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:

\[ F ::= E | D. \]

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:

\[
\begin{align*}
C \ ::= & \quad \text{‘printable} \quad \text{(e.g., ‘A)} \\
& \quad \text{‘\code} \quad \text{(e.g., ‘\73)} \\
& \quad \text{‘\control-character} \quad \text{(e.g., ‘\c)} \\
& \quad \text{‘\escape-character} \quad \text{(e.g., ‘\n)} \\
escape-character \ ::= & \quad " | \ \text{a | b | n | r | t | v} \\
control-character \ ::= & \quad A | \cdots | Z | @ | [ | ] | | | \_ | ‘\blank \\
\end{align*}
\]

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:

\[
T ::= "C*" \quad \text{(e.g., "nHello world!")}
\]

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

**Figure A.1**
Syntax of Athena expressions

\[
S ::= 'I \quad \text{(Sort variables, e.g., ’S5)}
\]
| I \quad \text{(Constant sort constructors, e.g., Boolean or Int)} |
| (I S_1 \cdots S_n) \quad \text{(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:

\[
\pi ::= I \mid I:S \mid (I \pi^+) \mid (I \text{ as } \pi)
\]

where \( S \) ranges over sorts.

### A.2 Values

Figure A.3 shows the types of values that Athena phrases denote. Some brief remarks on each type follow.
A.2. VALUES

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 are discussed in Section 2.2; they are datatype or structure constructors, or else symbols introduced with 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 for all 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 \([V_1 \cdots V_n]\), \(n \geq 0\), possibly empty.

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 \([V_1 \cdots V_m]\), \(m \geq 0\); (2) an assumption base \(\beta\); and (3) a store \(\sigma\); and returns an ordered pair \((V, \sigma')\) consisting of a value \(V\) and a store \(\sigma'\), 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.

---

<table>
<thead>
<tr>
<th>(D ) ::=</th>
<th>Method calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>((apply-method \ E \ F_1 \cdots F_n))</td>
<td>Method calls (usual syntax)</td>
</tr>
<tr>
<td>!!(\ E \ F_1 \cdots F_n)!!</td>
<td>Conclusion-annotated deductions</td>
</tr>
<tr>
<td>conclude ( F \ D )</td>
<td>Hypothetical deductions</td>
</tr>
<tr>
<td>assume ( I := F \ D )</td>
<td>Named hypothetical deductions</td>
</tr>
<tr>
<td>assume ( I_1 := F_1; \cdots; I_{n+1} := F_{n+1} \ D )</td>
<td>Conjunctive named hypothetical deductions</td>
</tr>
<tr>
<td>suppose-absurd ( F \ D )</td>
<td>Proofs by contradiction</td>
</tr>
<tr>
<td>generalize-over ( E \ D )</td>
<td>Universal generalizations</td>
</tr>
<tr>
<td>pick-any ( I \ D )</td>
<td>Universal generalizations</td>
</tr>
<tr>
<td>pick-any ( I : S \ D )</td>
<td>Universal generalizations</td>
</tr>
<tr>
<td>with-witness ( E \ F \ D )</td>
<td>Existential instantiations</td>
</tr>
<tr>
<td>pick-witness ( I ) for ( F \ D )</td>
<td>Existential instantiations</td>
</tr>
<tr>
<td>pick-witness ( I ) for ( F \ D )</td>
<td>Existential instantiations</td>
</tr>
<tr>
<td>pick-witnesses ( I_1, I_2, \cdots, I_q ) for ( F \ D )</td>
<td>Existential instantiations</td>
</tr>
<tr>
<td>by-induction ( F \ [\pi_1 \Rightarrow D_1 \mid \cdots \mid \pi_n \Rightarrow D_n] )</td>
<td>Structural induction</td>
</tr>
<tr>
<td>datatype-cases ( F \ [\pi_1 \Rightarrow D_1 \mid \cdots \mid \pi_n \Rightarrow D_n] )</td>
<td>Structural case analysis</td>
</tr>
<tr>
<td>check ( {F_1 \Rightarrow D_1 \mid \cdots \mid F_n \Rightarrow D_n} )</td>
<td>check deductions</td>
</tr>
<tr>
<td>match ( F \ [\pi_1 \Rightarrow D_1 \mid \cdots \mid \pi_n \Rightarrow D_n] )</td>
<td>match deductions</td>
</tr>
<tr>
<td>let ( {\pi_1 : V_1; \cdots; \pi_n : V_n} ) ( \Rightarrow D )</td>
<td>let deductions</td>
</tr>
<tr>
<td>letrec ( {\pi_1 := E_1; \cdots; \pi_n := E_n} ) ( \Rightarrow D )</td>
<td>letrec deductions</td>
</tr>
<tr>
<td>try ( {D_1 \mid \cdots \mid D_n} )</td>
<td>try deductions</td>
</tr>
</tbody>
</table>

Figure A.2
Syntax of Athena deductions
<table>
<thead>
<tr>
<th>Terms</th>
<th>Function symbols</th>
<th>Sentential connectives and quantifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Substitutions</td>
<td>Sentences</td>
<td>Lists of values</td>
</tr>
<tr>
<td>Cells</td>
<td>Function values</td>
<td>Method values</td>
</tr>
<tr>
<td>Tables</td>
<td>Vectors</td>
<td>Characters</td>
</tr>
</tbody>
</table>

Figure A.3
Types of Athena values

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 \([V_1 \cdots V_m]\), \(m \geq 0\); (2) an assumption base \(\beta\); and (3) a store \(\sigma\); and returns an ordered pair \((p, \sigma')\) consisting of a sentence \(p\) and a store \(\sigma'\) (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 are 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 sequences 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 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 \texttt{equal?} to such values. Equality for characters is obvious: Two characters are the same iff they have the same code. Two lists of values \([V_1 \cdots V_n]\) and \([V_1' \cdots V_m']\) 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 \( \rho \), assumption base \( \beta \), store \( \sigma \), and symbol set \( \gamma \). We clarify these parameters below:

1. An environment \( \rho \) 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 \( \beta \) is a finite set of sentences.
3. A store \( \sigma \) 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 \( \gamma \) is a collection of function symbols and their respective signatures\(^\text{2}\) along with a collection of sort constructors and their arities; \( \gamma \) also includes information 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 \( \rho, \beta, \sigma, \) and \( \gamma \), is one of three things:

1. a pair \((V, \sigma')\) consisting of a value \( V \) and a store \( \sigma' \), where \( V \) is the output of the evaluation and \( \sigma' \) reflects any side effects accumulated during the evaluation; or

\(^2\) See page 24 for a discussion of signatures.
2. a pair consisting of an error message and a store $\sigma'$, indicating the occurrence of an error during the computation (a store $\sigma'$ 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: $\sigma' = \sigma$. 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 $\sigma'$; in such cases the result of evaluating $F$ becomes that same error message and $\sigma'$.

