A Athena Reference

THIS APPENDIX provides a compact description of the syntax and operational semantics of Athena's core language constructs.

A.1 Syntax

As explained in Chapter 2, there are two main syntactic categories in Athena: *expressions* (*E*) and *deductions* (*D*). A *phrase F* is either an expression or a deduction:

$$F ::= E \mid D.$$

The syntax of expressions is specified in Figure A.1, while that of deductions is shown in Figure A.2. Figure A.4 depicts the syntax of patterns. In the rest of this section we describe the syntax of character constants (*C*), string constants (*T*), and the sorts (*S*) used for annotations. ASCII characters are generated by the following grammar:

C	::=	‘*printable*	(e.g., ‘`A`)
	|	‘`\`*code*	(e.g., ‘`\73`)
	|	‘`\^`*control-character*	(e.g., ‘`\^C`)
	|	‘`\`*escape-character*	(e.g., ‘`\n`)
escape-character	::=	`" | \ | a | b | n | r | f | t | v`	
control-character	::=	`A` | ··· | `Z | @ | [ | ] | / | ^ | _ |` ‘`\blank`	

where *printable* is any printable ASCII character other than the backslash `\` (i.e., any character with ASCII code from 32 through 91 or 93 through 127, inclusive); and *code* is any sequence of one, two, or three digits: `9`, `03`, `124`, and so on. Since any ASCII character with code *c* can be expressed as ‘`\`*c*, the other three ways of representing characters are strictly speaking redundant, but convenient nevertheless. Escape characters have standard meanings; for example, `\n` is the newline character, `\t` is the tab, and so forth. An Athena string *T* is simply a list of characters. String constants can be directly input between quotes:

$$T \quad ::= \quad \texttt{"}C^*\texttt{"} \qquad \text{(e.g., } \texttt{"\textbackslash nHello world!"}\text{)}$$

where *C* here is just as described by the foregoing grammar for characters, except that the opening quote mark ‘ is omitted.[1] Thus, control characters and escape characters can be directly embedded inside string constants. As a notational convention, for any nonterminal *X*, we write X^* for a sequence of zero or more strings generated by *X*, and X^+ for a sequence of one or more such strings.

Finally, the following grammar gives the syntax of sorts:

1 If included, it will count as a separate character.

E	::=	I	Identifiers
	|	C	Characters
	|	T	String constants
	|	()	Unit
	|	$?I{:}S$	Term variables (annotated)
	|	$?I$	Term variables (unannotated)
	|	$'I$	Meta-identifiers
	|	**check** $\{F_1$ => E_1 | $\cdots$ | F_n => $E_n\}$	Conditional expressions
	|	**lambda** (I^*) E	Procedures
	|	$(E\ F^*)$	Applications
	|	$[F^*]$	Lists
	|	**method** (I^*) D	Methods
	|	**let** $\{\pi_1$:= F_1; $\cdots$; π_n := $F_n\}$ E	Let expressions
	|	**letrec** $\{I_1$:= E_1; $\cdots$; I_n := $E_n\}$ E	Recursive let expressions
	|	**match** F $\{\pi_1$ => E_1 | $\cdots$ | π_n => $E_n\}$	Match expressions
	|	**try** $\{E_1$ | $\cdots$ | $E_n\}$	Try expressions
	|	**cell** F	Cells
	|	**set!** E F	Assignments
	|	**ref** E	References
	|	**while** F E	Loops
	|	**make-vector** E F	Vector creation
	|	**vector-sub** E_1 E_2	Vector access
	|	**vector-set!** E_1 E_2 F	Vector assignment
	|	**(seq** F^***)**	Sequences
	|	**(&&** F^***)**	Short-circuit boolean “and”
	|	**(||** F^***)**	Short-circuit boolean “or”

Figure A.1
Syntax of Athena expressions

S	::=	$'I$	(Sort variables, e.g., `'S5`)
	|	I	(Constant sort constructors, e.g., `Boolean` or `Int`)
	|	$(I\ S_1 \cdots S_n)$	(Compound sorts, e.g., `(List Int)`)

Note that the patterns that may appear in a clause of a **by-induction** or **datatype-cases** proof are of a restricted form; their syntax is described by the following simple grammar:

$$\pi ::= I \mid I{:}S \mid (I\ \pi^+) \mid (I\ \textbf{as}\ \pi)$$

where S ranges over sorts.

D	::=	**(apply-method** $E\ F_1 \cdots F_n$**)**	Method calls
	|	$(!E\ F_1 \cdots F_n)$	Method calls (usual syntax)
	|	**conclude** $F\ D$	Conclusion-annotated deductions
	|	**assume** $F\ D$	Hypothetical deductions
	|	**assume** I := $F\ D$	Named hypothetical deductions
	|	**suppose-absurd** $F\ D$	Proofs by contradiction
	|	**generalize-over** $E\ D$	Universal generalizations
	|	**pick-any** $I\ D$	Universal generalizations
	|	**pick-any** I:$S\ D$	Universal generalizations
	|	**with-witness** $E\ F\ D$	Existential instantiations
	|	**pick-witness** I **for** $F\ D$	Existential instantiations
	|	**pick-witnesses** $I_1\ I_2\ \cdots\ I_n$ **for** D	Existential instantiations
	|	**by-induction** F {π_1 => D_1 | $\cdots$ | π_n => D_n}	Structural induction
	|	**datatype-cases** F {π_1 => D_1 | $\cdots$ | π_n => D_n}	Structural case analysis
	|	**check** {F_1 => D_1 | $\cdots$ | F_n => D_n}	**check** deductions
	|	**match** F {π_1 => D_1 | $\cdots$ | π_n => D_n}	**match** deductions
	|	**let** {π_1 := F_1; $\cdots$; π_n := F_n} D	**let** deductions
	|	**letrec** {I_1 := E_1; $\cdots$; I_n := E_n} D	**letrec** deductions
	|	**try** {D_1 | $\cdots$ | D_n}	**try** deductions

Figure A.2
Syntax of Athena deductions

A.2 Values

Figure A.3 shows the types of values that Athena phrases denote. Some brief remarks on each type follow.

1. The unit value, denoted by the expression `()`, is a single special object, distinct from all other values. It is used primarily as the result of expressions with side effects.
2. Function symbols were discussed in Section 2.2; they are datatype or structure constructors, or else symbols introduced via **`declare`**. This means that symbols such as `false` and `N.+` are actual values, possible results of computations.
3. Terms and sentences are as explained in Chapter 2.
4. The sentential connectives are the five operators used to build compound sentences: `not`, `and`, `or`, `if`, and `iff`. In this group of values we also have the quantifiers `forall` and `exists`. These are values, so they too can be the results of computations.
5. A list of values is just that: a finite list $[V_1, \ldots, V_n]$, $n \geq 0$, possibly empty.[2]

2 We use commas to separate the elements of such lists in order to emphasize that these are not the same as Athena lists. The latter are syntactic objects (expressions), whereas value lists are abstract objects: finite sequences of values.

6. A *function value* is the mathematical object denoted by an Athena procedure. Specifically, a function value is a computable ternary function that takes as arguments (1) a list of values $[V_1, \ldots, V_m]$, $m \geq 0$; (2) an assumption base β; and (3) a store σ; and returns an ordered pair (V, σ') consisting of a value V and a store σ', or else halts in error or diverges. Function values are thus higher-order: some of the V_i inputs might themselves be function values, and the output returned by applying a function value may itself be a function value.
7. A *method value* is the deductive analogue of a function value; it is the mathematical object denoted by an Athena method. Specifically, a method value is a computable ternary function that takes as arguments (1) a list of values $[V_1, \ldots, V_m]$, $m \geq 0$; (2) an assumption base β; and (3) a store σ; and returns an ordered pair (p, σ') consisting of a sentence p and a store σ' (or else generates an error or diverges). Note that a method value may take other method values as inputs, but it may not produce them as results. (Function values, by contrast, may produce method values as results.)
8. Characters are individual ASCII symbols.
9. A substitution is a computable function from term variables to terms that is the identity almost everywhere; substitutions were discussed in Section 2.14.8.
10. Tables and maps are type-tagged finite functions from values to values. So while a table and a map might abstractly represent the same underlying function from values to values, they are guaranteed to be distinct entities.
11. Finally we have cells and vectors, which act as storage containers that can hold arbitrary values (possibly other cells and/or vectors). For mathematical purposes, a cell may be associated with a unique natural number l, and we may then think of the computer's memory as an infinite list: cell 0, cell 1, cell 2, cell 3, and so on. All of these cells are initially unassigned. Vector values are modeled as lists of cells.

These families of values are not quite pairwise disjoint; there is some overlap between (a) terms and function symbols, and (b) terms and sentences. First, a constant symbol such as `zero` or `peter` counts both as a symbol and as a term. And second, a term of sort `Boolean` counts both as a term and as a sentence. In each of these two cases, a value of one type may be coerced into the other type, as required by the context. It is fine, for instance, to pass a term t of sort `Boolean` to a procedure that expects a sentence; t will just be treated as a sentence. Athena performs such conversions automatically.

