This paper has been mechanically scanned. Some errors may have been inadvertently introduced.
On the Formal Static Semantics of SHIFT

Sergio Yovine

September 1998

PATH TECHNICAL NOTE 98-3
This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department of Transportation, Federal Highway Administration.

The contents of this report reflect the views of the authors who are responsible for the facts and the accuracy of the data presented herein. The contents do not necessarily reflect the official views or policies of the State of California. This report does not constitute a standard, specification, or regulation.

Report for MOU 258
On the formal static semantics of SHIFT
(Technical Note)

Sergio Yovine *†
California PATH, UC Berkeley
Richmond Field Station Bldg. 452
1301 S. 46th St, Richmond CA 94804

July 28, 1998

1 Introduction

The precise definition of a specification language like SHIFT involves three steps. The first step consists in the formal description of the grammar which determines the syntactical form of a SHIFT program. The grammar of SHIFT is given in [4]. However, not every grammatically correct specification could be given a formal meaning. Therefore, the second step consists in defining the subset of such specifications that are well-formed, that is, those that have a meaning. This step is called static semantics. Finally, the third step consists in defining the meaning or behavior of the well-formed SHIFT specifications. This step, called dynamic semantics, is discussed in [1].

This document defines the static semantics of SHIFT. It is mainly concerned with giving a set of rules to formally associate a type with every grammatical expression used in a specification, in order to determine whether it is well-typed. In some sense, these rules formalize the type-checking phase of the SHIFT compiler.

Section 2 presents the preliminary definitions and notation. The rest of the document is organized following the grammar. We have focused on the static semantics of expressions, variable declarations and events. The rest of the syntactic constructs could be formalized following the same ideas.

*Also at Verimag, Centre Equation, 2 Av. de Vignate, 38610 Gières, France. E-mail: Sergio.Yovine@imag.fr
†Partially supported by NSF grant ECS-9725148
2 Preliminary definitions and notation

2.1 Semantic objects

The first task in presenting the semantics is to define the objects concerned. The class of syntactic objects, such as reserved words, special constants, comments, identifiers and so on, is defined in the SHIFT Reference Manual [4]. Here we deal with the so-called semantic objects that are used to describe the meaning (type) of the syntactic objects.

Simple objects

The simplest semantic objects are the constants, the identifiers and the basic type names. We use `const` to range over the set of constants `Const`, `id`, `var`, `strid`, and `statename` to range over the set of identifiers `Id`, and `s` to range over `Id U {}`. This is summarize in the following table:

<table>
<thead>
<tr>
<th>const</th>
<th>E</th>
<th>Const</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id, var, strid, statename</code></td>
<td>E</td>
<td>Id</td>
</tr>
<tr>
<td><code>s</code></td>
<td>E</td>
<td>Id U {}</td>
</tr>
</tbody>
</table>

The set of basic types is:

\[
\text{BasicType} = \text{Boo1} \cup \text{Nber} \cup \text{Cont} \cup \text{Disc}
\]

where boo1 corresponds to boolean, Nber corresponds to number, Cont corresponds to continuous number, and Disc corresponds to discrete.

The set of basic type names is built from the set of basic types and the set of identifiers (user defined types):

\[
\text{tynam} \in \text{TyNam} = \text{BasicType} \cup \text{Id}
\]

Compound objects

The compound objects are built from the simple objects by applying some operation (map, Cartesian product, etc) to the simple objects. The compound objects for the static semantics of SHIFT are the types, denoted \( \tau \), the variable environments, denoted \( \nu \), the structure environments, denoted \( \sigma \), and the structures, denoted Str. These compound objects are defined as follows:

\[
\tau \in \text{Type} = \text{TyNam} \cup \text{set}((\text{Type})) \cup \text{array}((\text{Type}))
\]

\[
\nu, \nu' \in \text{VarEnv} = \text{Id} \rightarrow \text{Type}
\]

\[
\sigma, \sigma' \in \text{StrEnv} = \text{Id} \rightarrow \text{Str}
\]

\[
\phi \in \text{Str} = \text{StrEnv} \times \text{Id}^3 \times \text{Id} \times \ldots
\]