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 $\rho$ to some value $V$, then the output value is $V$; it is an error if $I$ is unbound in $\rho$.
- **Unit**: The value of the expression () is always the unit value.
- **Term variables, meta-identifiers, and characters** are always self-evaluating, regardless of $\rho, \beta, \sigma$, and $\gamma$.
- **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

\[
\text{check} \{ F_1 \Rightarrow E_1 | \cdots | F_n \Rightarrow E_n \}
\]

in $\rho, \beta, \sigma$, and $\gamma$, we first evaluate $F_1$ in $\rho, \beta, \sigma$, and $\gamma$, possibly resulting in a value $V_1$ and store $\sigma_1$. 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 $\rho, \beta, \sigma_1$, and $\gamma$. Otherwise, if $V_1$ is

---

3. 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.

4. In accordance with the convention made above, if the evaluation of $F_1$ in $\rho, \beta, \sigma$, and $\gamma$ generates an error and some store $\sigma_1$, we simply return that as the result of the entire check expression.
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 $\rho, \beta, \sigma_1$, and $\gamma$, possibly resulting in a value $V_2$ and a store $\sigma_2$. If $V_2$ is true, then the final result is that of evaluating $E_2$ in $\rho, \beta, \sigma_2$, and $\gamma$. Otherwise, if $V_2$ is false, $F_3$ is evaluated in $\rho, \beta, \sigma_2$, and $\gamma$; its value is compared to true, and so forth. Clearly, evaluation might diverge if some $F_1$ 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 $\sigma$ 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 $\sigma_n$; 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 $\sigma_i$. 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 $\rho, \beta, \sigma_{n-1}$, and $\gamma$.

- **Procedures**: The value of lambda $(I_1 \cdots I_n)$ $E$ in $\rho, \beta, \sigma$, and $\gamma$ is a function value that takes a list of $n$ values $V_1, \ldots, V_n$ along with an assumption base $\beta'$ and a store $\sigma'$ as arguments, and produces the result of evaluating the body $E$ in $\rho[I_1 \mapsto V_1, \ldots, I_n \mapsto V_n], \beta', \sigma'$, and $\gamma$. Note that the 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 $\rho, \beta, \sigma$, and $\gamma$ is a method value that takes a list of $n$ values $V_1, \ldots, V_n$ along with an assumption base $\beta'$ and store $\sigma'$ as arguments, and produces the result of evaluating the deduction $D$ in $\rho[I_1 \mapsto V_1, \ldots, I_n \mapsto V_n], \beta', \sigma'$, and $\gamma$. Here too, the 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 $\rho, \beta, \sigma$, and $\gamma$ is obtained as follows. First, $E$ is evaluated in $\rho, \beta, \sigma$, and $\gamma$, possibly resulting in a value $V$ and store $\sigma'$. We then proceed with a case analysis of $V$:

1. If $V$ is a function value $\phi$, we evaluate the arguments $F_1, \ldots, F_n$ sequentially, starting with $F_1$ in $\rho, \beta, \sigma'$, and $\gamma$, to obtain values $V_1, \ldots, V_n$ and a store $\sigma_n$. This sequential evaluation threads the store from the evaluation of $F_i$ to that of $F_{i+1}$. Specifically, first we evaluate $F_1$ in $\rho, \beta, \sigma'$, and $\gamma$ to obtain a value $V_1$ and a store $\sigma_1$; then we evaluate $F_2$ in $\rho, \beta, \sigma_1$, and $\gamma$ to obtain a value $V_2$ and store $\sigma_2$; we continue in this fashion until we evaluate $F_n$ in $\rho, \beta, \sigma_{n-1}$, and $\gamma$, resulting in a value $V_n$ and store $\sigma_n$. The final result is that of applying $\phi$ to the list $[V_1 \cdots V_n], \beta$, and $\sigma_n$.

---

5 The keyword else may only appear in the position of $F_n$. 
2. If $V$ is a function symbol $f$ of arity $n$, we sequentially evaluate $F_1, \ldots, F_n$, starting with $F_1$ in $\rho$, $\beta$, $\sigma$ and $\gamma$, to obtain values $V_1, \ldots, V_n$ and a store $\sigma_n$. 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 $\sigma_n$ as the result. (We also return an error message and the store $\sigma_n$ 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 $\sigma_n$. 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 $\sigma_n$. If the sentence is not well sorted or if some $V_i$ is not a sentence, we return an error message and the store $\sigma_n$.

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 $\sigma'$. We then evaluate $F_1, \ldots, F_n$ sequentially to obtain values $V_1, \ldots, V_n$ and a store $\sigma_n$. 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 $\sigma_n$; otherwise the resulting value is the quantified sentence:

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

provided that it is well sorted, while the resulting store is $\sigma_n$. If the above is not a well-sorted sentence, we return an error message and the store $\sigma_n$.6

5. If $V$ is a substitution $\theta$, we check whether $n = 1$. If not, we return an error message along with the store $\sigma'$ and halt. Otherwise we evaluate $F_1$ in $\rho$, $\beta$, $\sigma'$, and $\gamma$ to obtain a value $V_1$ and a store $\sigma_1$. 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 $\sigma_1$, provided that $\theta(t_1)$ (respectively, $\theta(p_1)$) is well sorted; if it is not, we return an error message and $\sigma_1$.7
- 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 $[\theta(V_1) \cdots \theta(V_n)]$ and the store $\sigma_1$, provided that each $\theta(V_i)$ is well sorted. If one is not, or if $V_1$ is not such a list, we return an error message and the store $\sigma_1$.
- If $V_1$ is neither a term (or sentence) nor a list of terms (and/or sentences), we halt with an error message and the store $\sigma_1$.

6. If $V$ is a map, we check whether $n = 1$. If not, like before, we return an error message along with the store $\sigma'$ and halt. Otherwise we evaluate $F_1$ in $\rho$, $\beta$, $\sigma'$, and $\gamma$ to obtain a value $V_1$ and a store $\sigma_1$. 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 $\sigma_1$. If $V_1$ is not a key in that map, we return an error message and $\sigma_1$.

---

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

7 We write $\theta(t)$ (respectively, $\theta(p)$) for the term obtained by applying the substitution $\theta$ to the term $t$ (respectively, sentence $p$).
A.3. OPERATIONAL SEMANTICS

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 \( \sigma' \).

- **match expressions**: To evaluate \( \text{match } F \{ \pi_1 \Rightarrow E_1 | \cdots | \pi_n \Rightarrow E_n \} \) in \( \rho, \beta, \sigma, \) and \( \gamma \), we first evaluate the discriminant \( F \) in \( \rho, \beta, \sigma, \) and \( \gamma, \) possibly obtaining a value \( V \) and store \( \sigma' \). We then go through the patterns \( \pi_1, \ldots, \pi_n \) sequentially, trying to find a pattern that is matched by \( V \) in \( \rho \) and \( \gamma \) (using the algorithm of Section A.4). If no such pattern is found, we halt with an error message and \( \sigma' \). Otherwise, let \( \pi_j \) be the first matching pattern and let \( \{ I_1 \mapsto V_1, \ldots, I_k \mapsto V_k \} \) and \( \tau \) 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 \( \rho[I_1 \mapsto V_1, \ldots, I_k \mapsto V_k], \beta, \sigma', \) and \( \gamma \). If that result is a term or sentence, then \( \tau \) is applied to it before returning.

