May 1997 Draft of the ASM Guide

Yuri Gurevich

Abstract

This is only a draft of a portion of the ASM guide, but there are no plans to change this part in any substantial way, only to augment it.

Contents

• Introduction (to be written)
• States
• Actions
• Basic Rules
• Shortcuts
• First-order extensions
• Nondeterministic Basic Rules
• Etc.
1 States

What mathematical objects are needed to describe arbitrary states of arbitrary algorithms on their natural abstraction levels? Is there one kind of mathematical objects sufficient for the job? Mathematical logic offers us a good candidate: structures. The notion of structure was introduced in [Tarski 1936] and quickly became standard. Experience confirms the suitability of structures.

For the use of structure in dynamic situations, a slight modification of the notion of structure is convenient. Modified structures will be called states.

Remark for logicians. Tarski structures may be called one-sorted first-order structures. It is well known, however, that many-sorted and/or higher-order structures are naturally represented by Tarski structures. For example, a topological space can be represented as a Tarski structure with subuniverses Points, PointSets, the binary relation $\in$ and a unary relation Open over PointSets. In states, such subuniverses will be called universes. Accordingly, the base set of the state may be called a superuniverse.

$\square$

1.1 Vocabularies

A vocabulary is a finite collection of function names, each of a fixed arity. Some function names may be marked as relational or static, or both. Relational names may be called predicates. In this guide, the Greek letter $\Upsilon$ is reserved to denote vocabularies, and function names are by default non-relational (that is, if we introduce a function name and do not say explicitly that it is relational, then it is not relational).

Every vocabulary contains the following logic names: the equality sign, nullary function names $true$, $false$, $undef$, unary predicate Boole and the names of the usual Boolean operations. All logic names are static. With the exception of $undef$, all logic names are relational. We forgo the precise syntax of function names here.

Examples 1. A vocabulary for oriented trees: a unary predicate Nodes and unary function names Parent, FirstChild and NextSibling.

2. A vocabulary for pointed oriented trees, that is oriented trees with a distinguished node. Add a nullary function name $C$ (an allusion to Current Node) to the tree vocabulary.

3. A vocabulary for (string) labeled oriented trees. Add the following names to the oriented tree vocabulary: a unary predicate Strings and a unary function Label.

4. Combine 2 and 3 to get a vocabulary for pointed labeled oriented trees. $\square$
### 1.2 Terms

Another syntactic category is variables which has a syntactic subcategory of Boolean variables; we forgo the precise syntax of variables as well. Terms are defined recursively, as in first-order logic. We use the same recursion to define Boolean terms.

- A variable is a term. If the variable is Boolean, then the term is Boolean.
- If \( f \) is a function name of arity \( r \) and \( t_1, \ldots, t_r \) are terms, then \( f(t_1, \ldots, t_r) \) is a term. If \( f \) is relational than the composed term is Boolean.

Remark. In the second clause, \( r \) may be zero. A pedantic question arises whether one should write \( f \) or \( f() \) in that case. Both are admissible. We use the shorter version.

**Examples** Here are several Boolean terms in the vocabulary of pointed oriented trees. The intended meaning should be clear:

\[
\begin{align*}
\text{Parent}(C) &= \text{undef} \\
\text{FirstChild}(C) &= \text{undef} \\
\text{NextSibling}(C) &= \text{undef} \\
\text{Parent}(\text{FirstChild}(C)) &= C \\
\text{Parent}(C) &= \text{Parent(NextSibling}(C))
\end{align*}
\]

### 1.3 States

A state \( A \) of vocabulary \( \mathcal{Y} = \text{Voc}(A) \) is a nonempty set \( X \) together with interpretations of the function names in \( \mathcal{Y} \) on \( X \). The set \( X \) is called the base set or superuniverse of \( A \); it will be denoted \( \text{BaseSet}(A) \). An \( r \)-ary function name is interpreted as a function from \( X^r \) to \( X \), a basic function of \( A \). In particular, a nullary name is interpreted as an element of \( X \). There are some constraints on the interpretations.

- \( \text{true, false, undef} \). The interpretations of these three names are distinct. We will not distinguish between these three names and their interpretations.

- Predicates. The interpretation of an \( r \)-ary predicate \( P \) is a function from \( X^r \) to \{true, false\}, a basic relation of \( A \). Think about a basic relation \( P \) as the collection of \( r \)-tuples \( \vec{a} \) such that \( P(\vec{a}) = \text{true} \). If relation \( P \) is unary it can be viewed as a universe.

- Boole and Boolean operations. Boole is the universe \{true, false\}. The Boolean operations behave in the usual way on Boole and produce false if at least one of the arguments is not Boolean. The equality sign is interpreted as the identity relation on \( X \).
Terminological remark. Elements of $X$ will also be called elements of $A$.

Remark on the Boolean terms. If a Boolean term $t$ evaluates to the element $true$ at a state $A$, we say that $t$ holds at $A$ and that $A$ satisfies $t$; symbolically $A \models t$. If $t$ evaluates to $false$, we say that $t$ fails at $A$.

Remark on the role of $undef$. Formally speaking, basic functions are total. However, we view them as being partial and define the domain $\text{Dom}(f)$ of an $r$-ary basic function $f$ as the set of $r$-tuples $\bar{x}$ such that $f(\bar{x}) \neq undef$. Let us stress though that $undef$ is an ordinary element of the superuniverse. Often, a basic function produces $undef$ if at least one argument equals $undef$, but this is not required and there are exceptions, for example, the equality produces $true$ at $(undef, undef)$. As a rule, $undef$ is not included in universes.

