is similar to the primitive method for double negation elimination, `dn`, but instead acts on an arbitrary depth of negated propositions.

```
(define (dn* P)
  (dmatch P
    ((not (not _)) (!dn* (!dn P)))
    (_ (!claim P))))

(define long-negation
  (not (not (not (not (not (not ?x)))))))

(assert long-negation)

(!dn* long-negation)
```

It should be obvious that the result of the deduction `(!dn* long-negation)` will be the proposition `?x`. What is interesting is how the semantics of Athena go about concluding this answer, and what the assumption base looks like during the deduction.

When the deduction starts, there is only one relevant proposition in the assumption base: `long-negation`. (There are other propositions in the assumption base, e.g. `true` and any others that might have been concluded or asserted earlier, but they do not matter for this deduction.) After being matched by the `(not (not _))` pattern, the primitive method `dn` is applied to `long-negation` and, since `long-negation` is in the assumption base, `dn` returns the proposition `(not (not (not (not ?x))))`. Since this value is the conclusion of a deduction and part of a method application (`dn*` is being applied), it is added to the assumption base for the scope of the recursive call to `dn*`.

For the recursion, the relevant propositions in the assumption base are `long-negation` and `(not (not (not (not ?x))))`. Like the previous call to `dn*`, there are at least two `not`s in `P`, and Athena enters another application of `dn*`. Here, `dn` returns the proposition `(not (not ?x))` since `(not (not (not (not ?x))))` is in the assumption base.

Again, the resulting proposition is added to the assumption base for the recursive call to `dn*`.

Inside this application of `dn*`, the assumption base now contains `long-negation`, `(not (not (not (not ?x))))`, and `(not (not ?x))`. Again, there is a recursive call to `dn*`, this time with the argument `?x` with respect to an assumption base that has `?x` as a proposition. Finally, `dn*` will produce the conclusion `?x` as a result of `claim`, since `?x` is in the assumption base. The applications of `dn*` proceed to unwind. The proposition `?x` is the result of each `dn*` method application including the outermost call; as the value propagates to the outermost call, the assumption base reverts back to containing merely `long-negation`. The result of evaluating a `apply-method` deduction at the top-level is the adding of its conclusion to the assumption base. Finally, the value returned is `?x`, and this proposition is now in the assumption base for future deductions to use.

## 5.4 Future Work

### 5.4.1 Stack Optimizations

In the current implementation, the assumption base stack *A* is just a list of propositions. They are ordered such that more recent propositions are searched first, to properly harness any spatial or temporal locality. When a proposition is added to *A*, there are no checks to determine whether the proposition (or one alphabetically equivalent to it) is already there. This waste is especially compounded whenever *A* is copied to the backup stack *B*.

A particularly nice optimization for the virtual machine implementation would be the replacement of the *A* and *B* stacks with a single structure. One possible replacement is a hashtable that accepted canonically renamed propositions for insertion, and which allowed arbitrary “rollback,” in that propositions could be removed in the order that they were last added. Currently, when a proposition is searched for in the assumption base, e.g. via the `KNOWN` operation, it is compared via an alphabetical equivalence test with every proposition in *A* until a match is found. With canonical renaming, the proposition would first be renamed with canonical variables such that the alphabetically equivalence test would degenerate to a literal equality test. Combined with hashing, literal equality against propositions in the assumption base could be done quickly.

In order to properly emulate the *B* stack, propositions would need to be able to be removed from the assumption base. There are issues

associated with this as well. For example, a proposition could be in the assumption base from multiple, different conclusions, and when the latest of those conclusions is rolled back, the proposition needs to remain in the assumption base.

### 5.4.2 Transition Optimizations

There were no real attempts made to minimize the number of operations required for the Athena virtual machine. In fact, every effort was made to reinforce the ease of debugging, sometimes at the cost of adding more operations. For example, `RAPPLY` and `RDAPPLY` are the same, except that they expect different closure types. However, the compiler enforces that the closure of the correct type is given to these operations, as they are both part of the construction of `letrec` or `dletrec`. They could be replaced by a more generic version of the same operation. Similarly, `MAPCLOSURE` and `MAPDCLOSURE` in pattern matching.

The operations for generating propositions, `AND`, `EXISTS`, and so on, could be condensed into a single operation that takes an argument specifying what proposition to create. This could also be done for the `TESTAND`, `TESTEXISTS`, etc. operations; the `TESTTYPE` operation could be extended to include the other `TESTING` operations. Finally, some of the pattern matching operations are always in a known order, e.g. a `MATCHQ2INIT` is always followed by a `MATCHQ2START`, and a `MATCHQ2END` is always followed by a `MATCHQ3`. These operations could be condensed, at the cost of legibility of the transitions that would result. The `MATCHANY` operation, in fact, is basically a `POP`. Again, for ease of implementation and ease of debugging, no attempt was made to condense these operations.