- **let expressions**: For semantic purposes, an expression of the form

\[
\text{let } \{ \pi_1 := F_1; \cdots; \pi_n := F_n \} \ E
\]

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 \( \text{match } F_1 \{ \pi_1 \Rightarrow E' \} \), where \( E' \) is the result of desugaring

\[
\text{let } \{ \pi_2 := F_2; \cdots; \pi_n := F_n \} \ E.
\]

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

\[
\text{letrec } \{ I_1 := E_1; \cdots; I_n := E_n \} \ E
\]

in \( \rho, \beta, \sigma, \) and \( \gamma \) is that of evaluating the following expression in \( \rho, \beta, \sigma, \) and \( \gamma : \)

\[
\text{let } \{ I_1 := \text{cell} () ; \cdots; I_n := \text{cell} () ; \},
- := \text{set! } I_1 \ E_1' ; 
\cdots ;
- := \text{set! } I_n \ E_n' \}
\]

where each \( E_j' \) is obtained from \( E_j \) by replacing every free occurrence of \( I_j^8 \) by \( \text{ref } I_j, \) \( j = 1, \ldots, n ; \) 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

\[
\text{try } \{ E_1 | \cdots | E_n \}
\]

in \( \rho, \beta, \sigma, \) and \( \gamma \), we first evaluate \( E_1 \) in \( \rho, \beta, \sigma, \) and \( \gamma \). If that results in a value \( V_1 \) and store \( \sigma_1 \), we return \( V_1 \) and \( \sigma_1 \). But if it results in an error message and a store \( \sigma_1 \), we go

---

8 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.
on to evaluate $E_2$ in $\rho$, $\beta$, $\sigma_1$, and $\gamma$; and so forth. If the evaluation of every $E_i$ fails, we return an error message and $\sigma_n$.

- **Cells**: To evaluate an expression of the form `cell` $F$ in $\rho$, $\beta$, $\sigma$, and $\gamma$, we first evaluate $F$ in $\rho$, $\beta$, $\sigma$, and $\gamma$ to obtain some value $V$ and store $\sigma'$. 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 $\rho$, $\beta$, $\sigma$, and $\gamma$, we evaluate $E$ in $\rho$, $\beta$, $\sigma$, and $\gamma$, possibly obtaining a value $V$ and store $\sigma'$. If the value $V$ is not a memory location (cell), we return an error message and $\sigma'$. If it is a memory location $l$, we return the value $\sigma'(l)$ and the store $\sigma'$.

- **Assignments**: To evaluate an expression of the form `set`! $E$ $F$ in $\rho$, $\beta$, $\sigma$, and $\gamma$, we first evaluate $E$ in $\rho$, $\beta$, $\sigma$, and $\gamma$, possibly obtaining a value $V_1$ and store $\sigma_1$. If $V_1$ is not a memory location (cell), we halt with an error message and $\sigma_1$ as the result. If it is a cell $l$, we proceed to evaluate $F$ in $\rho$, $\beta$, $\sigma_1$, and $\gamma$, possibly obtaining a value $V_2$ and store $\sigma_2$. 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 $\rho$, $\beta$, $\sigma$, and $\gamma$:
  1. We evaluate $F$ in $\rho$, $\beta$, $\sigma$, and $\gamma$, possibly obtaining a value $V_1$ and store $\sigma_1$.
  2. If $V_1$ is the constant term `false`, we return the unit value and store $\sigma_1$. Otherwise, if $V_1$ is `true`, we evaluate $E$ in $\rho$, $\beta$, $\sigma_1$, and $\gamma$ to obtain some value $V_2$ and store $\sigma_2$. We then continue with the first step, only now $F$ is evaluated in $\rho$, $\beta$, $\sigma_2$, and $\gamma$ (rather than $\rho$, $\beta$, $\sigma$, and $\gamma$). If $V_1$ is neither `true` nor `false`, we return an error message and $\sigma_1$.

- **Short-circuit Boolean operations**: Expressions of the 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 $\rho$, $\beta$, $\sigma$, and $\gamma$ 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 $\rho$, $\beta$, $\sigma$, and $\gamma$, we start by evaluating $E$ in $\rho$, $\beta$, $\sigma$, and $\gamma$ to obtain a value $V_1$ and a store $\sigma_1$. If $V_1$ is not a nonnegative integer constant, we halt with an error message and $\sigma_1$. If it is a nonnegative integer constant $n$, we proceed to evaluate $F$ in $\rho$, $\beta$, $\sigma_1$, and $\gamma$, possibly obtaining a value $V_2$ and store $\sigma_2$. 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 $\rho$, $\beta$, $\sigma$, and $\gamma$, we first evaluate $E_1$ in $\rho$, $\beta$, $\sigma$, and $\gamma$, obtaining a value $V_1$ and store $\sigma_1$. If $V_1$ is not a list

---

9 It is an error if cell $l$ is unassigned, although this could never happen in our semantics.
of memory locations previously created by \texttt{make-vector}.\footnote{Strictly speaking this requires that we tag lists of cells created by \texttt{make-vector} to distinguish them from lists of cells that may have been created by other means, but this need not detain us here.} we output an error message and \(\sigma_1\). If \(V_1\) is a list of memory locations \([l_1 \cdots l_n]\) previously created by \texttt{make-vector}, \(n \geq 0\), we proceed to evaluate \(E_2\) in \(\rho, \beta, \sigma_1,\) and \(\gamma\), obtaining a value \(V_2\) and store \(\sigma_2\). If \(V_2\) is not a nonnegative integer constant, we halt with an error message and \(\sigma_2\). If it is a nonnegative integer constant \(i\), we return the value \(\sigma_2(l_{i+1})\) and \(\sigma_2\), provided that the index \(i\) is between 0 and \(n - 1\) (if it is not, we halt with an error message and \(\sigma_2\)).