Example An oriented tree with $n$ nodes gives rise to a state with $n + 3$ elements. In addition to $n$ nodes, the base set contains the obligatory logic elements $true, false, undef$. The universe Nodes contains the $n$ nodes and does not contain the logic elements. □

1.4 The reserve of a state

Many algorithms require additional space as they run. In the abstract setting, this seems to mean that the state acquires new elements. It may be more convenient to have a source of new elements inside the state.

The reserve of a state $A$ contains all elements of $A$ such that

- Every basic relation, with the exception of equality, evaluates to $false$ if at least one of its arguments belongs to the reserve.
- Every non-relational basic function evaluates to $undef$ if at least one of its arguments belongs to the reserve.
- No basic function outputs an element of the reserve.

Remark for those familiar with the Lipari guide. The function name Reserve is obsolete; the new and simpler definition of reserve is purely semantical. □

Dealing with algorithms that may require additional space, we will assume the following proviso.

Proviso Every state has an infinite reserve. □

Example The previous example needs to be revised when the proviso is in force. In addition to the nodes of the tree and the logic elements, the base set of the state includes an infinite reserve. □
1.5 Variable Assignments

A variable assignment over a state \( A \) is a function \( \zeta \) from some collection of variables to \( \text{BaseSet}(A) \). Of course, \( \zeta(v) \) is Boolean for every Boolean variable \( v \) in \( \text{Dom}(\zeta) \). The less obvious constraint is this: \( \zeta \) cannot assign the same reserve element to different variables.

If \( \zeta \) is a variable assignment over \( A \), then the pair \( B = (A, \zeta) \) is an expanded state, \( A = \text{State}(B) \), \( \zeta = \text{Assign}(B) \), \( \text{Voc}(B) = \text{Voc}(A) \), and \( \text{Var}(B) = \text{Dom}(\zeta) \). We do not distinguish between \( A \) and the expanded state \( (A, \zeta) \) where \( \zeta \) is the empty function.

Suppose that \( B \) is an expanded state \( (A, \zeta) \) as above, \( \bar{v} \) is a sequence \( v_1, \ldots, v_j \) of distinct variables, and \( \bar{a} \) is a sequence \( a_1, \ldots, a_j \) of elements of \( A \). Modify \( \zeta \) by assigning or reassigning \( a_i \) to \( v_i \) for \( i = 1, \ldots, j \). If the result \( \zeta' \) is a legal variable assignment, then \( \zeta' \) will be denoted \( \zeta(\bar{v} \mapsto \bar{a}) \) and the expanded state \( (A, \zeta') \) will be denoted \( B(\bar{v} \mapsto \bar{a}) \).

1.6 Term Evaluation

An expanded state \( A \) and a term \( t \) are appropriate for each other if the vocabulary of \( A \) contains all function names in \( t \) and the variables of \( A \) include the variables of \( t \).

Consider an expanded state \( A \) and let \( t \) range over terms appropriate for \( A \). The value \( \text{Val}_A(t) \) of a term \( t \) at \( A \) is defined by the obvious induction. If \( t \) is a variable and \( \zeta = \text{Assign}(A) \) then \( \text{Val}_A(t) = \zeta(t) \). If \( t = f(\bar{s}) = f(s_1, \ldots, s_r) \) then

\[
\text{Val}_A(t) = f_A\left(\text{Val}_A(s_1), \ldots, \text{Val}_A(s_r)\right) = f_A\left(\text{Val}_A(\bar{s})\right).
\]
2 Actions

In dynamic situations, it is convenient to view a state as a kind of memory that maps locations to values. Fix a state $A$ of vocabulary $\Upsilon$.

2.1 Locations

A *location* of $A$ is a pair $\ell = (f, \bar{a})$, where $f$ is a function name in $\Upsilon$ of some arity $r$ and $\bar{a}$ is an $r$-tuple of elements of $A$; location $\ell$ is *relational* if $f$ is a predicate. In the case that $f$ is nullary, $(f, ())$ is abbreviated to $f$. The element $f_A(\bar{a})$ is the *content* of location $\ell$. Thus a state can be viewed as a mapping from its locations to their contents.

**Example** Assume that we have a pointed tree and let $a$ be any of the nodes. Then some locations are

$$C, \ (\text{Parent}, a), \ (\text{FirstChild}, a), \ (\text{NextSibling}, a).$$

If $B$ is an expanded state and $A = \text{State}(B)$, then every term $t$ of the form $f(\bar{s})$, appropriate for $B$, gives rise to a location

$$\text{Loc}_B(t) = (f, \text{Val}_B(\bar{s}))$$

over $A$. (The only terms that do not have the form $f(\bar{s})$ are variables.)

2.2 Updates

An *update* of $A$ is a pair $\alpha = (\ell, b)$, where $\ell$ is a location of $A$ and $b$ an element of $A$; if $\ell$ is relational then $b$ must be Boolean. To *fire* $\alpha$ at $A$, put $b$ into the location $\ell$, that is, redefine $A$ to map $\ell$ to $b$. The other locations remain intact. The resulting state is the *sequel* of $A$ with respect to $\alpha$.