Given two variable environments \( \nu, \nu' \in \text{VarEnv} \), we denote by \( \nu + \nu' \), read \( \nu \) modified by \( \nu' \), the variable environment \( \nu'' \) with domain \( \text{Dom} \nu \cup \text{Dom} \nu' \) and values \( \nu''(x) = \nu'(x) \) if \( x \in \text{Dom} \nu' \) and \( \nu''(x) = \nu(x) \), otherwise.
The structure \( T(s) = \phi = (\nu, (S, O) D, \ldots) \) associated with the type component \( s \) is such that:

1. \( \nu \in \text{VarEnv. } \text{Dom} \nu \) is the set of variables declared in the component.
2. \( S \) is the set of state variables.
3. \( I \) is the set of input variables.
4. \( O \) is the set of output variables.

\( T(s) \) satisfies the following properties:

1. \( \text{Dom} \nu = S \cup I \cup O \).
2. \( (S, I, O) \) is a partition of \( \text{Dom} \nu \), i.e., \( S, I, \) and \( O \) are pair-wise disjoint.

We define \( \text{vars} \phi = \text{Dom} \nu, \text{states} \phi = S \), inputs \( \phi = I \), outputs \( \phi = O \), and discrete \( \phi = D \). For all \( x \in \text{vars} \phi \), we define \( \text{typeof} x \in \phi = \nu(x) \).

### 2.2 Contexts

We assume that the SHIFT parser constructs a context \(^1\). A context \( C \) is a triplet \((V, T, \prec)\) such that:

1. \( V \in \text{VarEnv. } \text{Dom} V \) is the set of global variables.
2. \( T \in \text{StrEnv. } \text{Dom} T \) is the set of component types.
3. \( \prec \subseteq \text{Dom} T \times (\text{Dom} T \cup \{\|\}) \) is the inheritance relation.

\( C \) satisfies the following properties:

1. \( \text{Dom} V \cap \text{Dom} T = \emptyset \).
2. \( \prec \) is an acyclic, irreflexive and asymmetric relation.

For \( \tau, \tau', \tau \prec \tau' \) is read \( \tau \) inherits from \( \tau' \).

We define \( \text{Clo} T \) to be the least fixed-point of the equation

\[
X = \text{Dom} T \cup \text{BasicType} \cup \text{set}(X) \cup \text{array}(X).
\]

That is, \( \text{Clo} T \) is the set of types that can be built from the basic types and the user-defined types. We also write \( \text{Clo} C \) for \( \text{Clo} T \).

Since \( \text{Dom} V \) and \( \text{Dom} T \) are disjoint, we write \( C(x) \) to denote \( V(x) \) if \( x \in \text{Dom} V \), otherwise \( T(x) \). We define \( \text{comptypes} C = \text{Dom} T \) and \( \text{vars} C = \text{Dom} V \).

We extend the relation \( \prec \) to the set of all types as follows. We define \( \prec \subseteq \text{Clo} T \times (\text{Clo} T \cup \{\|\}) \) to be the largest relation such that:

\[
\begin{array}{cccc}
\gamma \prec \tau' & \tau \prec \tau' & \gamma \prec \tau' & \gamma \prec \tau' \\
\text{Cont} \prec \text{Nber} & \tau \prec \tau' & \text{set}(\tau) \prec \text{set}(\tau') & \text{array}(\tau) \prec \text{array}(\tau') & \tau \prec \tau'
\end{array}
\]

\(^1\)It is not difficult to formalize this construction.
In words, these rules state that, continuous number is a subtype of number, set (array) of type $\tau$ is a subtype of set (array) of type $\tau'$ whenever $\tau$ is a subtype of $\tau'$ and so on. These rules allows us, for instance, to compare a variable declared as a continuous number with a constant which is of type number, even if syntactically the two objects have different types.

2.3 Form of the rules

We have chosen to present the static semantics in a form known as natural semantics [3, 21]. It consists of a set of rules allowing sentences of the form

$$B \vdash P \Rightarrow M$$

which may be pronounce: “In the background $B$, the (syntactic) phrase $P$ has the meaning $M$”.

In the static semantics of $SHIFT$, the background is typically composed of a context $C$ and some auxiliary (local) information and the meaning is a compound semantic object such as a type or a variable environment.