- **Vector assignment:** To evaluate an expression \texttt{vector-set!} \(E_1\ E_2\ F\) in \(\rho, \beta, \sigma,\) and \(\gamma\), we first evaluate \(E_1\) in \(\rho, \beta, \sigma,\) and \(\gamma\), obtaining from it some value \(V_1\) and store \(\sigma_1\). If \(V_1\) is not a list of memory locations previously created by \texttt{make-vector}, we output an error message and \(\sigma_1\). If \(V_1\) is a list of memory locations \([l_1 \cdots l_n]\) previously created by \texttt{make-vector}, \(n \geq 0\), we proceed to evaluate \(E_2\) in \(\rho, \beta, \sigma_1,\) and \(\gamma\), obtaining a value \(V_2\) and store \(\sigma_2\). If \(V_2\) is not a nonnegative integer constant between 0 and \(n - 1\), we halt with an error message and \(\sigma_2\). Otherwise, if it is a some such constant \(i\), we proceed to evaluate \(F\) in \(\rho, \beta, \sigma_2,\) and \(\gamma\), obtaining from it some value \(V_3\) and store \(\sigma_3\). 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 \texttt{(=} \(E\ F_1 \cdots F_n\)) in \(\rho, \beta, \sigma,\) and \(\gamma\), we first evaluate \(E\) in \(\rho, \beta, \sigma,\) and \(\gamma\), obtaining a value \(V\) and store \(\sigma'\) from it. If \(V\) is not a method value, we halt with an error message and \(\sigma'\). 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 \(\rho, \beta, \sigma_{i-1},\) and \(\gamma\), obtaining from it a value \(V_i\) and store \(\sigma_i\). 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 \(\beta'\). When all arguments have been evaluated, we apply \(M\) to the list of values \([V_1 \cdots V_n]\), \(\beta \cup \beta'\), and \(\sigma_n\). 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 \texttt{conclude} \(F\ D\) in \(\rho, \beta, \sigma,\) and \(\gamma\), we first evaluate \(F\) in \(\rho, \beta, \sigma,\) and \(\gamma\), to obtain a value \(V\) and a store \(\sigma'\). If \(V\) is not a sentence, we halt with an error message and \(\sigma'\). Otherwise, if \(V\) is a sentence \(p\), we proceed to evaluate the body \(D\) in \(\rho, \beta, \sigma',\) and \(\gamma\), obtaining from it a conclusion \(q\) and store \(\sigma''\). If \(p\) and \(q\) are alpha-equivalent, we halt with \(p\) and \(\sigma''\) as the output; otherwise we halt with an appropriate error message and \(\sigma''\).

- **Hypothetical deductions:** To evaluate \texttt{assume} \(F\ D\) in \(\rho, \beta, \sigma,\) and \(\gamma\), we first evaluate \(F\) in \(\rho, \beta, \sigma,\) and \(\gamma\), producing a value \(V\) and store \(\sigma'\). If \(V\) is not a sentence, we halt with an error message and \(\sigma'\). Otherwise, if \(V\) is a sentence \(p\), we proceed to evaluate the body \(D\)
in \( \rho, \beta \cup \{p\}, \sigma', \) and \( \gamma, \) obtaining from it some sentence \( q \) and store \( \sigma'' \). We then return the conditional \( (p \implies q) \) and \( \sigma'' \) as the result.

**Named hypothetical deductions**: A deduction \( \text{assume } I := F \ D \) is treated as syntax sugar for the following:

\[
\begin{align*}
\text{let } & \{I := F\} \\
\text{assume } & I \\
D
\end{align*}
\]

**Conjunctive named hypothetical deductions**: A deduction of the form

\[
\text{assume } I_1 := F_1; \ldots ; I_{n+1} := F_{n+1} \ D
\]

for \( n > 0 \) is treated as syntax sugar for the following:

\[
\begin{align*}
\text{let } & \{I_1 := F_1; \ldots ; I_{n+1} := F_{n+1}\} \\
\text{assume } & (I_1 & I_2 & \cdots & I_{n+1}) \\
D
\end{align*}
\]

- **Proof by contradiction**: To evaluate \( \text{suppose-absurd } F \ D \) in \( \rho, \beta, \sigma, \) and \( \gamma, \) we begin by evaluating \( F \) in \( \rho, \beta, \sigma, \) and \( \gamma, \) producing a value \( V \) and store \( \sigma'. \) If \( V \) is not a sentence, we halt with an error message and \( \sigma''. \) If \( V \) is a sentence \( p, \) we evaluate the body \( D \) in \( \rho, \beta \cup \{p\}, \sigma', \) and \( \gamma, \) obtaining from it some sentence \( q \) and store \( \sigma'''. \) If \( q \) is the constant \( false, \) we return \( (\sim p) \) and \( \sigma''' \) as the result; if \( q \) is not \( false, \) we output an error message and \( \sigma'''. \)

- **Universal generalizations**: To evaluate a deduction \( \text{generalize-over } E \ D \) in \( \rho, \beta, \sigma, \) and \( \gamma, \) we first evaluate \( E \) in \( \rho, \beta, \sigma, \) and \( \gamma, \) obtaining some value \( V \) and store \( \sigma'. \) If \( V \) is a variable \( x \) (say, \( ?p:Person \)), we first check to see that \( x \) does not occur free in \( \beta. \) \(^{11}\) If it does, we halt with an error message and \( \sigma'. \) If \( x \) does not occur free in \( \beta, \) we proceed to evaluate the body \( D \) in \( \rho, \beta, \sigma', \) and \( \gamma, \) obtaining a conclusion \( p \) and store \( \sigma'''. \) We then return the sentence \( (\forall x \ p) \) and \( \sigma''', \) provided that \( (\forall x \ p) \) is well sorted (if it is not, we output an error message and \( \sigma'''). \)

- **pick-any universal generalizations**: A deduction of the form \( \text{pick-any } I \ D \) is treated as syntax sugar for:

\[
\begin{align*}
\text{let } & \{I := (\text{fresh-var})\} \\
\text{generalize-over } & I \\
D
\end{align*}
\]

---

\(^{11}\) A variable \( ?I:S \) is said to occur free in an assumption base \( \beta \) 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 \( \beta = \{?p:?'S3\} \), then \( ?z:Person \) is considered to occur free in \( \beta, \) as \( Person \) is an instance of \( 'S3. \)
A.3. OPERATIONAL SEMANTICS

And a deduction of the form \texttt{pick-any} \( I : S \ D \) is treated as syntax sugar for

\begin{verbatim}
let \{ I := (fresh-var "S") \}
generalize-over I
D
\end{verbatim}

- \textbf{with-witness deductions}: To evaluate \texttt{with-witness} \( E \ F \ D \) in \( \rho, \beta, \sigma, \) and \( \gamma \), we first evaluate \( E \) in \( \rho, \beta, \sigma, \) and \( \gamma \), to obtain a value \( V \) and store \( \sigma' \). If \( V \) is not an Athena variable, or if it is a variable that occurs free in \( \beta \), we halt with an error message and \( \sigma' \). Otherwise, if \( V \) is a variable \( w \) that does not occur free in \( \beta \), we proceed to evaluate \( F \) in \( \rho, \beta, \sigma', \) and \( \gamma \), obtaining a value \( V' \) and store \( \sigma'' \). If \( V' \) is not an existential quantification, we halt with an error message and \( \sigma'' \). 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 \( \sigma'' \)). We then evaluate the deduction \( D \) in \( \rho, \beta' \cup \{ q' \}, \sigma'' \), and \( \gamma \), obtaining some sentence \( r \) and store \( \hat{\sigma} \). If the variable \( w \) occurs free in \( r \), we halt with an error message and \( \hat{\sigma} \), otherwise we return \( r \) and \( \hat{\sigma} \).