Finally, a word about value equality. Any two values of the same type may be compared for equality (e.g., with the primitive binary procedure `equal?`). Equality for sentences is alpha-equivalence. For terms it is syntactic identity modulo sort renaming. A cell is only identical to itself. If you think about cells as locations indexed by natural numbers, that makes sense: locations i and j are the same iff $i = j$. Accordingly, different cells may have identical contents. Similar remarks apply to vectors. Two substitutions are equal iff they

The unit value	Function symbols	Sentential connectives and quantifiers
Terms	Sentences	Lists of values
Substitutions	Function values	Method values
Cells	Vectors	Characters
Tables	Maps	

Figure A.3
Types of Athena values

have identical supports, and assign the same term to each variable in their supports. Equality is not decidable for function and method values, and an error will occur if one attempts to apply `equal?` to such values. Equality for characters is obvious: two characters are the same iff they have the same code. Two lists of values $[V_1, \ldots, V_n]$ and $[V'_1, \ldots, V'_m]$ are identical iff $n = m$ and V_i is equal to V'_i for $i = 1, \ldots, n$. Two tables (or two maps) are identical iff they have the exact same domain and range; that is, they map the same keys to the same values. The unit value is only identical to itself. Finally, identity on function symbols, sentential connectives, and quantifiers is clear: A symbol or a sentential connective or quantifier is identical only to itself. Values of different types are considered distinct by default, unless one of the values can be converted to the type of the other and the two can then be determined to be identical on the basis of the above conventions.

A.3 Operational semantics

In this section we specify in detail the result of evaluating any phrase F in a given environment ρ, assumption base β, store σ, and symbol set γ. We clarify these parameters below:

1. An *environment* ρ is a computable function that maps any given identifier I either to a value V or to a special *unbound* token.
2. An *assumption base* β is a finite set of sentences.
3. A *store* σ is a computable function that maps any natural number (representing a memory location) to a value (the location's contents) or to a special *unassigned* token. Infinitely many numbers must be unassigned in any given store.
4. A *symbol set* γ is a collection of function symbols and their respective signatures,[3] along with a collection of sort constructors and their arities; γ also includes information

3 See page 24 for a discussion of signatures.

on whether a given sort constructor is a datatype or structure, and if so, which function symbols are its constructors.

The result of evaluating a phrase F with respect to given ρ, β, σ, and γ, is one of three things:

1. a pair (V, σ') consisting of a value V and a store σ', where V is the output of the evaluation and σ' reflects any side effects accumulated during the evaluation; or
2. a pair consisting of an error message and a store σ', indicating the occurrence of an error during the computation (a store σ' is still necessary to reflect side effects accrued prior to the error); or
3. nontermination.

The specification of the evaluation process below is given in English, but without ambiguity. It can serve as the basis for implementing a core Athena interpreter.

Since the evaluation of most expressions leaves the store unaffected, if we do not explicitly specify what the new store is then it should be assumed that it is the same as that in which the phrase was evaluated, so that $\sigma' = \sigma$. Also, unless we explicitly say otherwise, we will generally assume that the evaluation of a phrase F is immediately halted if the evaluation of a subphrase of F produces an error message and a store σ'; in such cases the result of evaluating F becomes that same error message and σ'.

The evaluation algorithm proceeds by a case analysis of the syntactic structure of the given phrase. We begin with expressions, but first a piece of notation. Consider any function f from a set A to a set B. When $a_1, \ldots, a_n$ are distinct elements of A and $b_i \in B$, $i = 1, \ldots, n$, we write $f[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$ for that function from A to B which maps a_i to b_i and every other $x \in A$ to $f(x)$.

- **Identifiers**: If the given expression is an identifier I, and if I is bound in the given environment ρ to some value V, then the output value is V; it is an error if I is unbound in ρ.[4]
- **Unit**: The value of the expression `()` is always the unit value.
- **Term variables**, **meta-identifiers**, and **characters** are always self-evaluating, regardless of ρ, β, σ, and γ.
- **String constants**: A string constant results in a list of the characters that constitute the string, in the given order.
- **`check` expressions**: To evaluate an expression of the form

4 Note that when a function symbol f is first introduced, the name f is automatically bound to that function symbol. The name f can later perhaps be redefined (bound to another value), but the underlying function symbol will continue to exist in the relevant symbol set (and can always be retrieved, for example, by the `string->symbol` procedure). Numerals (such as `5` or `3.14`) can be understood as always being bound to the corresponding numeric terms.

$$\texttt{check}\ \{F_1\ \texttt{=>}\ E_1\ \texttt{|}\ \cdots\ \texttt{|}\ F_n\ \texttt{=>}\ E_n\}$$

in ρ, β, σ, and γ, we first evaluate F_1 in ρ, β, σ, and γ, possibly resulting in a value V_1 and store σ_1.[5] If V_1 is the constant term `true`, then the result of the entire **`check`** expression is the result obtained by evaluating E_1 in ρ, β, σ_1, and γ. Otherwise, if V_1 is `false`, the process continues with the second clause, taking into account any side effects that the evaluation of F_1 might have engendered: The phrase F_2 is evaluated in ρ, β, σ_1, and γ, possibly resulting in a value V_2 and a store σ_2. If V_2 is `true`, then the final result is that of evaluating E_2 in ρ, β, σ_2, and γ. Otherwise, if V_2 is `false`, F_3 is evaluated in ρ, β, σ_2, and γ; its value is compared to `true`, and so forth. Clearly, evaluation might diverge if some F_i or E_i diverges. It is an error if there are no alternatives (i.e., if $n = 0$), in which case we return an appropriate error message and the store σ as the result of the evaluation; or if no alternative succeeds (i.e., no F_i ever produces `true`), in which case we return an error message and the store σ_n; or if some F_i results in a value other than `true` or `false`, in which case we return an error message and the store σ_i. If F_n is the keyword **`else`** and no F_j produced `true`, $j < n$, then the final result is that of evaluating E_n in ρ, β, σ_{n-1}, and γ.[6]

- **Procedures**: The value of **`lambda`** $(I_1 \cdots I_n)\ E$ in ρ, β, σ, and γ is a function value that takes a list of n values $V_1, \ldots, V_n$ along with an assumption base β' and a store σ' as arguments, and produces the result of evaluating the body E in $\rho[I_1 \mapsto V_1, \ldots, I_n \mapsto V_n]$, β', σ', and γ. Note that the lexical environment and symbol set are statically determined, whereas the store and the assumption base are dynamic. The evaluation of procedures always terminates successfully.
- **Methods**: The value of **`method`** $(I_1 \cdots I_n)\ D$ in ρ, β, σ, and γ is a method value that takes a list of n values $V_1, \ldots, V_n$ along with an assumption base β' and store σ' as arguments, and produces the result of evaluating the deduction D in $\rho[I_1 \mapsto V_1, \ldots, I_n \mapsto V_n]$, β', σ', and γ. Here too, the lexical environment and symbol set are statically determined while the store and the assumption base are dynamic. The evaluation of methods always terminates successfully.
- **Applications**: The value of an expression of the form

  $$(E\ F_1 \cdots F_n)$$

  in ρ, β, σ, and γ is obtained as follows. First, E is evaluated in ρ, β, σ, and γ, possibly resulting in a value V and store σ'. We then proceed with a case analysis of V:

  1. If V is a function value ϕ, we evaluate the arguments $F_1, \ldots, F_n$ sequentially, starting with F_1 in ρ, β, σ', and γ, to obtain values $V_1, \ldots, V_n$ and a store σ_n. This sequential

5 In accordance with the convention made above, if the evaluation of F_1 in ρ, β, σ, and γ generates an error and some store σ_1, we simply return that as the result of the entire **`check`** expression.

6 The keyword **`else`** may only appear in the position of F_n.

evaluation threads the store from the evaluation of F_i to that of F_{i+1}. Specifically, first we evaluate F_1 in ρ, β, σ', and γ to obtain a value V_1 and a store σ_1; then we evaluate F_2 in ρ, β, σ_1, and γ to obtain a value V_2 and store σ_2; we continue in this fashion until we evaluate F_n in ρ, β, σ_{n-1}, and γ, resulting in a value V_n and store σ_n. The final result is that of applying ϕ to the list $[V_1, \ldots, V_n]$, β, and σ_n.