**Example** Again, assume that we have a pointed tree and let $a, b$ be any two nodes. Then some updates are

$$(C, a), \ ((\text{Parent}, a), b), \ ((\text{FirstChild}, b), a).$$

Two updates $(\ell_1, b_1)$ and $(\ell_2, b_2)$ *clash* if $\ell_1 = \ell_2$ but $b_1 \neq b_2$. 

6
2.3 Update Sets

An update set over a state $A$ is simply a set of updates of $A$. In this section, we consider only finite update sets. For theoretical purposes, one may raise the issue of infinite update sets; that issue will be addressed elsewhere.

The reserve support $\text{ResSupp}(\beta)$ of an update set $\beta$ is the collection of reserve elements $a$ such that some update $(\ell, b) \in \beta$ involves $a$, that is $a = b$ or, if $\ell = (f, (a_1, \ldots, a_r))$, then $a = a_i$ for some $i$.

An update set $\beta$ is consistent if no two updates in $\beta$ clash. To fire a consistent update set $\beta = \{(\ell_i, b_i) : i \in I\}$, fire all its updates simultaneously. For all $i \in I$, the content of $\ell_i$ is reset to $b_i$; the other locations remain intact. The resulting state is the sequel of $A$ with respect to $\beta$. To fire an inconsistent update set, do nothing; the sequel of $A$ is $A$ itself.

Two update sets $\beta_1$ and $\beta_2$ clash if the union $\beta_1 \cup \beta_2$ is inconsistent.

Reserve Permutations A permutation $\pi$ of the reserve of state $A$ can be viewed as a permutation of the whole base set of $A$ which leaves all non-reserve elements intact. It is easy to see that $\pi$ is an automorphism of $A$ (that is an isomorphism from $A$ onto $A$). $\pi$ can be expanded to locations, updates and update sets in the following natural way:

- If $\ell$ is a location $(f, (a_1, \ldots, a_r))$, then $\pi(\ell) = (f, (\pi(a_1), \ldots, \pi(a_r)))$.
- If $\alpha$ is an update $(\ell, b)$, then $\pi(\alpha) = (\pi(\ell), \pi(b))$.
- If $\beta$ is a set $\{\alpha_i : i \in I\}$ of updates then $\pi(\beta) = \{\pi(\alpha_i) : i \in I\}$.

2.4 Actions

Call two update sets equivalent if

- either both update sets are consistent and there is a reserve permutation which maps one of them onto the other,
- or both sets are inconsistent.

An action is an equivalence class of update sets. (Recall that we consider only finite update sets.) The action containing an update set $\beta$ will be denoted $[\beta]$. The action $[\beta]$ is consistent if and only if $\beta$ is so.

Lemma 1 If $\beta' \in [\beta]$, then the sequel $B$ of $A$ with respect to $\beta$ and the sequel $B'$ of $A$ with respect to $\beta'$ are isomorphic. Furthermore, if $[\beta]$ is consistent, then any
reserve permutation π (viewed as a permutation of BaseSet(A)) which takes β to β′ is an isomorphism from B onto B′.

**Proof** The case of inconsistent [β] is obvious. So we assume that that [β] is consistent. Suppose that a reserve permutation π takes β to β′ and \( f(a_1, \ldots, a_r) = a_0 \) in B. We need to check only that \( f(\pi(a_1), \ldots, \pi(a_r)) = \pi(a_0) \) in B′. Let \( \ell \) be the location \( (f, (a_1, \ldots, a_r)) \).

Case 1: \( \ell \) does not occur in β. Then β does not change the content of \( \ell \), so that \( f(a_1, \ldots, a_r) = a_0 \) in A. Since \( \ell \) does not occur in β, \( \pi(\ell) \) does not occur in \( \pi(\beta) \) and thus \( \pi(\beta) \) does not change the content of \( \pi(\ell) \). It remains to show that \( f(\pi(a_1), \ldots, \pi(a_r)) = \pi(a_0) \) in A.

If \( a_1, \ldots, a_r \) are all non-reserve in A, then, in A, \( a_0 \) is non-reserve and we have

\[
f(\pi(a_1), \ldots, \pi(a_r)) = f(a_1, \ldots, a_r) = a_0 = \pi(a_0).\]

If some \( a_i, i \in [1..r] \), is a reserve element of A, then, in A, \( \pi(a_i) \) is a reserve element, \( a_0 = \text{undef} \), and thus

\[
f(\pi(a_1), \ldots, \pi(a_r)) = \text{undef} = \pi(\text{undef}) = \pi(a_0).\]

Case 2: \( \ell \) does occur in β. Since β is consistent, it contains a unique update \( \alpha \) of location \( \ell \). Clearly, \( \alpha = (\ell, a_0) \). Then \( \pi(\beta) \) contains the update \( (\pi(\ell), \pi(a_0)) \) and therefore \( f(\pi(a_1), \ldots, \pi(a_r)) = \pi(a_0) \) in B′. \( \square \)

The lemma justifies the following definitions: To perform an action [β], fire β. The sequel of A with respect to [β] is the sequel of A with respect to β. Thus the sequel of A with respect to [β] depends on the choice of an update set in [β], but it is defined uniquely up to isomorphism.