- \textbf{pick-witness deductions}: A deduction \texttt{pick-witness} \( I \) for \( F \ D \) is treated as syntax sugar for

\begin{verbatim}
let \{ I := (fresh-var) \}
with-witness I F D
\end{verbatim}

A deduction \texttt{pick-witness} \( I_1 \) for \( F \ I_2 \ D \) is treated as syntax sugar for

\begin{verbatim}
let \{ I_1 := (fresh-var); equant := F \}
match equant {
(exists (some-var v) body) =>
let \{ I_2 := (replace-var v I_1 body) \}
with-witness I_1 equant D
}
\end{verbatim}

(Recall that \( (\text{replace-var} \ v \ t \ p) \) 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.)

- \textbf{check deductions}: A deduction \texttt{check} \( \{ F_1 \Rightarrow D_1 \ | \ \cdots \ | \ F_n \Rightarrow D_n \} \) is evaluated by the same algorithm used for \texttt{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 \( \text{match } F \{ \pi_1 \Rightarrow D_1 \mid \cdots \mid \pi_n \Rightarrow D_n \} \) in \( \rho, \beta, \sigma, \text{ and } \gamma \), we first evaluate the discriminant \( F \) in \( \rho, \beta, \sigma, \text{ and } \gamma \), obtaining a value \( V \) and store \( \sigma' \). 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 \( \rho \) and \( \gamma \). If no such pattern is found, we halt with an error message and \( \sigma' \). Otherwise, let \( \pi_j \) be the first matching pattern and let \( \{ I_1 \mapsto V_1, \ldots, I_k \mapsto V_k \} \) and \( \tau \) be the set of bindings and sort valuation returned by the matching algorithm, respectively. The output then becomes the result of applying \( \tau \) to the conclusion obtained by evaluating \( D_j \) in \( \rho[I_1 \mapsto V_1, \ldots, I_k \mapsto V_k], \beta', \sigma', \text{ and } \gamma \). 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

\[
\text{let } \{ \pi_1 := F_1; \cdots; \pi_n := F_n \} \ D
\]

(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 \( \text{match } F_1 \{ \pi_1 \Rightarrow D' \} \), where \( D' \) is the result of desugaring \( \text{let } \{ \pi_2 := F_2; \cdots; \pi_n := F_n \} \ D \).

• letrec deductions: A deduction of the form \( \text{letrec } \{ I_1 := E_1; \cdots; I_n := E_n \} \ D \) is desugared into the following:

\[
\text{let } \{ I_1 := (\text{cell }()); \cdots I_n := (\text{cell }()); \ _ := (\text{set}! I_1 E_1'); \cdots \ _ := (\text{set}! I_n E_n'); \ D' \\
\]

where each \( E_j' \) is obtained from \( E_j \) by replacing every free occurrence of \( I_j \) by \( \text{(ref } I_j) \), \( j = 1, \ldots, n \); and \( D' \) is likewise obtained from \( D \).

• try deductions: To evaluate \( \text{try } \{ D_1 | \cdots | D_n \} \) in \( \rho, \beta, \sigma, \text{ and } \gamma \), we first evaluate \( D_1 \) in \( \rho, \beta, \sigma, \text{ and } \gamma \). If that results in a sentence \( p \) and store \( \sigma_1 \), we return \( p \) and \( \sigma_1 \) as the result. But if it results in an error message and a store \( \sigma_1 \), we go on to evaluate \( D_2 \) in \( \rho, \beta, \sigma_1, \text{ and } \gamma \); and so forth. If the evaluation of every \( D_i \) fails, we return an error message and \( \sigma_n \).

• Structural induction: To evaluate a deduction of the form

\[
\text{by-induction } F \{ \pi_1 \Rightarrow D_1 | \cdots | \pi_n \Rightarrow D_n \}
\]

(3)

in \( \rho, \beta, \sigma, \text{ and } \gamma \), we start by evaluating \( F \) in \( \rho, \beta, \sigma, \text{ and } \gamma \), obtaining a value \( V \) and store \( \sigma_1 \). If \( V \) is not a universally quantified sentence of the form

\[
(\forall I : S_D \ p)
\]

(4)
A.3. OPERATIONAL SEMANTICS

for some inductively generated sort \( S_D \) in \( \gamma \), we halt with an error message and \( \sigma_1 \). Otherwise, we check to make sure that the patterns \( \pi_1, \ldots, \pi_n \) are proper patterns for \( S_D \) (in accordance with \( \gamma \)) and that they are jointly exhaustive. We will now explain in detail what it means for the patterns to be “proper” for \( S_D \) in the context of \( \gamma \). Recall that a pattern \( \pi_1 \) here is not an arbitrary Athena pattern (as can appear, e.g., inside a \texttt{match} clause), but is instead an element of a more restricted class of patterns described by the following grammar:

\[
\pi ::= I \mid I:S \mid (I \leftarrow \pi) \mid (I \texttt{ as } \pi)
\]  

(5)

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

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

1. If \( \pi \) is an identifier \( I \), then if \( I \) is the name of a constant symbol \( c \) in \( \gamma \), we let \( t_\pi \) be \( c \) and return \( M \) unchanged. Otherwise \( I \) must be a pattern variable, and in that case we let \( t_\pi \) be a \textit{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., \texttt{'T145}).

2. If \( \pi \) is of the form \( I:S \) for some sort \( S \) in \( \gamma \), 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 \( I:S \) is a well-sorted term.

3. If \( \pi \) is of the form \( (I \, \pi_1 \cdots \pi_k) \) 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 \( \gamma \)), we output \( (I \, t_{\pi_1} \cdots t_{\pi_k}, M_k) \), provided that \( (I \, t_{\pi_1} \cdots t_{\pi_k}) \) is a well-sorted term (an error is generated if it is not).

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

---

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

13 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.
A pattern \( \pi_i \) in (3), \( i \in \{1, \ldots, n\} \), is considered proper for \( S_D \) in the context of \( \gamma \) provided that the term \( t_{\pi_i} \), 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 \( t_{\pi_i} \) 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 \((:: \text{head tail})\). 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 \((\text{List } 'T145)\). 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 \(?v11: (\text{List } 'T145)\) to head and tail, respectively: 

\[
M = \{ \ldots \text{head } \mapsto ?v10: 'T145, \text{tail } \mapsto ?v11: (\text{List } 'T145), \ldots \}.
\]

Once we have gone through each pattern \( \pi_i \) and computed the corresponding term \( t_{\pi_i} \) and mapping \( M_i \) and ensured that each \( \pi_i \) is proper for \( S_D \) in the context of \( \gamma \), we proceed to do the following for each clause \( \pi_i \Rightarrow D_i, i = 1, \ldots, n \). 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 inductive 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 \( \beta_i \) consist of \( \beta \) augmented with all the inductive hypotheses thus generated, and let \( \rho_i \) be the environment \( \rho \) augmented with the bindings of \( M_i \). We then evaluate \( D_i \) in \( \rho_i, \beta_i, \sigma_i, \) and \( \gamma \), producing some conclusion \( p_i \) and store \( \sigma_{i+1} \) (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 \( \sigma_{i+1} \). 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 \( t_{\pi_i} \). 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 \( \sigma_{n+1} \).