2. If V is a function symbol f of arity n, we sequentially evaluate $F_1, \ldots, F_n$, starting with F_1 in ρ, β, σ and γ, to obtain values $V_1, \ldots, V_n$ and a store σ_n. If each V_i is a term t_i, then the resulting value is the term $(f\ t_1 \cdots t_n)$, provided that this term is well sorted. If it is not, we return an error message and the store σ_n as the result. (We also return an error message and the store σ_n if some V_i is not a term.)
3. If V is a sentential connective $\circ$, we evaluate $F_1, \ldots, F_n$ sequentially to obtain values $V_1, \ldots, V_n$ and a store σ_n. If each V_i is a sentence p_i, then the resulting value is the sentence $(\circ\ p_1 \cdots p_n)$, provided that the latter is well sorted, and the resulting store is σ_n. If the sentence is not well sorted or if some V_i is not a sentence, we return an error message and the store σ_n.
4. If V is a quantifier Q, we check to make sure that $n > 1$; if not, we return an error message and the store σ'. We then evaluate $F_1, \ldots, F_n$ sequentially to obtain values $V_1, \ldots, V_n$ and a store σ_n. The first $n-1$ values must all be term variables $x_1, \ldots, x_{n-1}$, while the last value V_n must be a sentence p. If not, we halt with an error message and σ_n; otherwise the resulting value is the quantified sentence:

$$(Q\ x_1\ (Q\ x_2\ \cdots\ (Q\ x_n\ p)\ \cdots)),$$

provided that it is well sorted, while the resulting store is σ_n. If the above is not a well-sorted sentence, we return an error message and the store σ_n.[7]
5. If V is a substitution θ, we check whether $n = 1$. If not, we return an error message along with the store σ' and halt. Otherwise we evaluate F_1 in ρ, β, σ', and γ to obtain a value V_1 and a store σ_1. Then:

   - If V_1 is a term t_1 (or sentence p_1), the output is the term $\theta(t_1)$ (or sentence $\theta(p_1)$, respectively) and σ_1, provided that $\theta(t_1)$ (respectively, $\theta(p_1)$) is well sorted; if it is not, we return an error message and σ_1.[8]
   - If V_1 is a list `[`$V_1 \cdots V_n$`]` where each V_i is a term or sentence, then the output is the list $[\theta(V_1), \ldots, \theta(V_n)]$ and the store σ_1, provided that each $\theta(V_i)$ is well sorted. If one is not, of if V_1 is not such a list, we return an error message and the store σ_1.

7 Note that applications such as `(forall` x `.` p`)` are syntax sugar for `(forall` x p`)`. The latter is the more fundamental syntactic construct.

8 We write $\theta(t)$ (respectively, $\theta(p)$) for the term obtained by applying the substitution θ to the term t (respectively, sentence p).

- If V_1 is neither a term nor a list of terms, we halt with an error message and the store σ_1.

6. If V is a map, we check whether $n = 1$. If not, like before, we return an error message along with the store σ' and halt. Otherwise we evaluate F_1 in ρ, β, σ', and γ to obtain a value V_1 and a store σ_1. Then, if V_1 is a key in the map V, the output is the value that the map prescribes for that key, along with σ_1. If V_1 is not a key in that map, we return an error message and σ_1.
7. If V is neither a function value, nor a function symbol, sentential connective, quantifier, substitution, or map, then we halt with an error message and σ'.

- **`match` expressions**: To evaluate `match` F $\{\pi_1$ `=>` E_1 `|` $\cdots$ `|` π_n `=>` $E_n\}$ in ρ, β, σ, and γ, we first evaluate the discriminant F in ρ, β, σ, and γ, possibly obtaining a value V and store σ'. We then go through the patterns $\pi_1, \ldots, \pi_n$ sequentially, trying to find a pattern that is matched by V in ρ and γ (using the algorithm of Section A.4). If no such pattern is found, we halt with an error message and σ'. Otherwise, let π_j be the first matching pattern and let $\{I_1 \mapsto V_1, \ldots, I_k \mapsto V_k\}$ and τ be the finite set of bindings and sort valuation, respectively, returned by the matching algorithm. The output then becomes the result of evaluating E_j in $\rho[I_1 \mapsto V_1, \ldots, I_k \mapsto V_k]$, β, σ', and γ. If that result is a term or sentence, then τ is applied to it before returning.
- **`let` expressions**: For semantic purposes, an expression of the form

$$\texttt{let}\ \{\pi_1\ \texttt{:=}\ F_1\texttt{;}\ \cdots\texttt{;}\ \pi_n\ \texttt{:=}\ F_n\}\ E \tag{1}$$

  is treated as syntax sugar. The desugaring proceeds by induction on n. When $n = 0$, (1) reduces to E. When $n > 0$, (1) is desugared into `match` F_1 $\{\pi_1$ `=>` $E'\}$, where E' is the result of desugaring

$$\texttt{let}\ \{\pi_2\ \texttt{:=}\ F_2\texttt{;}\ \cdots\texttt{;}\ \pi_n\ \texttt{:=}\ F_n\}\ E.$$

- **`letrec` expressions**: The result of evaluating

$$\texttt{letrec}\ \{I_1\ \texttt{:=}\ E_1\texttt{;}\ \cdots\texttt{;}\ I_n\ \texttt{:=}\ E_n\}\ E$$

  in ρ, β, σ, and γ is that of evaluating the following expression in ρ, β, σ, and γ :

```
let {I1 := (cell ()); ···; In := (cell ());
     _  := (set! I1 E1');
           ···;
     _  := (set! In En')}
  E'
```

where each E'_j is obtained from E_j by replacing every free occurrence of I_j[9] by `(ref` I_j`)`, $j = 1, \ldots, n$; and E' is likewise obtained from E. This desugaring is the classic way to "tie the knot," and it is often used to implement recursion when the implementation language supports state. The various E_i are usually **`lambda`** or **`method`** expressions.

- **`try` expressions**: The **`try`** construct implements backtracking: To evaluate

  `try` $\{E_1$ `|` $\cdots$ `|` $E_n\}$

  in ρ, β, σ, and γ, we first evaluate E_1 in ρ, β, σ, and γ. If that results in a value V_1 and store σ_1, we return V_1 and σ_1. But if it results in an error message and a store σ_1, we go on to evaluate E_2 in ρ, β, σ_1, and γ; and so forth. If the evaluation of every E_i fails, we return an error message and σ_n.

- **Cells**: To evaluate an expression of the form **`cell`** F in ρ, β, σ, and γ, we first evaluate F in ρ, β, σ, and γ to obtain some value V and store σ'. We then let l be the smallest natural number such that $\sigma'(l)$ is unassigned, and we return as output the cell represented by the location l along with the store $\sigma'[l \mapsto V]$.

- **References**: To evaluate an expression of the form **`ref`** E in ρ, β, σ, and γ, we evaluate E in ρ, β, σ, and γ, possibly obtaining a value V and store σ'. If the value V is *not* a memory location (cell), we return an error message and σ'. If it is a memory location l, we return the value $\sigma'(l)$ and the store σ'.[10]

- **Assignments**: To evaluate an expression of the form **`set!`** E F in ρ, β, σ, and γ, we first evaluate E in ρ, β, σ, and γ, possibly obtaining a value V_1 and store σ_1. If V_1 is not a memory location (cell), we halt with an error message and σ_1 as the result. If it is a cell l, we proceed to evaluate F in ρ, β, σ_1, and γ, possibly obtaining a value V_2 and store σ_2. We then return the unit value and the store $\sigma_2[l \mapsto V_2]$.

- **`while` loops**: To evaluate an expression of the form **`while`** F E in ρ, β, σ, and γ:

  1. We evaluate F in ρ, β, σ, and γ, possibly obtaining a value V_1 and store σ_1.
  2. If V_1 is the constant term `false`, we return the unit value and store σ_1. Otherwise, if V_1 is `true`, we evaluate E in ρ, β, σ_1, and γ to obtain some value V_2 and store σ_2. We then continue with the first step, only now F is evaluated in ρ, β, σ_2, and γ (rather than ρ, β, σ, and γ). If V_1 is neither `true` nor `false`, we return an error message and σ_1.

- **Short-circuit boolean operations**: Expressions of the `&&` and `||` form `(&&` $F_1 \cdots F_n$`)` and `(||` $F_1 \cdots F_n$`)` are evaluated as explained in Section 2.16, making sure to thread the store in the obvious manner (the store resulting from the evaluation of F_1 in ρ, β, σ, and

9 We have not given a precise definition of when an identifier occurs free inside a phrase, but an intuitive understanding will suffice for present purposes.

10 It is an error if cell l is unassigned, although this could never happen in our semantics.

γ becomes the store in which the evaluation of F_2 takes place, if F_2 is evaluated at all; and so on).