### 2.5 The Sum of Actions

**Lemma 2** Let \( \gamma_1, \ldots, \gamma_n \) be actions.

1. There exist update sets \( \beta_i \in \gamma_i \) with pairwise disjoint reserve supports.

2. If update sets \( \beta_i \in \gamma_i \) have pairwise disjoint reserve supports and if update sets \( \beta'_i \in \gamma_i \) have pairwise disjoint reserve supports, then

\[
\beta_1 \cup \cdots \cup \beta_n \text{ is equivalent to } \beta'_1 \cup \cdots \cup \beta'_n.
\]
Proof
1. Pick arbitrary $\beta_i' \in \gamma_i$. We construct the desired $\beta_i$ by induction on $i$. Suppose that $\beta_j$ have been constructed for all $j < i$ and let $\text{ResSupp}(\beta_i') = \{a_1, \ldots, a_m\}$ where the $m$ elements are distinct. Pick distinct reserve elements $b_1, \ldots, b_m$ outside of $\bigcup_{j<i} \text{ResSupp}(\beta_j)$ as well as outside of $\{a_1, \ldots, a_m\}$. The desired $\beta_i = \pi(\beta_i')$ where

$$\pi(a) = \begin{cases} b_k & \text{if } a = a_k \\ a_k & \text{if } a = b_k \\ a & \text{otherwise} \end{cases}$$

2. Let $X_i = \text{ResSupp}(\beta_i)$ and $X_i' = \text{ResSupp}(\beta_i')$. For each $i$, there exists a reserve permutation $\pi_i$ that moves $\beta_i$ to $\beta_i'$. Since sets $X_i$ are pairwise disjoint and sets $X_i'$ are pairwise disjoint, we can define a partial map

$$\pi_0(a) = \pi_i(a) \text{ if } a \in X_i$$

on $\bigcup_i X_i$. Extend $\pi_0$ to a reserve permutation $\pi$ in an arbitrary way. Clearly, $\pi$ moves $\bigcup_i \beta_i$ to $\bigcup_i \beta_i'$. □

To define the sum of actions $\gamma_1, \ldots, \gamma_n$, choose update sets $\beta_i \in \gamma_i$ with pairwise disjoint reserve supports.

$$\gamma_1 + \cdots + \gamma_n = [\beta_1 \cup \cdots \cup \beta_n].$$

2.6 Action Families

An action family $\Gamma$ over $A$ is a nonempty set of actions over $A$. To fire $\Gamma$ over $A$, nondeterministically choose one action $\gamma \in \Gamma$ and perform it at $A$. 

9
3 Basic Rules

3.1 Syntax

*Rules* are defined inductively.

The Skip Rule

skip

is a rule.

Update Rules

\[ f(\bar{s}) := t \]

is a rule with *head* function \( f \). If \( f \) is relational, then the term \( t \) must be Boolean. Here \( \bar{s} \) is a tuple \((s_1, \ldots, s_r)\) of terms where \( r = \text{Arity}(f) \geq 0 \).

Conditional Rules If \( g \) is a Boolean term and \( R_1, R_2 \) are rules then

\begin{verbatim}
if g then R_1
else R_2
endif
\end{verbatim}

is a rule.

Blocks If \( R_1, R_2 \) are rules then

\begin{verbatim}
do in-parallel
   R_1
   R_2
endo
d\end{verbatim}

is a rule with *components* \( R_1, R_2 \). Do-in-parallel rules are called *blocks*. 

Import Rules  If \( v \) is a variable and \( R_0 \) is a rule, then

\[
\text{import } v \\
R_0(v) \\
\text{endimport}
\]

is a rule with head variable \( v \) and body \( R_0 \). Free and bound (occurrences of) variables are defined in the obvious way with “import \( v \)” binding \( v \). Usually \( v \) occurs freely in \( R_0(v) \) (hence the notation \( R_0(v) \)), but this is not required.

3.2 Informal Semantics

A rule \( R \) and an expanded state \( A \) are appropriate for each other if \( \text{Voc}(R) \subseteq \text{Voc}(A) \) and \( \text{FreeVar}(R) \subseteq \text{Var}(A) \). Assume that \( R \) and \( A \) are appropriate for each other. We explain informally the effect of firing of \( R \) at \( A \).

The Skip Rule  skip causes no change.

Update Rules  An update rule \( f(\bar{s}) := t \) causes an update \( (\text{Loc}_A(f(\bar{s})), \text{Val}_A(t)) \). Recall that \( \text{Loc}_A(f(\bar{s})) = (f, \text{Val}_A(\bar{s})) \).

Conditional Rules  Let \( R \) be a conditional rule \( \text{if } g \text{ then } R_1 \text{ else } R_2 \text{ endif} \). To fire \( R \) at \( A \), examine the guard \( g \). If \( g \) holds at \( A \), then fire \( R_1 \); otherwise fire \( R_2 \).

Blocks  The components of a block may contradict each other. For example, consider

\[
\text{do in-parallel} \\
\quad f := \text{true} \\
\quad f := \text{false} \\
\text{enddo}
\]

To fire a block \( R \), determine first if the components are mutually consistent at a given expanded state \( A \). If yes, then fire them simultaneously. If not, do nothing; \( R \) is inconsistent at \( A \).

Import Rules  Suppose that \( R \) is an import rule with head variable \( v \) and the body \( R'(v) \). Choose a fresh reserve element \( a \) and fire \( R'(a) \).
It is supposed that different imports produce different reserve elements. For example, the block

\[
\text{do in-parallel} \\
\text{import } v \quad \text{Parent}(v) := C \quad \text{endimport} \\
\text{import } v \quad \text{Parent}(v) := C \quad \text{endimport} \\
\text{enddo}
\]

creates two children of node C (rather than sometimes two and sometimes one). The reserve can be thought of as additional memory (or storage space) partitioned into disjoint units. Then an import reflects allocation of a unit of memory. It would be wrong to allocate the same unit of memory to different processes at the same time.