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 \( \lambda \)-calculus expressions as follows:

```plaintext
datatype Exp :=
  (var Ide)      # variables
| (abs Ide Exp)  # abstractions
| (app Exp (List Exp)) # applications
```
A.3. OPERATIONAL SEMANTICS

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:

```latex
\textbf{by-induction} \ (\forall e \ . \ \text{exp-property} \ e) \ \{ \\
(\text{var} \ x) \Rightarrow D_1 \\
| \ (\text{abs} \ x \ e) \Rightarrow D_2 \\
| \ (\text{app} \ \text{proc} \ \text{args}) \Rightarrow D_3 \\
\}
```

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 $\text{args}$:

```latex
\textbf{by-induction} \ (\forall e \ . \ \text{exp-property} \ e) \ \{ \\
(\text{var} \ x) \Rightarrow D_1 \\
| \ (\text{abs} \ x \ e) \Rightarrow D_2 \\
| \ (\text{app} \ \text{proc} \ \text{args}) \Rightarrow \\
\quad \text{let} \ {\text{list-property} := \lambda (L) \ (\text{exp-property} \ (\text{app} \ \text{proc} \ L))}; \\
\quad \text{lemma} := \textbf{by-induction} \ (\forall L \ (\text{list-property} \ L)) \ \{ \\
\quad \text{nil} \Rightarrow D_3 \\
\quad | \ (: \ \text{head} : \text{Exp} \ \text{tail} : (\text{List} \ \text{Exp})) \Rightarrow D_4 \\
\quad \} \\
\} \\
(!\text{uspec lemma args}) \\
\}
```

Here, inside $D_4$ we will not only get an appropriate inductive hypothesis for tail (namely, (list-property tail)),\footnote{Strictly speaking, of course, when we say “tail” we are referring to the fresh term variable to which tail will be bound.} 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 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 \((\text{app proc args})\). 15 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

\[
\text{datatype-cases } F \{ \pi_1 \Rightarrow D_1 \mid \cdots \mid \pi_n \Rightarrow D_n \}
\]

is evaluated (in some \(\rho, \beta, \sigma, \text{ and } \gamma\)) 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:

\[
\text{datatype-cases } F \text{ on } E \{ \pi_1 \Rightarrow D_1 \mid \cdots \mid \pi_n \Rightarrow D_n \}.
\]

Here, the evaluation of \(F\) in \(\rho, \beta, \sigma, \text{ and } \gamma\) must produce a sentence \(p\), along with a store \(\sigma_1\). Further, the evaluation of \(E\) in \(\rho, \beta, \sigma_1, \text{ and } \gamma\) 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 \(t_{\pi_i}\), where \(t_{\pi_i}\) is obtained from \(\pi_i\) exactly as described above for by-induction proofs. There are no inductive hypotheses in this case, but each \(D_i\) is evaluated in \(\beta\) augmented with

---

15 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 \(\text{head}\), that might not be so if \(\text{list-property}\) 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 \(D_1\)) is \((\text{list-property} (:: \text{head} \text{tail}))\), that is, \((\text{exp-property} (\text{app proc} (:: \text{head} \text{tail})))\). The prior goal (in the current stack of inductive clauses) is \((\text{exp-property} (\text{app proc} \text{args}))\). Thus, the current goal matches the prior goal under the substitution \([\text{args} \mapsto (:: \text{head} \text{tail})]\), where \(\text{args}\) is a variable in the outer inductive pattern and \((:: \text{head} \text{tail})\) is the current (inner) inductive pattern. Since \(\text{head}\) 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-sentences in \text{lib/basic/util.ath}).
the identity \( x : S = t_{n_i} \). If the evaluation of each \( D_i \) in the corresponding environment (obtained from \( \rho \) and \( \pi_i \) as in a by-induction proof), the aforementioned assumption base, the appropriate store, and \( \gamma \), results in the correct sentence (namely, \( p \) with every free occurrence of \( x : S \) replaced by \( t_{n_i} \)), 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 \( \pi \), with respect to some environment \( \rho \) and symbol set \( \gamma \), 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 \( \pi \), along with a sort valuation, that is, a finite function from sort variables to sorts. Suppose, for example, that \( V \) is the term \((S S \text{zero})\) and \( \pi \) is the pattern \((S n)\). Here we have a successful match, resulting in the environment \( \{ n \mapsto (S \text{zero}) \} \) (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 \( \pi \), (c) an environment \( \rho \), and (d) a symbol set \( \gamma \); and outputs either (i) a failure token, indicating that \( V \) does not match \( \pi \) with respect to \( \rho \) and \( \gamma \), or else (ii) an environment \( \rho' \) that represents a successful match, along with a sort valuation \( \tau' \). Both the output environment \( \rho' \) and the output sort valuation \( \tau' \) are built up in stages, and it is convenient to express the algorithm so that it takes \( \rho' \) and \( \tau' \) 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

\[
\text{(pair } ?x:'S ?y:'T)\]

is successfully matched against a pattern like \((\text{pair left:} \text{Ide right:} \text{Real})\). Intuitively, \((\text{pair } ?x:'S ?y:'T)\) 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 \((\text{pair } ?x: \text{Ide } ?y: \text{Real})\), obtained by replacing ‘\( S \)’ by ‘\( \text{Ide} \)’ and ‘\( T \)’ by
Figure A.4
Syntax of Athena patterns

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.
A.4. PATTERN MATCHING

In summary, then, the algorithm takes as input: (a) a value $V$; (b) a pattern $\pi$; (c) an environment $\rho$; (d) a symbol set $\gamma$; (e) the auxiliary environment $\rho'$ that is to be incrementally built up; and (f) a sort valuation $\tau$. 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 $(\rho'', \tau')$ consisting of an environment $\rho''$ and a sort valuation $\tau'$ that extend $\rho'$ and $\tau$, respectively. The algorithm proceeds by a case analysis of the structure of $\pi$:

- Case 1: $\pi$ is the wildcard pattern _. In that case return $(\rho', \tau)$.
- Case 2: $\pi$ is a term variable of sort $S$ (such as $?x:\text{Boolean}$). In that case, if $V$ is a term variable of the same name and of sort $T$ such that $\tau (S)$ and $\tau (T)$ are unifiable under a unifier $\tau'$, return $(\rho', \tau'' \tau')$, where $\tau''$ is the composition of $\tau$ and $\tau'$; otherwise fail.
- Case 3: $\pi$ 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 $(\rho', \tau)$; otherwise fail.
- Case 4: $\pi$ 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 $\rho'$ and $\tau$ unchanged. Otherwise we consult $\gamma$ 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 $\tau (S_t)$ and $\tau (S)$ can be unified under some most general unifier $\tau'$. If not, we fail, otherwise we return $\rho'$ unchanged along with the composition of $\tau'$ and $\tau$.
   - If there is not, we check to see if the value $V$ is that exact same function symbol $(I)$. If it is not, we fail; otherwise we return $\rho'$ and $\tau$ unchanged.

2. If it is not, that means that $I$ is a pattern variable. We then check to see if $\rho'$ already assigns a value $V_I$ to $I$:
   - Suppose that it does. Then we again check whether there is a sort annotation $S$:

---

16 The arguments $\rho$ and $\gamma$ 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.

17 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.