The semantics is therefore given in a structured way (following the grammar) as a set of rules of the form

$$\frac{B_1 \vdash P_1 \Rightarrow M_1 \ldots B_n \vdash P_n \Rightarrow M_n}{B \vdash P \Rightarrow M}$$

that allow the meaning of a compound phrase $P$ to be obtained from the meanings of its constituent phrases $P_1, \ldots, P_n$.

In the presentation of the rules, expressions (phrases, sentences, etc) within single angle brackets ( ) are called first options, those within double brackets ((( )) are called second options and so on. To reduce the number of rules, we have adopted the following convention: In each instance of a rule, the first options must be either all present or all absent; similarly the second options and so on.

The following sections 3 describe the set of rules for expressions, variable declarations and events. Each section is divided into subsections, each one devoted to a class of phrases obtained according to the grammar. The form of the sentences that apply to each class is shown in the right hand side of the title of the corresponding subsection. Each subsection is divided in two parts, namely grammar and rules. To keep this document short, we have omitted the english explanation of the rules which can be inferred by reading the rules as suggested above and from the intuitive meaning of the syntactic constructs given in the $SHIFT$ Reference Manual [4]. We also assume that the reader is familiar with the language $SHIFT$. 

4
3 Static Semantics of Expressions

3.1 Expressions

3.1.1 Grammar

\[
\text{expr} ::= \text{atexp} \\
| \text{selector} \\
| \text{createxp} \\
| \text{compexpr} \\
| \text{setexpr} \\
| \text{arrayexpr}
\]

3.1.2 Rules

\[
\begin{align*}
C, s, \nu \vdash \text{expr} &\Rightarrow \tau' \vdash \tau \\
C, s, \nu \vdash \text{expr} &\Rightarrow \tau
\end{align*}
\]

3.2 Atomic Expressions

3.2.1 Grammar

\[
\text{atexp} ::= \text{const} \\
| \text{nil} \\
| \text{self} \\
| \text{statename} \\
| ( \text{expr} )
\]

3.2.2 Rules

\[
\begin{align*}
C, s, \nu \vdash \text{const} &\Rightarrow \text{typeof const} \\
\tau \in \text{comptypes} C &\Rightarrow C, s, \nu \vdash \text{nil} \Rightarrow \tau \\
C, s, \nu \vdash \text{self} &\Rightarrow s \\
\text{statename} \notin \text{discrete} C(s) &\Rightarrow C, s, \nu \vdash \text{statename} \Rightarrow \text{Disc} \\
C, s, \nu \vdash \text{expr} &\Rightarrow \tau \\
C, s, \nu \vdash ( \text{expr} ) &\Rightarrow \tau
\end{align*}
\]
3.3 Selectors

3.3.1 Grammar

\[
\text{selector} ::= \text{uar} \\
| \text{uar} (\text{selector})
\]

3.3.2 Rules

\[
C, s, \nu \vdash \text{uar} \Rightarrow \tau, s' \\
C, s, \nu \vdash \nu \Rightarrow \tau
\]

\[
C, s, \nu \vdash \text{selector} \Rightarrow \tau' \\
\tau' \in \text{comptypes} C \quad C, \tau', \nu \vdash \text{uar} \Rightarrow \tau, s' \\
C, s, \nu \vdash \text{uar} (\text{selector}) \Rightarrow \tau
\]

3.4 Create Expressions

3.4.1 Grammar

\[
\text{createexpr} ::= \text{create} (\text{strid} (\text{initlist}))
\]

\[
\text{initlist} ::= \text{initializer} (\text{initlist})
\]

\[
\text{initializer} ::= \text{uar} ::= \text{expr}
\]

3.4.2 Rules

\[
\text{strid} \in \text{comptypes} C \\
C, s, \nu \vdash \text{create} (\text{strid} (\text{initlist})) \Rightarrow \text{strid}
\]

\[
C, s, s' \vdash \text{initializer} \Rightarrow \nu' \\
\text{Dom} \nu \cap \text{Dom} \nu' = \emptyset
\]

\[
C, s, s' \vdash \text{create} (\text{strid} (\text{initlist})) \Rightarrow \nu
\]

\[
C, s, s' \vdash \nu (\text{initlist}) \Rightarrow \nu' \\
\text{Dom} \nu \cap \text{Dom} \nu' = \emptyset
\]

\[
\text{if expr then expr else expr}
\]