- **Vector creation**: To evaluate an expression `make-vector` E F in ρ, β, σ, and γ, we start by evaluating E in ρ, β, σ, and γ to obtain a value V_1 and a store σ_1. If V_1 is not a nonnegative integer constant, we halt with an error message and σ_1. If it is a nonnegative integer constant n, we proceed to evaluate F in ρ, β, σ_1, and γ, possibly obtaining a value V_2 and store σ_2. We then let $l_1, \ldots, l_n$ be the smallest n natural numbers such that $\sigma_2(l_i)$ is unassigned for each $i = 1, \ldots, n$, and we return as output the list of memory locations $[l_1 \cdots l_n]$ along with the store $\sigma_2[l_1 \mapsto V_2, \ldots, l_n \mapsto V_2]$.
- **Vector access**: To evaluate an expression `vector-sub` E_1 E_2 in ρ, β, σ, and γ, we first evaluate E_1 in ρ, β, σ, and γ, obtaining a value V_1 and store σ_1. If V_1 is not a list of memory locations previously created by `make-vector`,[11] we output an error message and σ_1. If V_1 is a list of memory locations $[l_1 \cdots l_n]$ previously created by `make-vector`, $n \geq 0$, we proceed to evaluate E_2 in ρ, β, σ_1, and γ, obtaining a value V_2 and store σ_2. If V_2 is not a nonnegative integer constant, we halt with an error message and σ_2. If it is a nonnegative integer constant i, we return the value $\sigma_2(l_{i+1})$ and σ_2, provided that the index i is between 0 and $n - 1$ (if it is not, we halt with an error message and σ_2).
- **Vector assignment**: To evaluate an expression `vector-set!` E_1 E_2 F in ρ, β, σ, and γ, we first evaluate E_1 in ρ, β, σ, and γ, obtaining from it some value V_1 and store σ_1. If V_1 is not a list of memory locations previously created by `make-vector`, we output an error message and σ_1. If V_1 is a list of memory locations $[l_1 \cdots l_n]$ previously created by `make-vector`, $n \geq 0$, we proceed to evaluate E_2 in ρ, β, σ_1, and γ, obtaining a value V_2 and store σ_2. If V_2 is not a nonnegative integer constant between 0 and $n - 1$, we halt with an error message and σ_2. Otherwise, if it is a some such constant i, we proceed to evaluate F in ρ, β, σ_2, and γ, obtaining from it some value V_3 and store σ_3. We then return the unit value along with the store $\sigma_3[l_{i+1} \mapsto V_3]$.

We continue with the evaluation of deductions, which again proceeds by a case analysis of syntactic structure:

- **Method calls**: To evaluate a deduction $(!E\ F_1 \cdots F_n)$ in ρ, β, σ, and γ, we first evaluate E in ρ, β, σ, and γ, obtaining a value V and store σ' from it. If V is not a method value, we halt with an error message and σ'. Otherwise, if V is a method value M, we go on to evaluate the arguments $F_1, \ldots, F_n$, in that turn. Letting $\sigma_0 = \sigma'$ and $\beta' = \emptyset$, we evaluate each F_i in ρ, β, σ_{i-1}, and γ, obtaining from it a value V_i and store σ_i. In addition, if the argument F_i is a deduction whose value V_i is a conclusion (sentence) p_i, we add p_i to β'. When all arguments have been evaluated, we apply M to the list of values $[V_1, \ldots, V_n]$,

11 Strictly speaking this requires that we tag lists of cells created by `make-vector` to distinguish them from lists of cells that may have been created by other means, but this need not detain us here.

$\beta \cup \beta'$, and σ_n. Note that the assumption base in which M is applied will include the conclusion of every argument phrase F_i that is a deduction.

- **Conclusion-annotated deductions**: To evaluate a deduction `conclude` F D in ρ, β, σ, and γ, we first evaluate F in ρ, β, σ, and γ, to obtain a value V and a store σ'. If V is not a sentence, we halt with an error message and σ'. Otherwise, if V is a sentence p, we proceed to evaluate the body D in ρ, β, σ', and γ, obtaining from it a conclusion q and store σ''. If p and q are alpha-equivalent, we halt with p and σ'' as the output; otherwise we halt with an appropriate error message and σ''.
- **Hypothetical deductions**: To evaluate `assume` F D in ρ, β, σ, and γ, we first evaluate F in ρ, β, σ, and γ, producing a value V and store σ'. If V is not a sentence, we halt with an error message and σ'. Otherwise, if V is a sentence p, we proceed to evaluate the body D in ρ, $\beta \cup \{p\}$, σ', and γ, obtaining from it some sentence q and store σ''. We then return the conditional (`p ==> q`) and σ'' as the result.
- **Named Hypothetical deductions**: A deduction `assume` I := F D is treated as syntax sugar for the following:

```
let {I := F}
  assume I
    D
```

- **Proof by contradiction**: To evaluate `suppose-absurd` F D in ρ, β, σ, and γ, we begin by evaluating F in ρ, β, σ, and γ, producing a value V and store σ'. If V is not a sentence, we halt with an error message and σ'. If V is a sentence p, we evaluate the body D in ρ, $\beta \cup \{p\}$, σ', and γ, obtaining from it some sentence q and store σ''. If q is the constant `false`, we return (`~ p`) and σ'' as the result; if q is not `false`, we output an error message and σ''.
- **Universal generalizations**: To evaluate a deduction `generalize-over` E D in ρ, β, σ, and γ, we first evaluate E in ρ, β, σ, and γ, obtaining some value V and store σ'. If V is a variable x (say, `?p:Person`), we first check to see that x does not occur free in β.[12] If it does, we halt with an error message and σ'. If x does not occur free in β, we proceed to evaluate the body D in ρ, β, σ', and γ, obtaining a conclusion p and store σ''. We then return the sentence (`forall x p`) and σ'', provided that (`forall x p`) is well sorted (if it is not, we output an error message and σ'').
- `pick-any` **universal generalizations**: A deduction of the form `pick-any` I D is treated as syntax sugar for:

```
let {I := (fresh-var)}
```

12 A variable `?`I`:`S is said to occur free in an assumption base β iff there is some $p \in \beta$ such that p contains a free occurrence of some variable of the form `?`I`:`S', where S is an instance of S'. Thus, if $\beta = \{$`(P ?z:'S3)`$\}$, then `?z:Person` is considered to occur free in β, as `Person` is an instance of `'S3`.

```
generalize-over I
  D
```

And a deduction of the form **pick-any** $I : S$ D is treated as syntax sugar for

```
let {I := (fresh-var "S")}
  generalize-over I
    D
```

- **with-witness deductions**: To evaluate **with-witness** E F D in ρ, β, σ, and γ, we first evaluate E in ρ, β, σ, and γ, to obtain a value V and store σ'. If V is not an Athena variable, or if it is a variable that occurs free in β, we halt with an error message and σ'. Otherwise, if V is a variable w that does not occur free in β, we proceed to evaluate F in ρ, β, σ', and γ, obtaining a value V' and store σ''. If V' is not an existential quantification, we halt with an error message and σ''. Otherwise V' must be an existential quantification $p =$ `(exists` y `.` q`)`. If the phrase F is a deduction, let $\beta' = \beta \cup \{p\}$; otherwise let $\beta' = \beta$. Now, let q' be the sentence obtained from the body q by replacing every free occurrence of y by the variable w, provided that the result is well sorted (if not, we halt with an error message and σ''). We then evaluate the deduction D in ρ, $\beta' \cup \{q'\}$, σ'', and γ, obtaining some sentence r and store $\widehat{\sigma}$. If the variable w occurs free in r, we halt with an error message and $\widehat{\sigma}$, otherwise we return r and $\widehat{\sigma}$.
- **pick-witness deductions**: A deduction **pick-witness** I **for** F D is treated as syntax sugar for

```
let {I := (fresh-var)}
  with-witness I F D
```

A deduction **pick-witness** I **for** F I_2 D is treated as syntax sugar for

```
let {I1 := (fresh-var);
     equant := F}
  match equant {
    (exists (some-var v) body) =>
      let {I2 := (replace-var v I1 body)}
        with-witness I1 equant D
  }
```

(Recall that `(replace-var` v t p`)` produces the sentence obtained from p by replacing every free occurrence of variable v with the term t, provided that the result is well sorted.)

- **check deductions**: A deduction **check** $\{F_1$ `=>` D_1 `|` $\cdots$ `|` F_n `=>` $D_n\}$ is evaluated by the same algorithm used for **check** expressions, save for the obvious difference that,

barring error or divergence, the output value will always be a sentence produced by some D_i.

- **`match` deductions**: To evaluate **`match`** F {π_1 => D_1 | $\cdots$ | π_n => D_n} in ρ, β, σ, and γ, we first evaluate the discriminant F in ρ, β, σ, and γ, obtaining a value V and store σ'. If the discriminant is a deduction, then V must be a sentence p, and we let $\beta' = \beta \cup \{p\}$. If, by contrast, F is an expression rather than a deduction, we let $\beta' = \beta$. We then go through the patterns $\pi_1, \ldots, \pi_n$ sequentially, trying to find a pattern that is matched by V in ρ and γ. If no such pattern is found, we halt with an error message and σ'. Otherwise, let π_j be the first matching pattern and let $\{I_1 \mapsto V_1, \ldots, I_k \mapsto V_k\}$ and τ be the set of bindings and sort valuation returned by the matching algorithm, respectively. The output then becomes the result of applying τ to the conclusion obtained by evaluating D_j in $\rho[I_1 \mapsto V_1, \ldots, I_k \mapsto V_k]$, β', σ', and γ. Note that if the discriminant is a deduction, then its conclusion is available as a lemma inside D_j.
- **`let` deductions**: A deduction of the form

$$\texttt{let}\ \{\pi_1\ \texttt{:=}\ F_1;\ \cdots;\ \pi_n\ \texttt{:=}\ F_n\}\ D \tag{2}$$