18 For any sort $S$ and sort valuation $\tau$, $\tau (S)$ denotes the sort obtained from $S$ by replacing every occurrence of a sort variable in $S$ by the unique sort that $\tau$ assigns to that variable (if a variable is not in the domain of $\tau$, then it is returned unchanged). This was written as $T(S)$ in Section 2.8, but we use the simpler notation here.

19 Note that a constant term (such as zero or nil) can be coerced into a function symbol and vice versa.
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 \( \rho' \) and \( \tau \) 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 \( \tau' \), then 
let \( t_1 \) and \( t_1' \) be the terms obtained from \( t \) and \( t' \), respectively, by applying the 
composition of \( \tau' \) and \( \tau \) to their sorts.\(^{20}\) If these two terms are identical, then 
we return \( \rho' \) 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) \( \rho' \) augmented with \( \{I \mapsto V\} \); and (b) \( \tau \) 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 \( \tau' \). If not, we fail. Otherwise, we return 
(a) \( \rho' \) augmented with \( \{I \mapsto t'\} \), where \( t' \) is the term obtained by applying the 
composition of \( \tau' \) and \( \tau \) to \( t \); and (b) the said composition.

Case 5: \( \pi \) is a list pattern of the form \( [\pi_1 \ldots \pi_n] \). In that case, if \( V \) is a list of values 
\( [V_1 \ldots V_n] \), we return the result of trying to sequentially match \( V_1, \ldots, V_n \) against the 
patterns \( \pi_1, \ldots, \pi_n \) in \( \rho \), \( \gamma \), \( \rho' \), and \( \tau \) (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: \( \pi \) is a list pattern of the form \( \text{list-of } \pi_1 \pi_2 \). In that case, if \( V \) is a nonempty 
list of values \( [V_1 \ldots V_n] \), \( n > 0 \), we return the result of trying to sequentially match the 
values \( V_1, [V_2 \ldots V_n] \) against the patterns \( \pi_1, \pi_2 \) (again, in \( \rho \), \( \gamma \), \( \rho' \), and \( \tau \) ). Otherwise we fail.

Case 7: \( \pi \) is a list pattern of the form \( \text{split } \pi_1 \pi_2 \). 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 \( \pi_1, \pi_2 \) under \( \rho \), \( \gamma \), \( \rho' \), and \( \tau \), producing a result \( (\rho'', \tau') \). 
If so, we return that result. If no such decomposition of \( V \) exists, we fail.

Case 8: \( \pi \) is of the form \( \text{val-of } I \). We then check to see if \( I \) is bound in \( \rho \). If it is not, 
we fail. If it is bound to some value \( V' \), we check whether the two values \( V \) and \( V' \) are

\(^{20}\) To apply a sort valuation to a term means to apply the sort valuation to every sort annotation in that term.
identical. If the values do not admit equality testing, or if they do but are not identical, we fail, else we return $\rho'$ and $\tau$ unchanged.

- **Case 9:** $\pi$ is of the form $\langle \text{bind } I \; \pi' \rangle$, or equivalently, $\langle I \; \text{as } \pi' \rangle$. In that case we match $V$ against $\pi'$ in $\rho$, $\gamma$, $\rho'$, and $\tau$. If that fails, we fail. Otherwise, if it produces a result $(\rho'', \tau')$, we return $(\rho''[I \mapsto V], \tau')$.

- **Case 10:** $\pi$ is of the form $\langle \pi' \; \text{where } E \rangle$. We then match $V$ against $\pi'$ in $\rho$, $\gamma$, $\rho'$, and $\tau$. If that fails, we fail. Otherwise, if we get a result $(\rho'', \tau')$, we evaluate the expression $E$ in the environment $\rho$ augmented with the bindings in $\rho''$, $\gamma$, and some appropriate assumption base and state. If that evaluation results in true, we return $(\rho'', \tau')$, otherwise we fail.

- **Case 11:** $\pi$ 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 $n = 2$, $\pi_1$ is a quantifier pattern,\(^{21}\) and $\pi_2$ is a list pattern, then:
      
      * If $t$ is term of sort $\text{Boolean}$, so that it can be treated as a sentence $p$, try to match it as a sentence against $\pi$ (in $\rho$, $\gamma$, $\rho'$, and $\tau$), using the algorithm given below (under case (ii)).
      * If $t$ is not a term of sort $\text{Boolean}$, fail.

    2. Otherwise, we distinguish the following subcases:
      
      * $n = 1$ and $\pi_2$ is a list pattern. In that case we try to sequentially match the values $f, [t_1 \cdots t_m]$ against the patterns $\pi_1, \pi_2$ (in $\rho$, $\gamma$, $\rho'$, and $\tau$).
      * Otherwise, if $n > 1$ or $\pi_2$ 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 $\rho$, $\gamma$, $\rho'$, and $\tau$).

  - (ii) $V$ is a sentence $p$. We then distinguish two subcases, on the basis of the pattern:
    
    1. $n = 2$, $\pi_1$ is a quantifier pattern, and $\pi_2$ is a list pattern. In that case, we first express $p$ in the form

        $$\langle Q \; x_1 \cdots x_k \; p' \rangle$$

    for some quantifier $Q$, $k \geq 0$, and a sentence $p'$ that is not of the form $(\langle Q \; y_1 \cdots y_m \; p'' \rangle)$, $m > 0$. 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

---

\(^{21}\)That is, one of the two pattern constants forall, exists; or a pattern of the form $\langle \text{some-quant } J \rangle$; or else a pattern of the form $\langle \text{bind } I \; \pi \rangle$, where $\pi$ is a quantifier pattern.
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 \( \pi_1 \) is of the form (some-quant I) or (bind I \( \pi' \)) for some quantifier pattern \( \pi' \), fail. Otherwise, let \( \rho'' = \rho' \) and \( \tau' = \tau \), and continue.

* If \( k > 0 \) then match \( Q \) against the pattern \( \pi_1 \) in \( \rho, \gamma, \rho', \) and \( \tau \), and let \( (\rho'', \tau') \) 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 \( \pi_2 \) in \( \rho, \gamma, \rho'' \) and \( \tau' \), and let \( (\rho''', \tau'') \) be the result.

* Return the result of matching the sentence \( (Q \ x_{i+1} \cdots x_k \ p') \) against \( \pi_3 \) in \( \rho, \gamma, \rho''', \) and \( \tau'' \). (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 \( \pi \) (in \( \rho, \gamma, \rho' \), and \( \tau \)).

* 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 \( \pi_2 \) is a list pattern, we sequentially match \( \circ, [p_1 \cdots p_k] \) against the patterns \( \pi_1, \pi_2 \) in \( \rho, \gamma, \rho', \) and \( \tau \).