### 3.3 Denotational Semantics

The deterministic denotation \( \text{Den}(R) \) of a rule \( R \) is a function on expanded states \( A \) appropriate for \( R \). Each \( \text{Den}(R)(A) \) (or \( \text{Den}(R, A) \) for brevity) is an action over \( \text{State}(A) \). To fire \( R \) at \( A \), perform the action \( \text{Den}(R, A) \) at \( \text{State}(A) \). The sequel of \( A \) with respect to \( R \) is the sequel of \( \text{State}(A) \) with respect to \( \text{Den}(R, A) \). \( \text{Den}(R, A) \) is defined by induction on \( R \).

**The Skip Rule**  \( \text{Den}(\text{skip}, A) = [\emptyset] \).

**Update Rules** If \( R \) is an update rule \( f(\vec{s}) := t \), and \( \alpha \) is the update \( (\text{Loc}_A(f(\vec{s})), \text{Val}_A(t)) \), then

\[
\text{Den}(R, A) = [\{\alpha\}]
\]

**Conditional Rules** If \( R \) is a conditional rule \( \text{if } g \text{ then } R_1 \text{ else } R_2 \text{ endif} \), then

\[
\text{Den}(R, A) = \begin{cases} 
\text{Den}(R_1, A) & \text{if } A \models g \\
\text{Den}(R_2, A) & \text{otherwise}.
\end{cases}
\]

**Import Rules** If \( R \) is an import rule with head variable \( v \) and body \( R_0 \), then

\[
\text{Den}(R, A) = \text{Den}(R_0, A(v \mapsto a))
\]

where \( a \) is any reserve element outside the range of \( \text{Assign}(A) \).
Blocks If $R$ is a block with components $R_1, R_2$, then

\[
\text{Den}(R, A) = \text{Den}(R_1, A) + \text{Den}(R_2, A)
\]

3.4 Syntactic Sugar

Blocks The pair of keywords do in-parallel/endo may be replaced with block/endblock or shortened to do/endo.

It is easy to see that the operation do-in-parallel over rules is associative. Use

```
do in-parallel
    R_1
    R_2
    R_3
endo
```

without worrying whether it abbreviates

```
do in-parallel
    R_1
    do in-parallel
        R_2
        R_3
endo
endo
```

```
do in-parallel
    R_1
    R_2
endo
```

or

```
do in-parallel
    R_1
    R_2
endo
```

Similarly use

```
do in-parallel
    R_1
    : 
    : 
    R_k
endo
```

for any $k$ including $k = 0$ (in which case the rule is a synonym of \texttt{skip}) and $k = 1$ (in which case the rule is a synonym of $R_1$). If two or more components fit on the same line, separate them by commas:
do in-parallel
   \( R_1, R_2 \)
   \( R_3 \)
enddo

The operation do-in-parallel is also commutative and thus the order of components does not matter.

For brevity, the keywords do in-parallel/endo, may be simplified to do/endo, replaced by block/endblock, or omitted altogether (where no confusion arises).

**Remark** In the Lipari guide, by definition, a block can have an arbitrary finite number of components. The difference is of no importance. The present definition is slightly more convenient for theoretical purposes. \( \Box \)

**Conditional Rules** Abbreviate

\[
\text{if } g_1 \text{ then } R_1 \\
\text{else} \\
\quad \text{if } g_2 \text{ then } R_2 \\
\quad \text{else } R_3 \\
\quad \text{endif} \\
\text{endif}
\]

Similarly define conditional rules with multiple clauses:

\[
\text{if } g_1 \text{ then } R_1 \\
\text{elseif } g_2 \text{ then } R_2 \\
\ldots \\
\text{elseif } g_k \text{ then } R_k \\
\text{else } R_{k+1} \\
\text{endif}
\]

The else clause can be omitted in case \( R_{k+1} \) is \textbf{skip}. In particular we have if-then rules of the form

\[
\text{if } g \text{ then } R_1 \text{ endif}
\]

**Remark** In the Lipari guide, by definition, conditional rules had the form
if \( g_1 \) then \( R_1 \)
elseif \( g_2 \) then \( R_2 \)
...
elseif \( g_k \) then \( R_k \)
endif

Again, the difference in the definitions is of no importance; the present form is a little more convenient for theoretical purposes. \( \square \)

**Import Rules**  Abbreviate

\[
\begin{align*}
& \text{import } v_1 \\
& \quad \text{import } v_2 \\
& \quad R \\
& \quad \text{endimport} \\
& \text{endimport}
\end{align*}
\]

and generalize this to more than two imports. Further, abbreviate

\[
\begin{align*}
& \text{import } v_1, \ldots, v_k \\
& \quad U(v_1) := \text{true} \\
& \quad \vdots \\
& \quad U(v_k) := \text{true} \\
& \quad R \\
& \quad \text{endimport}
\end{align*}
\]

\[
\begin{align*}
& \text{extend } U \text{ with } v_1, \ldots, v_k \\
& \quad R \\
& \quad \text{endextend}
\end{align*}
\]

Here \( U \) is any unary predicate.