### 5.4.3 Pattern Matching Optimizations

As mentioned earlier the compiler is not optimized, and it follows that that compilation of patterns is also not optimized. Some very interesting optimizations for pattern matching in ML are given in [BM85]; they are potentially useful because Athena's pattern matching is similar to ML. However, ML has linear pattern matching, in which a variable can not occur more than once in a pattern. Athena, however, uses nonlinear pattern matching, in which repeated variables add the constraint that their values are equal. Also, matching in ML has the freedom to examine components of the discriminant in any order, whereas Athena semantics requires the examination to be in order. Thus [BM85], in its

current form, would only be potentially useful for a subset of Athena patterns.

Functional pattern matching compilation is also discussed in [Wad87] for the language Miranda. There may be issues in refitting the optimizations into Athena — in particular with Athena’s nonlinear matching and Miranda’s “partial,” curried functions — but it could prove useful for those wanting to optimize pattern matching for the Athena VM.

#### 5.4.4 Full Athena Implementation

The difficulty in extending the VM to handle polymorphic Athena is mainly that sort checking becomes significantly more complicated. The objects representing terms and propositions would need additional state, such as flags to note if the object has been previously sort checked and legal, and perhaps a cache of what the final sort was and the sorts of object’s variables. The VM would need explicit operations for sort checking — perhaps even a stack solely for sort checking, although the dump stack *D* might be used in practice.

The addition of the `split` pattern would probably involve adding several operations and looping constructs similar to those required for matching compound patterns. It is unclear how much additional work would be required for `by-induction-on` and automatically generated methods for structures. Both would likely require the machine to have knowledge about what constitutes a structure — information which the machine currently does not have access to.

#### 5.4.5 Garbage Collection

The current implementation has no implicit garbage collection except for that inherent to java. Most notably, variables are never garbage collected by java as there are always references to them in the `VarFactory`, and cells are never cleaned up as they are present as elements in an `ArrayList` that represents the *M* stack. However, since the entire state of the machine is encapsulated into the seven stacks (discounting the state associated with sorts and symbols, which would not be garbage collected anyway), there could be a separate thread running (perhaps while waiting at the repl loop) to remove variables and cells that are no longer reachable. One potential difficulty would be the garbage collection of objects that are part of circular closures generated by `letrec` and `dletrec`.

### 5.4.6 Alternative VM Implementation

There should be a substantial performance increase to be had by implementing the virtual machine in C/C++, or any language that compiles to the hardware level. For Cardelli's Fam, "the instructions of the machine are not supposed to be interpreted, but assembled into machine code and then executed" [Car83, pp. 1]. The architecture in question was VAX, however. It is yet to be seen how much work might be involved in compiling Athena to a modern architecture using this VM as intermediate code, but one could expect significant increases in speed.



## Chapter 6

# Conclusion

In designing and implementing a virtual machine, I found that emphasizing correctness over efficiency was an extremely important coding philosophy. Debugging the system was relatively straightforward, and I avoided dealing with optimized structures that could have introduced subtle bugs. It makes sense, first and foremost, to make sure that the machine works correctly before attempting to optimize it. All in all, I think the machine is rather elegant in its relative simplicity when compared to the power provided by the Athena programming language. These simple transitions, when properly joined, can perform a number of useful and nontrivial deductions.

There have not been extensive tests, but initial evidence shows that the Jam java implementation is slower than the Arkoudas' canonical Athena interpreter, which is implemented in SML-NJ. Additional work in optimizing this architecture should eventually yield a faster implementation.

Overall, I feel I learned a great deal about Lisp and functional programming. In particular, I had to implement the core functionality required to make a functional language interpreter out of a programming language that does not have Lisp primitives. I also acquired a great deal of respect for the Athena programming language as I was working through implementing its semantics. I believe that this work has reawakened a personal interest in non-imperative programming languages.

In conclusion, the virtual machine and compiler specified in this thesis performs not only the computational and functional aspects of the Athena programming language, but also embodies and handles, at a very low level, the core deductions that are required of a type- $\omega$

denotational proof language.

The source for Jam is available at  
<http://web.mit.edu/tarvizo/Public/jam/>.

# Bibliography

- [Ark99a] Konstantine Arkoudas. Athena as a programming language. Unpublished manuscript, 1999.
- [Ark99b] Konstantine Arkoudas. An Athena tutorial. Unpublished manuscript, 1999.
- [Ark01] Konstantine Arkoudas. Type- $\omega$  DPLs. MIT AI Memo 2001-27, 2001.
- [BM85] Marianne Baudinet and David MacQueen. Tree Pattern Matching for ML (extended abstract), December 1985.
- [Car83] Luca Cardelli. The Functional Abstract Machine. Technical report, AT&T Bell Laboratories, 1983. Technical Report Number TR-107.
- [Hen80] Peter Henderson. *Functional Programming: Application and Implementation*. Prentice/Hall International, Englewood Cliffs, NJ, 1980.
- [Lan64] Peter J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6: 308–320, 1964.
- [Wad87] Philip Wadler. Efficient Compilation of Pattern-Matching. In Simon L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 5. Prentice Hall International (UK) Ltd, Hertfordshire, United Kingdom, 1987.
- [War83] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Part, CA, 1983.