  is treated as syntax sugar, with the desugaring proceeding by induction on n. When $n = 0$, (2) reduces to D. When $n > 0$, (2) is desugared into **`match`** F_1 {π_1 => D'}, where D' is the result of desugaring **`let`** {π_2 := F_2; $\cdots$; π_n := F_n} D.
- **`letrec` deductions**: A deduction of the form **`letrec`** {I_1 := E_1; $\cdots$; I_n := E_n} D is desugared into the following:

```
let {I_1 := (cell ());
     ...
     I_n := (cell ());
     _ := (set! I_1 E'_1);
     ...
     _ := (set! I_n E'_n)}
  D'
```

  where each E'_j is obtained from E_j by replacing every free occurrence of I_j by `(ref` I_j`)`, $j = 1, \ldots, n$; and D' is likewise obtained from D.
- **`try` deductions**: To evaluate **`try`** {D_1 | $\cdots$ | D_n} in ρ, β, σ, and γ, we first evaluate D_1 in ρ, β, σ, and γ. If that results in a sentence p and store σ_1, we return p and σ_1 as the result. But if it results in an error message and a store σ_1, we go on to evaluate D_2 in ρ, β, σ_1, and γ; and so forth. If the evaluation of every D_i fails, we return an error message and σ_n.
- **Structural induction**: To evaluate a deduction of the form

$$\texttt{by-induction}\ F\ \{\pi_1\ \texttt{=>}\ D_1\ \texttt{|}\ \cdots\ \texttt{|}\ \pi_n\ \texttt{=>}\ D_n\} \tag{3}$$

in ρ, β, σ, and γ, we start by evaluating F in ρ, β, σ, and γ, obtaining a value V and store σ_1. If V is not a universally quantified sentence of the form

$$\texttt{(forall } I\texttt{:}S_D\ p\texttt{)} \tag{4}$$

for some inductively generated sort S_D in γ,[13] we halt with an error message and σ_1. Otherwise, we check to make sure that the patterns $\pi_1, \ldots, \pi_n$ are proper patterns for S_D (in accordance with γ) and that they are jointly exhaustive.[14] We will now explain in detail what it means for the patterns to be "proper" for S_D in the context of γ. Recall that a pattern π_i here is not an arbitrary Athena pattern (as can appear, e.g., inside a `match` clause), but is instead an element of a more restricted class of patterns described by the following grammar:

$$\pi ::= I \mid I\texttt{:}S \mid \texttt{(}I\ \pi^+\texttt{)} \mid \texttt{(}I\ \texttt{as}\ \pi\texttt{)} \tag{5}$$

where S ranges over sorts. An identifier I inside such a pattern is said to be a *pattern variable* iff it is not the name of a function symbol in γ. It is an error if there are any duplicate pattern variable occurrences in one of the patterns.

To each given pattern π we assign a term t_π, defined by structural recursion on the syntax of π as explained below. The algorithm takes as input not only the pattern π but also a finite function M mapping pattern variables to terms (mostly term variables), and it outputs not only the term t_π but also an extension of the mapping M. Initially the algorithm is invoked with $M = \emptyset$. The algorithm is this:

1. If π is an identifier I, then if I is the name of a constant symbol c in γ, we let t_π be c and return M unchanged. Otherwise I must be a pattern variable, and in that case we let t_π be a *fresh* variable x and we return x and $M[I \mapsto x]$. Note that the sort of this fresh Athena variable will be some fresh sort variable (e.g., `'T145`).
2. If π is of the form $I\texttt{:}S$ for some sort S in γ, we proceed as above except that now, if I is a pattern variable, the fresh variable x will be of sort S (rather than some fresh sort variable, as before). Also, if I is not a pattern variable, we need to ensure that $I\texttt{:}S$ is a well-sorted term.
3. If π is of the form $\texttt{(}I\ \pi_1 \cdots \pi_k\texttt{)}$ for some $k > 0$, we apply the algorithm recursively to $(\pi_1, M_0), (\pi_2, M_1), \ldots, (\pi_k, M_{k-1})$, starting with $M_0 = M$, obtaining outputs $(t_{\pi_1}, M_1), \ldots, (t_{\pi_k}, M_k)$. Thus, each output map M_i becomes the input map to the next call. Finally, if I is the name of a constructor of S_D (according to the information in γ), we output $(\texttt{(}I\ t_{\pi_1} \cdots t_{\pi_k}\texttt{)}, M_k)$, provided that $\texttt{(}I\ t_{\pi_1} \cdots t_{\pi_k}\texttt{)}$ is a well-sorted term (an error is generated if it is not).

13 By an inductively generated sort we mean a sort $\texttt{(}S\ S_1 \cdots S_k\texttt{)}$, $k \geq 0$, whose outer sort constructor S is the name of a datatype or structure. Thus, e.g., both `N` and `(List 'T35)` are datatype sorts.

14 Joint exhaustiveness means that every canonical term of sort S_D matches some such pattern. There is an algorithm for deciding that condition, though we need not describe it here.

4. Finally, if π is of the form (*I* **as** π), we apply the algorithm recursively to (π, M) to obtain an output (t_π, M_1) and we then return $(t_\pi, M_1[I \mapsto t_\pi])$.

A pattern π_i in (3), $i \in \{1, \ldots, n\}$, is considered proper for S_D in the context of γ provided that the term t_{π_i} obtained by applying the above algorithm to $(\pi_i, \emptyset)$ is a legal term of sort S_D.

Note that the final sort checking of a term t_{π_i} might refine the sorts of the fresh variables that appear in the mapping M produced by the algorithm. Suppose, for example, that the pattern is `(:: head tail)`. Here `::` is a function symbol (a constructor of the datatype `List`), while `head` and `tail` are pattern variables. Since both `head` and `tail` are unannotated, they will be mapped to fresh variables of completely unconstrained sorts, say to `?v10:'T145` and `?v11:'T147`, respectively. Later, sort inference will refine these sorts by realizing that `'T147` must be `(List 'T145)`. We assume that the sorts of the various fresh variables that appear in the map returned by the algorithm have been properly updated in this fashion to reflect constraints inferred by sort checking. In this case, for instance, the map M might assign the fresh variables `?v10:'T145` and `?v11:(List 'T145)` to `head` and `tail`, respectively: $M = \{\ldots,$ `head` $\mapsto$ `?v10:'T145, tail` $\mapsto$ `?v11:(List 'T145), ...`$\}$.

Once we have gone through each pattern π_i and computed the corresponding term t_{π_i} and mapping M_i and ensured that each π_i is proper for S_D in the context of γ, we proceed to do the following for each clause π_i `=>` D_i, $i = 1, \ldots, n$. First, we compute all the appropriate inductive hypotheses for the clause. Specifically, for each fresh variable x in the range of M_i whose sort is an instance of the datatype sort S_D, we generate an inductive hypothesis, obtained from the body p by replacing every free occurrence of $I{:}S_D$ by x. Let β_i consist of β augmented with all the inductive hypotheses thus generated, and let ρ_i be the environment ρ augmented with the bindings of M_i. We then evaluate D_i in ρ_i, β_i, σ_i, and γ, producing some conclusion p_i and store σ_{i+1} (this new store will become the one in which the deduction of the next clause will be evaluated). We must now check that p_i is of the right form; if it is not, we will halt with an error message and σ_{i+1}. We say that p_i is of the right form if it is alpha-equivalent to the sentence we obtain from p by replacing every free occurrence of $I{:}S_D$ by t_{π_i}. If p_i is indeed of the right form, we continue with the next clause. If this process is successfully carried out for every clause, we return as output the universal generalization (4) along with the store σ_{n+1}.

The generation of inductive hypotheses is somewhat more sophisticated than just described, allowing, in particular, for nested inductive proofs. We give a brief illustrating example. Suppose we have a datatype describing the abstract syntax of λ-calculus expressions as follows:

```
datatype Exp :=
  (var Ide)                    # variables
| (abs Ide Exp)                # abstractions
| (app Exp (List Exp))         # applications
```

The interesting point for our purposes here is that the reflexive constructor App takes a *list* of expressions as its second argument. Should we ever need to perform an induction on the structure of that list in the midst of performing an outer structural induction on Exp, appropriate inductive hypotheses should be generated not only for the tail of that list, but also for the head of the list on the basis of the outer induction. Suppose, for example, that we have some unary procedure exp-property that takes an arbitrary expression e and builds some sentence expressing a proposition about e, and we are interested in proving (forall e . exp-property e) (where e is a variable ranging over Exp). We proceed by structural induction:

```
by-induction (forall e . exp-property e) {
  (var x) => D1
| (abs x e) => D2
| (app proc args) => D3
}
```

Consider now D_3, the proof in the third clause. It is not uncommon in such cases to proceed by induction on the structure of the list args, that is, to show that for all such lists L, we have (exp-property (app proc L)). Our goal will then follow simply by specializing L with args:

```
by-induction (forall e . exp-property e) {
  (var x) => D1
| (abs x e) => D2
| (app proc args) =>
    let {list-property := lambda (L)
                            (exp-property (app proc L));
         lemma := by-induction (forall L (list-property L)) {
                    nil => D3
                  | (:: head:Exp tail:(List Exp)) => D4
                  }
        }
    (!uspec lemma args)
}
```

Here, inside D_4 we will not only get an appropriate inductive hypothesis for tail (namely, (list-property tail)),[15] but we will also get one for head, namely (exp-property head). Athena knows that it is appropriate to generate this outer inductive hypothesis for head because it is keeping track of the nesting of the various structural

15 Strictly speaking, of course, when we say "tail" we are referring to the fresh term variable to which tail will be bound.

inductions and their respective goals, as well as the sorts of the corresponding universally quantified variables, and realizes that, in the given context, `head` will in fact be a "smaller" expression than the outer `(app proc args)`, since `head` will essentially be a proper part of `args`, which is itself a proper part of `(app proc args)`.[16] Inductive proofs can be nested to an arbitrarily deep level (not just lexically, i.e., as determined by the nesting of the proofs in the actual text, but dynamically as well, via method calls), and for every clause of every such proof, appropriate inductive hypotheses will be generated for every enclosing inductive proof.