**The Ends**  One can use a simple \texttt{end} instead of \texttt{endif}, \texttt{enddo}, etc. though the fuller forms make parsing easier. This remark applies also to rule constructs introduced in later sections.
4 Shortcuts

Some convenient syntactic sugar is offered.

4.1 The let Construct

Sometimes a lengthy term \( t \) appears several times in a rule. In such cases, the auxiliary let construct saves the trouble of writing \( t \) over and over again.

Syntax

\[
\text{let } x = t \\
R_0(x) \\
\text{endlet}
\]

where a Boolean variable if \( t \) is Boolean-valued.

Semantics The let rule above is equivalent to the rule

\[
R_0(x \mapsto t)
\]

where \( R_0(x \mapsto t) \) is the result of substituting \( t \) for \( x \) (with all the care that substitutions require, as in first-order logic).

4.2 If-then-else Terms

Extend the definition of terms with the following clause. If \( b \) is a Boolean term and \( t_1, t_2 \) are terms, then

\[
(\text{if } b \text{ then } t_1 \text{ else } t_2)
\]

is a term. In the case when \( t_1, t_2 \) are Boolean, the new term is Boolean (and equivalent to \( (b \land t_1) \lor (\neg b \land t_2) \)).

The semantics is obvious. Alternatively, one can introduce a new ternary logic function IfThenElse.
4.3 The try Construct

4.3.1 Syntax

Try Rules If \( R_1, R_2 \) are rules then

\[
\begin{align*}
\text{try} & \quad R_1 \\
\text{else} & \quad R_2 \\
\text{endif}
\end{align*}
\]

is a rule with positive component \( R_1 \) and negative component \( R_2 \).

4.3.2 Semantics

Let \( R \) be a try rule with positive component \( R_1 \) and negative component \( R_2 \), and let \( A \) be an expanded state appropriate for \( R \). To fire \( R \) at \( A \), check if \( R_1 \) is consistent at \( A \). If yes, fire \( R_1 \); otherwise fire \( R_2 \). In other words,

\[
\text{Den}(R, A) = \begin{cases} 
\text{Den}(R_1, A) & \text{if this action is consistent;} \\
\text{Den}(R_2, A) & \text{otherwise.}
\end{cases}
\]

4.3.3 Guaranteed Consistency

Abbreviate

\[
\begin{align*}
\text{try} & \quad \text{do in-parallel} \\
& \quad R_1, R_2 \\
& \quad \text{endo} \\
\text{else} & \quad R_3 \\
\text{endtry}
\end{align*}
\]

Try in-parallel

\[
\begin{align*}
\text{try in-parallel} & \quad R_1, R_2 \\
& \quad \text{to} \\
& \quad \text{ifclash} \\
& \quad R_3 \\
& \quad \text{endo}
\end{align*}
\]

Theorem 1 If \( R \) is any rule where the do-in-parallel construct is used only within the try-in-parallel construct, then the action \( \text{Den}(R, A) \) is consistent at any expanded state appropriate for \( R \).
4.3.4 Try Rules can be Eliminated

**Theorem 2** Every rule in the basic programming language extended with the try construct is equivalent to a basic rule.
5 First-Order Extensions

First, we enrich basic rules by introducing first-order terms. Then we generalize the do-in-parallel construct.

5.1 First-Order Terms

Even though the definition of terms in the States section is the usual definition of terms in first-order logic, the treatment of relations as Boolean-valued functions allows us to represent quantifier-free first-order assertions as Boolean terms. Many applications require more general first-order terms.

5.1.1 Syntax of First-Order Terms

Extend the definition of terms of Section 1 with the following clause.

- If \( v \) is a variable and \( g(v), s(v) \) are Boolean terms, then

\[
(\forall v : g(v)) s(v), \ (\exists v : g(v)) s(v)
\]

are terms with head variable \( v \), guard \( g(v) \) and body \( s(v) \). Both terms are Boolean.

The definition of free and bound variables is the obvious one. In particular, \( v \) is bound in \((\forall v : g(v)) s(v)\) and \((\exists v : g(v)) s(v)\). It is not required that the variable \( v \) has free occurrences in the guard \( g(v) \) or the body \( s(v) \), but it usually does. Terms defined in Section 1 may be called quantifier-free.

In applications, the guard ensures the finiteness and feasibility of the quantification range. Logically, quantification makes perfect sense even if the range is infinite. In any case, the guard \( g(v) \) may be the term true in which case it may be omitted.

Notice that the enrichment of the notion of term enriches the notion of rule.

Remark. It seems contrary to the tradition of first-order logic to use strange terms like \( f((\exists v : g(v)) s(v)) \) which use quantified terms as arguments. We could forbid such strange terms; then a Boolean term would be not necessarily a term. But it does not really matter. Rules with strange terms can be transformed to equivalent rules without strange terms. For example,
\[ f((\exists v : g(v)) s(v)) := t \]
is equivalent to
\[
\begin{align*}
\text{if } (\exists v : g(v)) s(v) \text{ then} & \\
& f(\text{true}) := t \\
\text{else} & \\
& f(\text{false}) := t \\
\text{endif}
\end{align*}
\]

5.1.2 Semantics of First-Order Terms

Define

\[
\text{Range}_A(v : g(v)) = \left\{ a \in \text{BaseSet}(A) : a \text{ is non-reserve and } A \models g(a) \right\}
\]

where, as usual in first-order logic, \( A \models g(a) \) means \( A(v \mapsto a) \models g(v) \). Here the extended state \( A \) is appropriate for the term \( (\exists v) g(v) \), so that every \( A(v \mapsto a) \) is appropriate for \( g(v) \).