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

* 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: \( \pi \) is a filter pattern, that is, of the form (some-co \ 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 \( \rho' \) and \( \tau \) unchanged. In case (a), we return \( \rho'[I \mapsto V] \) and \( \tau \). When \( \pi \) 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 \( (\rho', \tau) \) in case (b). Likewise for the remaining filter patterns. For instance, if \( \pi \) is of the form (a) (some-sent-co I) or (b) (some-sent-co _), 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 \( (\rho', \tau) \) 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 \(\rho, \gamma, \rho', \text{ and } \tau\), we do the following: First, if \(n \neq m\), we fail. Otherwise, if \(n = 0\), we return \((\rho', \tau)\). Finally, if \(n = m\) and \(n > 0\), we match \(V_1\) against \(\pi_1\) in \(\rho, \gamma, \rho', \text{ and } \tau\), resulting in some output pair \((\rho_1', \tau_1)\), and then we proceed to sequentially match (recursively) \(V_2, \ldots, V_n\) against \(\pi_2, \ldots, \pi_m\) in \(\rho, \gamma, \rho_1', \text{ and } \tau_1\).

### 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 might have:

\[
\begin{align*}
\text{datatype } N := & \text{ zero } | (\text{ succ pred}:N) \\
\end{align*}
\]

This introduces the selector \(\text{ pred}\) as a function from natural numbers to natural numbers such that \((\text{ pred succ n} = n)\) for all \(n:N\). It is generally a good idea to introduce selectors. 22

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 \(\text{ head}\) as a selector for the first argument position of \(\text{ nat-cons}\) and \(\text{ tail}\) as a selector for the second argument position:

\[
\begin{align*}
\text{datatype } \text{ Nat-List} := & \text{ nat-nil } | (\text{ nat-cons head:N tail:Nat-List)} \\
\end{align*}
\]

Axioms specifying the semantics of selectors are automatically generated and can be obtained by applying the procedure \(\text{ selector-axioms}\) to the name of the datatype:

\[
\begin{align*}
> \text{ (selector-axioms "Nat-List")} \\
\end{align*}
\]

\[
\begin{align*}
\text{ List: [} \\
(\forall v561:N \\
(\forall v562:\text{Nat-List} \\
(= (\text{head (nat-cons v561 v562))} \\
v561)))} \\
(\forall v561:N \\
(\forall v562:\text{Nat-List} \\
(= (\text{tail (nat-cons v561 v562))} \\
\text{v562}))) \\
\] \\
\end{align*}
\]

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.
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:

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


> (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).
### A.6. PREFIX SYNTAX

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>declare I: [S_1 \ldots S_n] \rightarrow S</code></td>
<td>(declare <code>I (\rightarrow (S_1 \ldots S_n) S))</code></td>
</tr>
<tr>
<td><code>declare I: (I_1, \ldots, I_k) [S_1 \ldots S_n] \rightarrow S</code></td>
<td>(declare <code>I ((I_1 \ldots I_k) \rightarrow (S_1 \ldots S_n) S))</code></td>
</tr>
<tr>
<td><code>declare I_1, \ldots, I_m: (I_1', \ldots, I_k') [S_1 \ldots S_n] \rightarrow S</code></td>
<td>(declare <code>I_1, \ldots, I_m ((I_1' \ldots I_k') \rightarrow (S_1 \ldots S_n) S))</code></td>
</tr>
<tr>
<td><code>assert E_1, \ldots, E_n</code></td>
<td>(assert <code>E_1, \ldots, E_n</code>)</td>
</tr>
<tr>
<td><code>assert I := E</code></td>
<td>(assert <code>I := E</code>)</td>
</tr>
<tr>
<td><code>set-flag I &quot;on&quot;/&quot;off&quot;</code></td>
<td>(set-flag <code>I &quot;on&quot;/&quot;off&quot;</code>)</td>
</tr>
<tr>
<td><code>overload I_1 I_2</code></td>
<td>(overload <code>I_1 I_2</code>)</td>
</tr>
<tr>
<td><code>set-precedence I E</code></td>
<td>(set-precedence <code>I E</code>)</td>
</tr>
<tr>
<td><code>set-precedence (I_1, \ldots, I_n) E</code></td>
<td>(set-precedence <code>(I_1, \ldots, I_n) E</code>)</td>
</tr>
<tr>
<td><code>lambda (I_1, \ldots, I_n) E</code></td>
<td>(lambda <code>(I_1, \ldots, I_n) E</code>)</td>
</tr>
<tr>
<td><code>method (I_1, \ldots, I_n) D</code></td>
<td>(method <code>(I_1, \ldots, I_n) D</code>)</td>
</tr>
<tr>
<td><code>cell F</code></td>
<td>(cell <code>F</code>)</td>
</tr>
<tr>
<td><code>set! E F</code></td>
<td>(set! <code>E F</code>)</td>
</tr>
<tr>
<td><code>ref E</code></td>
<td>(ref <code>E</code>)</td>
</tr>
<tr>
<td><code>while F E</code></td>
<td>(while <code>F E</code>)</td>
</tr>
<tr>
<td><code>make-vector E F</code></td>
<td>(make-vector <code>E F</code>)</td>
</tr>
<tr>
<td><code>vector-sub E_1 E_2</code></td>
<td>(vector-sub <code>E_1 E_2</code>)</td>
</tr>
<tr>
<td><code>vector-set! E_1 E_2 F</code></td>
<td>(vector-set! <code>E_1 E_2 F</code>)</td>
</tr>
<tr>
<td><code>check \{ F_1 \Rightarrow F_1', \ldots, F_n \Rightarrow F_n' \}</code></td>
<td>(check <code>\{ F_1 \rightarrow F_1', \ldots, F_n \rightarrow F_n' \}</code>)</td>
</tr>
<tr>
<td><code>match F [\pi_1 \Rightarrow F_1, \ldots, \pi_n \Rightarrow F_n]</code></td>
<td>(match <code>F (\pi_1 \rightarrow F_1, \ldots, \pi_n \rightarrow F_n)</code></td>
</tr>
<tr>
<td><code>let (\pi_1 := F_1; \ldots; \pi_n := F_n) F</code></td>
<td>(let <code>(\pi_1 := F_1; \ldots; \pi_n := F_n) F</code>)</td>
</tr>
<tr>
<td><code>letrec (I_1 := E_1; \ldots; I_n := E_n) F</code></td>
<td>(letrec <code>(I_1 := E_1; \ldots; I_n := E_n) F</code>)</td>
</tr>
<tr>
<td><code>try F_1; \ldots; F_n</code></td>
<td>(try <code>F_1; \ldots; F_n</code>)</td>
</tr>
<tr>
<td><code>assume F D</code></td>
<td>(assume <code>F D</code>)</td>
</tr>
<tr>
<td><code>assume I := F D</code></td>
<td>(assume <code>(I F D)</code></td>
</tr>
<tr>
<td><code>pick-any I D</code></td>
<td>(pick-any <code>I D</code>)</td>
</tr>
<tr>
<td><code>pick-witness I for F D</code></td>
<td>(pick-witness <code>I F D</code>)</td>
</tr>
<tr>
<td><code>pick-witnesses I_1, \ldots, I_n for F D</code></td>
<td>(pick-witnesses <code>(I_1, \ldots, I_n) F D</code>)</td>
</tr>
</tbody>
</table>