- **Structural case analysis**: A deduction of the form

$$\texttt{datatype-cases}\ F\ \{\pi_1\ \texttt{=>}\ D_1\ \texttt{|}\ \cdots\ \texttt{|}\ \pi_n\ \texttt{=>}\ D_n\} \tag{6}$$

  is evaluated (in some ρ, β, σ, and γ) exactly like a structural induction deduction of the form (3), except that no inductive hypotheses are generated for the various cases.

There is a variant of the **`datatype-cases`** construct that is occasionally convenient, namely:

$$\texttt{datatype-cases}\ F\ \texttt{on}\ E\ \{\pi_1\ \texttt{=>}\ D_1\ \texttt{|}\ \cdots\ \texttt{|}\ \pi_n\ \texttt{=>}\ D_n\}$$

Here, the evaluation of F in ρ, β, σ, and γ must produce a sentence p, along with a store σ_1. Further, the evaluation of E in ρ, β, σ_1, and γ must produce a variable $x{:}S$, where S is a datatype, and where p possibly contains free occurrences of $x{:}S$. The idea now is that we want to derive p, and we will proceed by a constructor case analysis on $x{:}S$. That is, because we know that $x{:}S$ is of sort S, and because we know that every value of S is obtainable by some constructor application, we essentially reason that $x{:}S$ must be of the form of one of the n listed patterns, $\pi_1, \ldots, \pi_n$. Assuming that these patterns jointly exhaust S (a condition that is checked after F and E are evaluated), we must then show that each deduction D_i derives the desired goal p but with every free occurrence of $x{:}S$ replaced by the term t_{π_i}, where t_{π_i} is obtained from π_i exactly as described above for **`by-induction`** proofs. There

16 Note, however, that the generation of outer induction hypotheses depends crucially on the form of the inner goal. While in a case such as shown here it would indeed be valid to generate an outer inductive hypothesis for `head`, that might not be so if `list-property` were defined differently. Athena will generate an outer inductive hypothesis only if it can ensure that it is appropriate to do so by conservatively matching the current (inner) goal to the outer goal. Specifically, an outer inductive hypothesis will be generated only if it can be shown that the current goal is an instance of the outer goal via some substitution of the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ such that (a) each x_i is a variable in the outer inductive pattern and (b) each t_i is the current inductive pattern. In this example, the current goal (inside D_4) is `(list-property (:: head tail))`, that is, `(exp-property (app proc (:: head tail)))`. The prior goal (in the current stack of inductive clauses) is `(exp-property (app proc args))`. Thus, the current goal matches the prior goal under the substitution {`args` ↦ `(:: head tail)`}, where `args` is a variable in the outer inductive pattern and `(:: head tail)` is the current (inner) inductive pattern. Since `head` is a variable in the current inductive pattern of the same sort as the universally generalized variable in the outer clause, and since the substitution conditions (a) and (b) are satisfied, an outer inductive hypothesis will be generated in this case. We do not define here the notion of one sentence matching another sentence under some substitution, but see page 97 for a related discussion (alternatively, consult the definition of the procedure `match-props` in `lib/basic/util.ath`).

π	::=	I	(Pattern variables, symbols, connectives, quantifers)
	|	$I{:}S$	(Sort-annotated pattern variables or constant symbols)
	|	$?I{:}S$	(Athena variables)
	|	$'I$	(Meta-identifiers)
	|	C	(Character constants)
	|	T	(String constants)
	|	()	(Unit pattern)
	|	_	(Wildcard pattern)
	|	(**bind** I π)	(Named patterns, same as (I **as** π))
	|	(**val-of** I)	(**val-of** patterns)
	|	(**list-of** π_1 π_2)	(**list-of** patterns)
	|	(**split** π_1 π_2)	(**split** patterns)
	|	[$\pi_1 \cdots \pi_n$]	(Fixed-length list patterns)
	|	($\pi_1 \cdots \pi_n$)	(Compound patterns)
	|	(π **where** E)	(Where patterns)
	|	(**some-var** J)	
	|	(**some-sent-con** J)	
	|	(**some-quant** J)	
	|	(**some-term** J)	
	|	(**some-atom** J)	
	|	(**some-sentence** J)	
	|	(**some-list** J)	
	|	(**some-cell** J)	
	|	(**some-vector** J)	
	|	(**some-proc** J)	
	|	(**some-method** J)	
	|	(**some-symbol** J)	
	|	(**some-table** J)	
	|	(**some-map** J)	
	|	(**some-sub** J)	
	|	(**some-char** J)	
J	::=	I | _	

Figure A.4
Syntax of Athena patterns

are no inductive hypotheses in this case, but each D_i is evaluated in β augmented with the identity $(x{:}S = t_{\pi_i})$. If the evaluation of each D_i in the corresponding environment (obtained from ρ and π_i as in a **by-induction** proof), the aforementioned assumption base, the appropriate store, and γ, results in the correct sentence (namely, p with every free occurrence of $x{:}S$ replaced by t_{π_i}), then p is finally produced as a result.

A.4 Pattern matching

Athena's pattern language is shown in Figure A.4. These patterns make it easy to take apart complicated structures. Disregarding parentheses and square brackets, a pattern is made up of (a) keywords such as **`list-of`** and **`some-quant`**, and (b) three kinds of identifiers:

1. The sentential connectives and quantifiers: `not`, `and`, `or`, `if`, `iff` (or their infix counterparts), along with `forall` and `exists`.
2. Function symbols, such as `nil`, `true`, and so on.
3. All remaining identifiers, which serve as *pattern variables*.

Function symbols along with sentential connectives and quantifiers are pattern constants, in that they can only be matched by the corresponding values. Pattern variables, on the other hand, can be matched by arbitrary values.

Pattern matching works as follows: a value V is matched against a pattern π, with respect to some environment ρ and symbol set γ, and results either in failure (in which case we say that V did not match the pattern); or in an environment (finite set of bindings) $\rho' = \{I_1 \mapsto V_1, \ldots, I_n \mapsto V_n\}$ that assigns values to the pattern variables of π, along with a sort valuation, that is, a finite function from sort variables to sorts. Suppose, for example, that V is the term `(S S zero)` and π is the pattern `(S n)`. Here we have a successful match, resulting in the environment {`n` $\mapsto$ `(S zero)`} (and the empty sort valuation).

In what follows we describe in detail an algorithm that takes as input: (a) a value V, (b) a pattern π, (c) an environment ρ, and (d) a symbol set γ; and outputs either (i) a failure token, indicating that V does not match π with respect to ρ and γ, or else (ii) an environment ρ' that represents a successful match, along with a sort valuation τ'. Both the output environment ρ' and the output sort valuation τ' are built up in stages, and it is convenient to express the algorithm so that it takes ρ' and τ' as two additional inputs. Initially the algorithm is called with $\rho' = \emptyset$ and $\tau' = \emptyset$.

The sort valuation is needed primarily in order to handle polymorphism. For instance, we would like to ensure that a term such as

```
(pair ?x:'S ?y:'T)
```

is successfully matched against a pattern like `(pair left:Ide right:Real)`. Intuitively, `(pair ?x:'S ?y:'T)` represents infinitely many terms, as many as can be obtained by consistently replacing the sort variables `'S` and `'T` by ground sorts. One of these infinitely many terms is the term `(pair ?x:Ide ?y:Real)`, obtained by replacing `'S` by `Ide` and `'T` by `Real`; and that term does indeed match the given pattern, under {`left` $\mapsto$ `?x:Ide`, `right` $\mapsto$ `?y:Real`}; our pattern-matching algorithm should infer this automatically. Roughly, when matching a term or sentence against a pattern, our algorithm will first try to unify the sorts

of the various components of the term (or sentence) with the sort constraints expressed in the pattern. If that succeeds, a match may be obtained; otherwise the match will fail. The incrementally built sort valuation is used primarily for that purpose.