3.5 Composed Expressions

3.5.1 Grammar

\[
\text{compexpr} ::= \text{expr op expr} \\
| \text{op expr} \\
| \text{expr op} \\
| \text{if expr then expr else expr}
\]
3.5.2 Rules

\[
\frac{C, s, \nu \vdash \text{expr} \Rightarrow \tau, i = 1, 2 \quad \text{op E InfixOps typeof } \text{op} \equiv \tau_{1} \tau_{2} \Rightarrow \tau}{C, s, \nu \vdash \text{expr}_1 \text{op} \text{expr}_2 \Rightarrow \tau}
\] (12)

\[
\frac{C, s, \nu \vdash \text{expr} \Rightarrow \tau \quad \text{op E PrefixOps typeof } \text{op} \equiv \tau_{1} \Rightarrow \tau}{C, s, \nu \vdash \text{expr}_1 \text{op} \Rightarrow \tau}
\] (13)

\[
\frac{C, s, \nu \vdash \text{expr} \Rightarrow \tau \quad \text{op E PostfixOps typeof } \text{op} \equiv \tau_{1} \Rightarrow \tau}{C, s, \nu \vdash \text{expr}_1 \text{op} \Rightarrow \tau}
\] (14)

\[
\frac{C, s, \nu \vdash \text{expr}_1 \Rightarrow \tau, i = 1, 2 \quad C, s, \nu \vdash \text{expr} \Rightarrow \text{Bool}}{C, s, \nu \vdash \text{if expr then expr}_1 \text{else expr}_2 \Rightarrow \tau}
\] (15)

\[
\frac{C, s, \nu \vdash \text{exists inexpr: expr} \Rightarrow \text{Bool}}{C, s, \nu \vdash \text{exists inexpr: expr} \Rightarrow \text{Bool}}
\] (16)

\[
\frac{C, s, \nu \vdash \text{inexpr} \Rightarrow \nu' \quad \tau \in \text{Ran } \nu'}{C, s, \nu \vdash \text{minel inexpr: expr} \Rightarrow \tau'} \text{ hasorder?}\tau'
\] (17)

\[
\frac{C, s, \nu \vdash \text{inexpr} \Rightarrow \nu' \quad \tau \in \text{Ran } \nu'}{C, s, \nu \vdash \text{maxel inexpr: expr} \Rightarrow \tau'} \text{ hasorder?}\tau'
\] (18)

3.6 Set-Expressions

3.6.1 Grammar

\[
\text{setexpr} ::= \text{all}
\]

\[
\mid \{ \text{exprlist}\}
\]

\[
\mid [\text{expr} \ldots \text{expr}]
\]

\[
\mid \{ \text{expr : inexprlist} (1 \text{expr})\}
\]

3.6.2 Rules

\[
\frac{C, s, \nu \vdash \text{all } \Rightarrow \text{set(Disc)}}{C, s, \nu \vdash \text{set(Disc)}}
\] (19)

\[
\frac{\tau \in \text{Clo } C \quad \langle C, s, \nu \vdash \text{exprlist } \Rightarrow \tau \rangle}{C, s, \nu \vdash \{ \langle \text{exprlist} \rangle \} \Rightarrow \text{set}()}\]

(20)
\[
\begin{align*}
C, s, \nu \vdash \text{expr}_i \Rightarrow \tau, i = 1, 2 & \quad \text{hasorder?} \\
C, s, \nu \vdash [\text{expr}_1 \ldots \text{expr}_2] \Rightarrow \text{set}(\tau)
\end{align*}
\]
\[
C, s, \nu \vdash \text{inexprlist} \Rightarrow \nu' \quad C, s, \nu + \nu' \vdash \text{expr} \Rightarrow \tau \quad (C, s, \nu + \nu' \vdash \text{expr} \Rightarrow \text{Bool})
\]
\[
C, s, \nu \vdash \{\text{expr} : \text{inexprlist} (1 \text{expr})\} \Rightarrow \text{set}(\tau)
\]