A pedantic point. In general \( a \) is not a term and therefore \( g(a) \) is not a term. It can be called a quasi-term.

Now let \( t_1 = (\forall v : g(v)) s(v) \) and \( t_2 = (\exists v : g(v)) s(v) \). If an expanded state \( A \) is appropriate for \( t_1, t_2 \), then

\[
\begin{align*}
\text{Val}_A(t_1) &= \begin{cases} 
\text{true} & \text{if } \text{Val}_{A(v \mapsto a)}(s(v)) = \text{true} \text{ for all } a \in \text{Range}_A(v : g(v)); \\
\text{false} & \text{otherwise.}
\end{cases} \\
\text{Val}_A(t_2) &= \begin{cases} 
\text{true} & \text{if } \text{Val}_{A(v \mapsto a)}(s(v)) = \text{true} \text{ for some } a \in \text{Range}_A(v : g(v)); \\
\text{false} & \text{otherwise.}
\end{cases}
\end{align*}
\]

Notice that the quantification is restricted to non-reserve elements. Intuitively, the reserve represents the outside world.

**Example** In a state that includes a directed graph with a distinguished node \( C \), the rule

\[
\begin{align*}
\text{if } (\exists v : v \in \text{Nodes}) E(C; v) \text{ then} & \\
& \text{Color}(C) := \text{Green} \\
\text{endif}
\end{align*}
\]

colors \( C \) green provided \( C \) has an outgoing edge. \( \square \)
5.1.3 Abbreviations

\[(\forall u, v : g(u, v)) s(u, v)\]

can be defined as a double quantification. One can impose no guard on \(u\):

\[(\forall u)(\forall v : g(u, v)) s(u, v)\];

one can impose the strongest guard on \(u\):

\[(\forall u : (\exists v) g(u, v)) (\forall v : g(u, v)) s(u, v)\];

any intermediate guard will work as well. Logically all these formulas are equivalent. The case of \(\exists\) is similar. This generalizes to quantification over triples of variables, quadruples of variables, etc.

**Example** Suppose that variables \(u, v\) have been declared to range over non-negative integers. The quantification scope of

\[(\forall u, v : u < v < 100) E(u, v)\]

consists of \(\frac{100 \cdot 99}{2}\) integer pairs \(u < v < 100\). □

5.2 Do-forall Rules

5.2.1 Syntax

Extend the definition of rules with the following clause:

**Do-forall Rules** If \(v\) is a variable, \(g(v)\) is a Boolean term and \(R_0(v)\) is a rule, then

```
  do forall v : g(v)
    R_0(v)
  enddo
```

is a rule with head variable \(v\), guard \(g(v)\) and body \(R_0\). The definition of free and bounded variables is obvious.

Remark. The do-forall construct replaces the var (or vary) construct of the Lipari guide.
5.2.2 Informal Semantics

A do forall rule is similar to the do in parallel rule, except that the component are not listed explicitly one after another. Suppose that \( R \) is the do forall rule above. At an expanded state \( A \) appropriate for \( R \), the components are \( R_0(a) \) where \( a \in \text{Range}_A(v : g(v)) \). To fire \( R \) at \( A \), fire simultaneously all these \( R_0(a) \) unless they are mutually inconsistent. In the latter case, do nothing.

A pedantic point. In general \( a \) is not a term and thus \( R_0(a) \) is not a rule. It may be called a quasi-rule. Firing \( R_0(a) \) at a given expanded state \( A \) appropriate for \( R \) means firing the rule \( R_0(v) \) at the further expanded state \( A(v \mapsto a) \).

Example  In an expanded state that includes a directed graph with variable \( u \) indicating a node, the rule

\[
\text{do forall } v, w : v \in \text{Nodes and } w \in \text{Nodes and } E(v, w), \\
\quad \text{Color}(v) := \text{Blue} \\
\quad \text{Color}(w) := \text{Yellow} \\
\text{enddo}
\]

colors blue all nodes with outgoing edges and colors yellow all nodes with incoming edges, unless the two sets of nodes intersect, in which case nothing is done. \( \square \)

5.2.3 do in parallel as a Special Case of do forall

The rule

\[
\text{do} \\
\quad R_1, R_2 \\
\text{endo}
\]

is equivalent to the rule

\[
\text{do forall } v : \text{Boole}(v) \\
\quad \text{if } v \text{ then } R_1 \text{ else } R_2 \text{ endif} \\
\text{enddo}
\]

where \( v \) is a fresh Boolean variable (and thus the guard \( \text{Boole}(v) \) is redundant).

Abbreviations  A two-parameter do rule
do forall u, v : g(u, v)
  R_0(u, v)
enddo

abbreviates

do forall u : (\exists v) g(u, v)
  do forall v : g(u, v)
    R_0(u, v)
  enddo
enddo

Instead of (\exists v) g(u, v), a more liberal guard may be used; logically this makes no difference.

In a similar way, define do-forall rules with any number k > 2 of head variables.
6 Nondeterministic Rules

6.1 Syntax

Extend the definition of rules with the following clause.