In summary, then, the algorithm takes as input: (a) a value V; (b) a pattern π; (c) an environment ρ; (d) a symbol set γ; (e) the auxiliary environment ρ' that is to be incrementally built up; and (f) a sort valuation τ.[17] The empty set $\emptyset$ is given as the value of the last two arguments when the algorithm is first invoked. The algorithm will either fail or it will produce a pair (ρ'', τ') consisting of an environment ρ'' and a sort valuation τ' that extend ρ' and τ, respectively.[18] The algorithm proceeds by a case analysis of the structure of π:

- Case 1: π is the wildcard pattern `_`. In that case return (ρ', τ).
- Case 2: π is a term variable of sort S (such as `?x:Boolean`). In that case, if V is a term variable of sort T such that $\tau(S)$ and $\tau(T)$ are unifiable under a most general unifier τ', return (ρ', τ''), where τ'' is the composition of τ and τ'; otherwise fail.
- Case 3: π is a meta-identifier (such as `'foo`), or a character (such as `‘A`), or a string (such as `"Hello world!"`), or the unit value `()`. Then if V is the exact same value (`'foo`, `‘A`, etc.), return (ρ', τ); otherwise fail.
- Case 4: π is an identifier I, possibly annotated with a sort S. Then if I is a sentential connective or quantifier, the match succeeds iff V is the corresponding value and there is no sort annotation attached to I, in which case we simply return ρ' and τ unchanged. Otherwise we consult γ to see whether I is a function symbol.

  1. If it is, we check to see if there is a sort annotation S:

     - If there is, we check whether the value V is a constant term t whose root is I. If it is not, we fail. If it is, let S_t be the sort of t. We then check to see whether $\tau(S_t)$[19] and $\tau(S)$ can be unified under some most general unifier τ'. If not, we fail, otherwise we return ρ' unchanged along with the composition of τ' and τ.

17 The arguments ρ and γ are only used for lookups, so these two values could be held fixed throughout. An inner matching algorithm that would do all the work could then take four arguments only: (a), (b), (e), and (f). But for simplicity we describe a single six-input algorithm here.

18 To be perfectly complete and precise, we would also need to pass two additional inputs to the matching algorithm: an assumption base and a store. These are needed to evaluate expressions in **`where`** patterns. And we would also need to modify the output of the pattern matching algorithm to include an output store, that which might be produced as a side effect of evaluating such expressions. However, because these are needed only for **`where`** patterns, we omit them from the overall specification in order to avoid further complicating the description of the algorithm. The changes that would be needed to arrive at a working implementation are straightforward.

19 For any sort S and sort valuation τ, $\tau(S)$ denotes the sort obtained from S by replacing every occurrence of a sort variable in S by the unique sort that τ assigns to that variable (if a variable is not in the domain of τ, then it is returned unchanged).

- If there is not, we check to see if the value V is that exact same function symbol (I).[20] If it is not, we fail; otherwise we return ρ' and τ unchanged.

2. If it is not, that means that I is a pattern variable. We then check to see if ρ' already assigns a value V_I to I:

   - Suppose that it does. Then we again check whether there is a sort annotation S:

     * If there is not, then we fail if V and V_I are not identical; otherwise, if the two values are the same, we return ρ' and τ unchanged.
     * If there is a sort annotation S, we proceed as follows. If either V or V_I is not a term value, we fail. Otherwise both V and V_I are term values, call them t and t' respectively, with corresponding sorts S_t and $S_{t'}$. Then if the sorts $\tau(S)$ and $\tau(S_t)$ are not unifiable, we fail. If they are unifiable under some τ', then let t_1 and t'_1 be the terms obtained from t and t', respectively, by applying the composition of τ' and τ to their sorts.[21] If these two terms are identical, then we return ρ' extended with the assignment that maps I to t_1, along with the aforementioned composition. If t_1 and t'_1 are not identical, we fail.

   - Suppose that it does not. We again check to see whether there is a sort annotation:

     * If not, then we return (a) ρ' augmented with $\{I \mapsto V\}$; and (b) τ unchanged.
     * If I is annotated with a sort S, then we check to see if the value V is a term t, with some sort S_t. If it is not a term, we fail. Otherwise, we check whether $\tau(S_t)$ and $\tau(S)$ are unifiable under some τ'. If not, we fail. Otherwise, we return (a) ρ' augmented with $\{I \mapsto t'\}$, where t' is the term obtained by applying the composition of τ' and τ to t; and (b) the said composition.

- Case 5: π is a list pattern of the form $[\pi_1 \cdots \pi_n]$. In that case, if V is a list of values $[V_1 \cdots V_n]$, we return the result of trying to sequentially match $V_1, \ldots, V_n$ against the patterns $\pi_1, \ldots, \pi_n$ in ρ, γ, ρ', and τ (see the paragraph at the end of this section on how to sequentially match a number of values $V_1, \ldots, V_n$ against patterns $\pi_1, \ldots, \pi_n$). Otherwise we fail.
- Case 6: π is a list pattern of the form (`list-of` π_1 π_2). In that case, if V is a nonempty list of values $[V_1 \cdots V_n]$, $n > 0$, we return the result of trying to sequentially match the values $V_1, [V_2 \cdots V_n]$ against the patterns π_1, π_2 (again, in ρ, γ, ρ', and τ). Otherwise we fail.

---

20 Note that a constant term (such as `zero` or `nil`) can be coerced into a function symbol and vice versa.

21 To apply a sort valuation to a term means to apply the sort valuation to every sort annotation in that term.

- Case 7: π is a list pattern of the form (**split** π_1 π_2). In that case, if V is not a list of values, we fail. Otherwise, we determine whether V can be expressed as the concatenation of two lists L_1 and L_2 such that L_1 is the smallest prefix of V for which the values L_1, L_2 sequentially match the patterns π_1, π_2 under ρ, γ, ρ', and τ, producing a result (ρ'', τ'). If so, we return that result. If no such decomposition of V exists, we fail.
- Case 8: π is of the form (**val-of** I). We then check to see if I is bound in ρ. If it is not, we fail. If it is bound to some value V', we check whether the two values V and V' are identical. If the values do not admit equality testing, or if they do but are not identical, we fail, else we return ρ' and τ unchanged.
- Case 9: π is of the form (**bind** I π'), or equivalently, (I **as** π'). In that case we match V against π' in ρ, γ, ρ', and τ. If that fails, we fail. Otherwise, if it produces a result (ρ'', τ'), we return $(\rho''[I \mapsto V], \tau')$.
- Case 10: π is of the form (π' **where** E). We then match V against π' in ρ, γ, ρ', and τ. If that fails, we fail. Otherwise, if we get a result (ρ'', τ'), we evaluate the expression E in the lexical environment ρ augmented with the bindings in ρ'', γ, and some appropriate assumption base and state. If that evaluation results in `true`, we return (ρ'', τ'), otherwise we fail.
- Case 11: π is of the form $(\pi_1\ \pi_2 \cdots \pi_{n+1})$, for $n > 0$. As explained in Section 2.11, patterns of this form are used for decomposing terms and sentences. Accordingly, we distinguish the following cases:
  - (i) V is a term t of the form $(f\ t_1 \cdots t_m)$, $m \geq 0$. We then distinguish two subcases:
    1. If $n = 2$, π_1 is a quantifier pattern,[22] and π_2 is a list pattern, then:
       - If t is term of sort `Boolean`, so that it can be treated as a sentence p, try to match it as a sentence against π (in ρ, γ, ρ', and τ), using the algorithm given below (under case (ii)).
       - If t is not a term of sort `Boolean`, fail.
    2. Otherwise, we distinguish the following subcases:
       - $n = 1$ and π_2 is a list pattern. In that case we try to sequentially match the values $f, [t_1 \cdots t_m]$ against the patterns π_1, π_2 (in ρ, γ, ρ', and τ).
       - Otherwise, if $n > 1$ or π_2 is not a list pattern, we try to sequentially match the values $f, t_1, \ldots, t_m$ against the patterns $\pi_1, \pi_2, \ldots, \pi_{n+1}$ (in ρ, γ, ρ', and τ).
  - (ii) V is a sentence p. We then distinguish two subcases, on the basis of the pattern:

22 That is, one of the two pattern constants `forall`, `exists`; or a pattern of the form (**some-quant** J); or else a pattern of the form (**bind** I π), where π is a quantifier pattern.

1. $n = 2$, π_1 is a quantifier pattern, and π_2 is a list pattern. In that case, we first express p in the form

$$(Q\ x_1 \cdots x_k\ p') \tag{7}$$

for some quantifier Q, $k \geq 0$, and a sentence p' that is *not* of the form $(Q\ y_1 \cdots y_m\ p'')$, $m > 0$. Note that this can always be done, for any sentence p, although the identity of Q will not be uniquely determined if $k = 0$, that is, if p is not an actual quantified sentence (a condition that might affect the first of the steps below). After we express p in the form (7), we do the following, in sequential order:

   * If $k = 0$, then if the quantifier pattern π_1 is of the form (`some-quant` I) or (`bind` $I\ \pi'$) for some quantifier pattern π', fail. Otherwise, let $\rho'' = \rho'$ and $\tau' = \tau$, and continue.
   * If $k > 0$ then match Q against the pattern π_1 in ρ, γ, ρ', and τ, and let (ρ'', τ') be the output of that match. (If the match fails, we fail.)
   * Match the largest possible prefix $[x_1 \cdots x_i]$, $i \leq k$, against π_2 in ρ, γ, ρ'' and τ', and let (ρ''', τ'') be the result.
   * Return the result of matching the sentence $(Q\ x_{i+1} \cdots x_k\ p')$ against π_3 in ρ, γ, ρ''', and τ''. (If $k = 0$ then this sentence will simply be p'.)