3.7 Array-Expressions

3.7.1 Grammar

\[
\text{arrayexpr} ::= [\text{exprlist}] \quad \text{array}(\tau)
\]

3.7.2 Rules

\[
\begin{align*}
\tau \in \text{Clo} C & \quad (C, s, \nu \vdash \text{exprlist} \Rightarrow \tau) \\
C, s, \nu \vdash [\text{exprlist}] \Rightarrow \text{array}(\tau)
\end{align*}
\]

\[
C, s, \nu \vdash \text{inexprlist} \Rightarrow \nu' \quad C, s, \nu + \nu' \vdash \text{expr} \Rightarrow \tau \quad (C, s, \nu + \nu' \vdash \text{expr} \Rightarrow \text{Bool})
\]
\[
C, s, \nu \vdash \{\text{expr} : \text{inexprlist} (1 \text{expr})\} \Rightarrow \text{array}(\tau)
\]

3.8 Lists of Expressions

3.8.1 Grammar

\[
\text{exprlist} ::= \text{expr} (, \text{exprlist})
\]

3.8.2 Rules

\[
\begin{align*}
C, s, \nu \vdash \text{expr} \Rightarrow \tau & \quad (C, s, \nu \vdash \text{exprlist} \Rightarrow \tau) \\
C, s, \nu \vdash (\text{expr}, \text{exprlist}) \Rightarrow \tau
\end{align*}
\]

3.9 In-Expressions

3.9.1 Grammar

\[
\text{inexpr} ::= \text{id in expr}
\]

3.9.2 Rules

\[
\begin{align*}
id & \notin \text{comptypes} C \\
C, s, \nu \vdash \text{expr} \Rightarrow \text{set}(\tau)
\end{align*}
\]
\[
C, s, \nu \vdash \text{id in expr} \Rightarrow \{\text{id} \mapsto \tau\}
\]

8
3.10 Lists of In-Expressions

3.10.1 Grammar

\[
inexprlist ::= \text{inexpr}(,\text{inexprlist})
\]

\[
3.10.2 \text{Rules}
\]

\[
C, s, \nu \vdash \text{inexpr} \Rightarrow \nu' \quad (C, s, \nu \vdash \text{inexprlist} \Rightarrow \nu'' \quad \text{Dom} \nu' \cap \text{Dom} \nu'' = \emptyset)
\]

\[
C, s, \nu \vdash \text{inexprlist} \Rightarrow \nu'(+\nu'')
\] (27)

3.11 Variables

\[
C, s, \nu \vdash \text{var} \Rightarrow \tau, s'
\]

\[
\frac{\text{var} \in \text{Dom} \nu \quad \nu(\text{var}) \in \text{Clo} C}{C, s, \nu \vdash \text{var} \Rightarrow \nu(\text{var}), \emptyset} \quad (28)
\]

\[
\frac{\text{var} \in \text{vars} C \quad C(\text{var}) \in \text{Clo} C}{C, s', \{\} \vdash \text{var} \Rightarrow C(\text{var}), \emptyset} \quad (29)
\]

\[
\frac{\text{var} \in \text{vars} C(s) \quad \tau = \text{typeof}\text{var in} C(s) \in \text{Clo} C}{C, s, \{\} \vdash \text{var} \Rightarrow \tau, s} \quad (30)
\]

\[
\frac{\text{var} \notin \text{vars} C(s) \quad s < s' \quad C, s', \{\} \vdash \text{var} \Rightarrow \tau, s''}{C, s, \{\} \vdash \text{var} \Rightarrow \tau, s''} \quad (31)
\]

4 Static Semantics of Variable Declarations

A variable declaration binds a variable identifier to a type. Variable declarations appear in global declarations, component-type declarations and in define clauses.