Choose Rules If \( v \) is a variable, \( g(v) \) is a Boolean term, and \( R_0(v) \) is a rule, then

\[
\begin{align*}
\text{choose} & \quad v : g(v) \\
& \quad R_0(v) \\
\text{endchoose}
\end{align*}
\]

is a rule with head variable \( v \), guard \( g(v) \) and body \( R_0(v) \). Free and bound variables are defined in the obvious way. Choose rules are called forks.

6.2 Informal Semantics

Let \( R \) be the choose rule above and let \( A \) be an expanded state appropriate for \( R \). To fire \( R \) at \( A \), check if the set

\[
\text{Range}_A(v : g(v)) = \{ a \in \text{BaseSet}(A) : a \text{ is nonreserve and } A \models g(a) \}
\]

is empty. If yes, do nothing. Otherwise, choose any \( a \in \text{Range}_A(v : g(v)) \) and fire the branch \( R_0(a) \) (that is fire the rule \( R_0(v) \) at \( A(v \mapsto a) \)).

Example In a state that includes a directed graph with at least one edge, the rule

\[
\begin{align*}
\text{choose} & \quad v, w : v \in \text{Nodes and } w \in \text{Nodes and } E(v, w) \\
& \quad \text{ColorEdge}(v, w) := \text{Green} \\
\text{endchoose}
\end{align*}
\]

colors green one of the edges in the graph. The rule does nothing if there are no edges in the graph. □

6.3 Denotational Semantics

The nondeterministic denotation \( \text{NDen}(R) \) of a rule \( R \) is a function on expanded states \( A \) appropriate for \( R \). Each \( \text{NDen}(R)(A) \) (or \( \text{NDen}(R, A) \) for brevity) is an
action family. To fire $R$ at $A$, choose an action $\gamma \in \text{NDen}(R, A)$ and perform it at State($A$); the sequel of State($A$) with respect to $\gamma$ is a sequel of $A$ with respect to $R$. We define $\text{NDen}(R, A)$ by induction on $R$. It is assumed that a given expanded state $A$ is appropriate for $R$.

**Skip and Update rules** If $R$ is skip or an update rule, then

$$\text{NDen}(R, A) = \{ \text{Den}(R, A) \}.$$  

**Conditional Rules** If $R$ is

if $g$ then $R_1$ else $R_2$ endif

then

$$\text{NDen}(R, A) = \begin{cases} \text{NDen}(R_1, A) & \text{if } A \models g(a); \\ \text{NDen}(R_2, A) & \text{otherwise} \end{cases}$$ 

**Do forall Rules** If $R$ is

do forall $v : g(v)$
  $R_0(v)$
endo

then

$$\text{NDen}(R, A) = \left\{ \sum_a \gamma_a : a \in \text{Range}_A(v : g(v)) \land \gamma \in \text{NDen}(R_0(v), A(v \mapsto a)) \right\}$$

**Import Rules** If $R$

import $v$
  $R_0(v)$
endimport

and if $a$ is any reserve element outside of the range of Assign($A$), then

$$\text{NDen}(R, A) = \text{NDen}(R_0, A(v \mapsto a)).$$
Choose Rules  If $R$ is

choose $v : g(v)$
\[ R_0(v) \]
endchoose

and $Q = \text{Range}_A(v : g(v))$ then

\[
\text{NDen}(R, A) = \begin{cases} 
\{ \{ [0] \} \} & \text{if } Q = \emptyset \\
\bigcup \{ \text{NDen}(R_0(v), A(v \mapsto a)) : a \in Q \} & \text{otherwise}
\end{cases}
\]

This completes the definition of NDen.

Remark  Notice a small deviation from the Lipari semantics of choose rules. If \( \text{Range}_A(v : g(v)) \) is empty then the action family \( \text{NDen}(R, A) \) is not empty but contains one action, namely the do-nothing action. Thus, choosing from the empty set of actions is not considered to be contradictory; the result of such a choice is the do-nothing action. □

6.4 Explicit Choice

Define

choose among
\[ R_1 \]
\[ R_2 \]
endchoose

as an abbreviation for

choose $v : \text{Bool}(v)$
\[
\text{if } v \text{ then } R_1 \text{ else } R_2 \text{ endif}
\]
endchoose

where variable $v$ is Boolean. This gives rise to explicit forks

choose among
\[ R_1, \ldots, R_k \]
endchoose

where $k$ is any natural number.
Example Consider a set of nodes with two distinguished nodes Source and Sink and with three unary partial functions BluePointer, GreenPointer, RedPointer from nodes to nodes. Question: Is there a path from Source to Sink that follows, at each step, one of the tree pointer functions? We construct a nondeterministic program for this task. The program uses an auxiliary nullary function C, intuitively the current node. Initially, C equals Source.

if C = Sink then Mode := Final
else
    choose
    C := BluePointer(C)
    C := GreenPointer(C)
    C := RedPointer(C)
endchoose
endif

\qed

6.5 Refining Forks

In applications, a fork

\begin{verbatim}
choose v : g(v) 
    R0(v)
endchoose
\end{verbatim}

may be an abstraction from a mechanism that implements a choice function \( f(\bar{u}) \) subject to constraint

\[(\exists v) g(v) \Rightarrow g(f(\bar{u})).\]

The choice function can be made explicit as an external function:

\begin{verbatim}
if (\exists v) g(v) then
    R0(f(\bar{u}))
endif
\end{verbatim}