2. Otherwise we distinguish the following additional subcases, obtained by analyzing the structure of p:

   * If p is an atomic sentence t, we match t as a term against π (in ρ, γ, ρ', and τ).
   * If p is a compound sentence of the form $(\circ\ p_1 \cdots p_k)$, for some sentential connective $\circ$ and $k > 0$, then:

     (a) If $n = 1$ and π_2 is a list pattern, we sequentially match $\circ, [p_1 \cdots p_k]$ against the patterns π_1, π_2 in ρ, γ, ρ', and τ.

     (b) Otherwise, we sequentially match $\circ, p_1, \ldots, p_k$ against the patterns $\pi_1, \pi_2, \ldots, \pi_{n+1}$ (in ρ, γ, ρ', and τ).

   * Finally, if p is a quantified sentence of the form $(Q\ x\ p')$, we sequentially match the values Q, x, p' against the patterns $\pi_1, \ldots, \pi_{n+1}$.

- (iii) If V is neither a term of the form $(f\ t_1 \cdots t_m)$ nor a sentence, we fail.

- Case 12: π is a filter pattern, that is, of the form (`some-`$\cdots$ J). Suppose first that it is of the form (a) (`some-var` I) or (b) (`some-var` _). In that case we check to see whether V is some term variable. If it is not, we fail. If it is, then, in case (b), we return ρ' and τ unchanged. In case (a), we return $\rho'[I \mapsto V]$ and τ. When π is of the form (a) (`some-atom` I) or (b) (`some-atom` _), we check to see whether V is an atomic sentence

(i.e., a term of sort Boolean), and if so, we return $(\rho'[I \mapsto V], \tau)$ in case (a) and (ρ', τ) in case (b). Likewise for the remaining filter patterns. For instance, if π is of the form (a) (**some-sent-con** I) or (b) (**some-sent-con** _), we check whether V is one of the five sentential connectives, and if so, we return $(\rho'[I \mapsto V], \tau)$ in case (a) and (ρ', τ) in case (b).

Finally, to *sequentially match* a number of values $V_1, \ldots, V_n$ against a number of patterns $\pi_1, \ldots, \pi_m$ in given ρ, γ, ρ', and τ, we do the following: First, if $n \neq m$, we fail. Otherwise, if $n = 0$, we return (ρ', τ). Finally, if $n = m$ and $n > 0$, we match V_1 against π_1 in ρ, γ, ρ', and τ, resulting in some output pair (ρ'_1, τ_1), and then we proceed to sequentially match (recursively) $V_2, \ldots, V_n$ against $\pi_2, \ldots, \pi_n$ in ρ, γ, ρ'_1, and τ_1.

A.5 Selectors

A constructor profile $(c\ S_1 \cdots S_n)$ in the definition of a datatype or structure can optionally have one or more *selectors* attached to various argument positions. To introduce a selector by the name of g for the i^{th} argument position of c, simply write the profile as: $(c\ S_1 \cdots g{:}S_i \cdots S_n)$. For instance, for the natural numbers we could have:

```
datatype N := zero | (succ pred:N)
```

This introduces the selector pred as a function from natural numbers to natural numbers such that (pred succ n = n) for all n:N. It is generally a good idea to introduce selectors.[23] When a constructor has arity greater than 1, we can introduce a selector for every argument position. For instance, in the definition of natural-number lists below, we introduce head as a selector for the first argument position of nat-cons and tail as a selector for the second argument position:

```
datatype Nat-List := nat-nil | (nat-cons head:N tail:Nat-List)
```

Axioms specifying the semantics of selectors are automatically generated and can be obtained by applying the procedure selector-axioms to the name of the datatype:

```
> (selector-axioms "Nat-List")

List: [
(forall ?v561:N
  (forall ?v562:Nat-List
    (= (head (nat-cons ?v561 ?v562))
       ?v561)))

(forall ?v561:N
```

23 And if the datatype is to be used in SMT solving (Section D.2), then every constructor *must* have a selector attached to every argument position.

```
  (forall ?v562:Nat-List
    (= (tail (nat-cons ?v561 ?v562))
       ?v562)))
]
```

The behavior of `head` or `tail` on `nat-nil` is unspecified. In general, the behavior of any selector of a constructor c when applied to a term built by a different constructor c' is unspecified.

However, it is possible to provide a total specification for selectors by instructing Athena to treat selectors as functions from the datatype at hand to *optional* values of the corresponding sorts. This is done by turning on the flag `option-valued-selectors`, which is off by default. For instance:

```
set-flag option-valued-selectors "on"

datatype Nat-List-2 := nat-nil-2 | (nat-cons-2 head-2:N tail-2:Nat-List-2);;

> (selector-axioms "Nat-List-2")

List: [
(forall ?v617:N
  (forall ?v618:Nat-List-2
    (= (head-2 (nat-cons-2 ?v617 ?v618))
       (SOME ?v617))))

(forall ?v617:N
  (forall ?v618:Nat-List-2
    (= (tail-2 (nat-cons-2 ?v617 ?v618))
       (SOME ?v618))))

(= (head-2 nat-nil-2)
   NONE)

(= (tail-2 nat-nil-2)
   NONE)
]
```

A.6 Prefix syntax

In this book we have used mostly infix syntax, both for issuing directives (such as **`declare`** and **`define`**) and for writing phrases. However, Athena also supports a fully prefix syntax based on s-expressions. An advantage of s-expressions is that they lend themselves exceptionally well to indentation, which can serve to clarify the structure of complex syntactic constructions. (That is why Athena always displays output in fully indented prefix.) Below

we present s-expression variants of all major syntax forms (shown on the left side in the infix form in which they have been used in this book):

declare I: [$S_1 \cdots S_n$] -> S	(**declare** I (-> ($S_1 \cdots S_n$) S))
declare I: ($I_1, \ldots, I_k$) [$S_1 \cdots S_n$] -> S	(**declare** I (($I_1 \cdots I_k$) -> ($S_1 \cdots S_n$) S))
declare $I_1, \ldots, I_m$: ($I'_1, \ldots, I'_k$) [$S_1 \cdots S_n$] -> S	(**declare** ($I_1 \cdots I_m$) (($I'_1 \cdots I'_k$) -> ($S_1 \cdots S_n$) S))
assert $E_1, \ldots, E_n$	(**assert** $E_1 \cdots E_n$)
assert I := E	(**assert** I := E)
set-flag I "on"/"off"	(**set-flag** I "on"/"off")
overload I_1 I_2	(**overload** I_1 I_2)
set-precedence I E	(**set-precedence** I E)
set-precedence ($I_1 \cdots I_n$) E	(**set-precedence** ($I_1 \cdots I_n$) E)
lambda ($I_1 \cdots I_n$) E	(**lambda** ($I_1 \cdots I_n$) E)
method ($I_1 \cdots I_n$) D	(**method** ($I_1 \cdots I_n$) D)
cell F	(**cell** F)
set! E F	(**set!** E F)
ref E	(**ref** E)
while F E	(**while** F E)
make-vector E F	(**make-vector** E F)
vector-sub E_1 E_2	(**vector-sub** E_1 E_2)
vector-set! E_1 E_2 F	(**vector-set!** E_1 E_2 F)
check {F_1 => E_1 | $\cdots$ | F_n => E_n}	(**check** (F_1 E_1) $\cdots$ (F_n E_n))
match F {π_1 => E_1 | $\cdots$ | π_n => E_n}	(**match** F (π_1 E_1) $\cdots$ (π_n E_n))
let {π_1 := F_1; $\cdots$; π_n := F_n} E	(**let** ((π_1 F_1) $\cdots$ (π_n F_n)) E)
letrec {I_1 := E_1; $\cdots$; I_n := E_n} E	(**letrec** ((I_1 E_1) $\cdots$ (I_n E_n)) E)
try {E_1 | $\cdots$ | E_n}	(**try** E_1 $\cdots$ E_n)
check {F_1 => D_1 | $\cdots$ | F_n => D_n}	(**dcheck** (F_1 D_1) $\cdots$ (F_n D_n))
match F {π_1 => D_1 | $\cdots$ | π_n => D_n}	(**dmatch** F (π_1 D_1) $\cdots$ (π_n D_n))
let {π_1 := F_1; $\cdots$; π_n := F_n} D	(**dlet** ((π_1 F_1) $\cdots$ (π_n F_n)) D)
letrec {I_1 := E_1; $\cdots$; I_n := E_n} D	(**dletrec** ((I_1 E_1) $\cdots$ (I_n E_n)) D)
try {D_1 | $\cdots$ | D_n}	(**dtry** D_1 $\cdots$ D_n)