4.1 Global Variables

4.1.1 Grammar

\[
globalvardec ::= \text{global vardec}
\]

\[
4.1.2 \text{Rules}
\]

\[
C \vdash \text{vardec} \Rightarrow \nu \quad \frac{}{C \vdash \text{globalvardec} \Rightarrow \nu} \quad (32)
\]

4.2 Variable Declarations

4.2.1 Grammar

\[
\text{vardec} ::= ty\text{var}(:=\text{expr})
\]
4.2.2 Rules
\[ C \vdash t : \tau \quad (C, s, \nu \vdash e \Rightarrow \tau) \]
\[ C, s, \nu \vdash t \text{ var} \{ := e \} \Rightarrow \{ \text{var} \Rightarrow \tau \} \quad (33) \]

4.3 Define Clauses
\[ C, s, \nu \vdash \text{defineclause} \Rightarrow \nu' \]

4.3.1 Grammar
\[ \text{defineclause} ::= \text{define} \{ \text{tempdeclist} \} \]

4.3.2 Rules
\[ (C, s, \nu \vdash \text{tempdeclist} \Rightarrow \nu') \]
\[ C, s, \nu \vdash \text{define} \{ \} \Rightarrow \{ \} (+\nu') \quad (34) \]

4.4 Temporary Declarations
\[ C, s, \nu \vdash \text{tempdeclist} \Rightarrow \nu' \]

4.4.1 Grammar
\[ \text{tempdeclist} ::= \text{vardec} \text{ (tempdeclist)} \]

4.4.2 Rules
\[ C, s, \nu \vdash \text{vardec} \Rightarrow \nu' \quad (C, s, \nu \vdash \nu' \vdash \text{tempdeclist} \Rightarrow \nu'' \quad \text{Dom} \nu' \cap \text{Dom} \nu'' = \emptyset) \]
\[ C, s, \nu \vdash \text{vardec} \text{ (tempdeclist)} \Rightarrow \nu' (+\nu') \quad (35) \]

5 Static Semantics of Events

5.1 List of Events
\[ C, s \vdash \text{evtlist} \Rightarrow \nu \]

5.1.1 Grammar
\[ \text{evtlist} ::= \text{evt} \text{ (evtlist)} \]

5.1.2 Rules
\[ C, s \vdash \text{evt} \Rightarrow \nu \quad (C, s \vdash \text{evtlist} \Rightarrow \nu' \quad \text{Dom} \nu \cap \text{Dom} \nu' = \emptyset) \]
\[ C, s \vdash \text{evt (evtlist)} \Rightarrow \nu (+\nu') \quad (36) \]
5.2 Events

5.2.1 Grammar

\[ \text{evt} ::= \text{id} \]
\[ \text{var} ::= \text{id} \]
\[ \text{war} ::= \text{id} \left( \text{one} \left( : \text{id}' \right) \right) \]
\[ \text{var} ::= \text{id} \left( \text{all} \right) \]

5.2.2 Rules

\[ \frac{id \in \text{events } C(s) \quad C, s \vdash \text{id} \Rightarrow \{\}} {C, s \vdash \text{id}} \quad (37) \]
\[ \frac{C, s \vdash \text{var} \Rightarrow \tau \quad \tau \in \text{comptypes } C \quad id \in \text{extevs } C(\tau) \quad C, s \vdash \text{war} :: \text{id} \Rightarrow \{\}} {C, s \vdash \text{war} :: \text{id} \Rightarrow \{\}} \quad (38) \]
\[ \frac{C, s \vdash \text{var} \Rightarrow \text{set}(\tau) \quad \tau \in \text{comptypes } C \quad id \in \text{extevs } C(\tau) \quad C, s \vdash \text{var} :: \text{id} \left( \text{one} \left( : \text{id}' \right) \right) \Rightarrow \{\} \left( \text{id}' \rightarrow \tau \right) \} {C, s \vdash \text{var} :: \text{id} \left( \text{one} \left( : \text{id}' \right) \right) \Rightarrow \{\} \left( \text{id}' \rightarrow \tau \right) \}} \quad (39) \]
\[ \frac{C, s \vdash \text{var} \Rightarrow \text{set}(\tau) \quad \tau \in \text{comptypes } C \quad id \in \text{extevs } C(\tau) \quad C, s \vdash \text{var} :: \text{id} \Rightarrow \{\}} {C, s \vdash \text{var} :: \text{id} \Rightarrow \{\}} \quad (40) \]

References


