The specification of PLAN[1]

Pankaj Kakkar

July 25, 2001

[1] This work was supported by DARPA under Contract N66001-96-C-852.

Contents

Chapter 1

Introduction

This document defines PLAN. It is intended to be of use to people interested in implementing PLAN, and together with [2], serves as a reference for exact conformity between different implementations.

The first part of this standard specifies the basis of the language, in three chapters that individually define the syntax, typing rules and evaluation semantics. The second part, in the appendices, defines extensions to the basis through library functions and predefined exceptions.

This document is modeled after [1], I have borrowed most of the style and methodology from that book.

Chapter 2

Syntax specification

2.1 Lexical analysis

PLAN is case sensitive. The following define lexical classes; Anything that does not fall into one of these classes should generate an error before parsing starts.

Reserved words

The following are *reserved words*. They may not be used as identifiers in PLAN programs.

```
OnRemote  OnNeighbor  OnNeighbour  foldl  foldr  unit   int
char    string    bool    host   blob  list  exn  dev  chunk
val  fun   exception  raise  try  handle  if  then   else
let  in  end  =  *  /  +  -  ::  <>  >  >=  < <=  |  and  or
~  not   hd   tl  fst   snd  ^  %  <<  >>  xori andi  ori
noti  explode   implode  ord  chr  void  svc  =>  -> (  )  :
[  ]  ,  ;
```

Literal constants

A non-empty sequence of decimal digits 0,...,9 such that the first digit is not 0 is an *integer constant*. 0 itself is also an *integer constant*. A `true` or `false` is a *boolean constant*. A `[]` is a *list constant*.

A *string constant* is a sequence, enclosed between quotes (") of 0 or more Printable characters (ASCII numbers 33-126), spaces or escape sequences, where the last is one of the following :

`\\`	`\`
`\"`	`"`
`\n`	Newline
`\b`	Backspace
`\r`	Carriage return
`\t`	Horizontal tab
ddd	The character with ASCII number *ddd* (d is a decimal digit)

A *character constant* is any Printable character, space or escape sequence enclosed in single quotes (').

We denote by CON the class of constants, i.e., the *integer, boolean, list, string, unit* and *character* constants defined above. We'll use *c,d,* etc. to range over these.

Identifiers

An *identifier* is a non-empty sequence of the following : a, ... ,z,A, ... ,Z,0, ... ,9, . , _ , ' such that the first character is in a, ... ,z,A, ... ,Z and the resulting sequence is not one of the reserved words.

We denote by IDENT the set of identifiers, and use x, y, z, etc. to range over them.

Comments

The tokens '`(*`' and '`*)`' are used to delimit an arbitrary string of characters. Nested comments are allowed, i.e., each '`*)`' should match with the nearest occurence of '`(*`' upstream in the token list.

2.2 The grammar

The phrase classes are TypeExp, TypeExpList, BaseType, Exp, UnaryOp, BinaryOp, ExpList, TupList, ArgList, Program, DefList, Def, ValDef, FunDef, ExpnDef, SvcDef, ParmDeclList and ParmDecl. The grammer that specifies these classes is described in Figures 1, 2 and 3.

The following conventions are followed when describing the grammar :

- *program* is used to range over Program and so on.
- The brackets $\langle$ and $\rangle$ enclose optional phrases.
- Precedence is lower for alternative forms that appear later.

typeexp	::=	*basetype*
		typeexp `list`
		typeexp `*` *typeexplist*
		`(` *typeexp* `)`
typeexplist	::=	*typeexp* `*` *typeexplist*
		typeexp
basetype	::=	`unit` | `int` | `char` | `string` | `bool` | `host`
		| `blob` | `exn` | `dev` | `chunk`

Figure 2.1: The grammar : Types and type expressions

exp ::= *c*
x
x (⟨ *arglist* ⟩)
`raise` *x*
`try` *exp* `handle` *x* `=>` *exp*
`let` *deflist* `in` *exp* `end`
`if` *exp* `then` *exp* `else` *exp*
unaryop exp
exp binaryop exp
`OnRemote` (*exp*, *exp*, *exp*, *exp*)
`OnNeighbor` (*exp*, *exp*, *exp*, *exp*)
`OnNeighbour` (*exp*, *exp*, *exp*, *exp*)
`foldl` (*exp*, *exp*, *exp*)
`foldr` (*exp*, *exp*, *exp*)
`|` *x* `|` (⟨ *arglist* ⟩)
(*explist*)
[*explist*]
(*tuplist*)
( )

unaryop ::= `~` | `not` | `hd` | `tl` | `fst` | `snd`
| `noti` | `explode` | `implode` | `ord` | `chr`

binaryop ::= `/` | `%` | `*` | `+` | `-`
| `and` | `or` | `<` | `<=` | `>` | `>=` | `=` | `!=`
| `::` | `^` | `<<` | `>>` | `xori` | `andi` | `ori`

explist ::= *exp* ; *explist*
exp

tuplist ::= *exp* , *tuplist*
exp , *exp*

arglist ::= *exp* , *arglist*
exp

Figure 2.2: The grammar : expressions

program	::=	*deflist*
deflist	::=	*def*
		def deflist
def	::=	*valdef*
		fundef
		svcdef
		expndef
valdef	::=	`val` x `:` *typeexp* `=` *exp*
fundef	::=	`fun` x `(` $\langle$ *parmdecllist* $\rangle$ `)` `:` *typeexp* `=` *exp*
expndef	::=	`exception` x
svcdef	::=	`svc` x `:` *typeexp*
parmdecllist	::=	*parmdecl* `,` *parmdecllist*
		parmdecl
parmdecl	::=	x `:` *typeexp*

Figure 2.3: The grammar : Declarations

Chapter 3

Typing PLAN

3.1 Semantic objects

Figure 3.1 defines semantic objects that we will use to attach types to syntactic objects defined in the previous section.

For each member c of CON, except [], there is a type τ_c which is one of `intT, boolT, charT` or `stringT`.

3.2 Typing rules

A type assertion is a sentence of the form

$$H \vdash phrase \Rightarrow H^{'}$$

where H is a type environment, *phrase* is a member of one of the syntactic classes defined in the previous section, and $H^{'}$ is either a type or a type environment. A *program* is defined to be well typed in a type environment H if there is a type environment $H^{'}$ such that the following sentence

$$H \vdash program \Rightarrow H^{'}$$

is true.

This section defines a set of typing rules that can be used to contruct a proof, given a program, that the program is well typed.

Operations

Figure 3.2 defines the types for some operations. The rest will be dealt with individually along with expressions.

$$
\begin{array}{rcl}
\tau & \in & \mathrm{Type} = \mathrm{BaseType} \cup \mathrm{ListType} \cup \mathrm{TupleType} \\
 & & \mathrm{BaseType} = \{\ \texttt{unitT},\ \texttt{intT},\ \texttt{charT},\ \texttt{stringT},\ \texttt{boolT},\ \texttt{hostT}, \\
 & & \texttt{blobT},\ \texttt{exnT},\ \texttt{devT},\ \texttt{chunkT},\ \texttt{voidT}\ \} \\
\tau\ \texttt{listT} & \in & \mathrm{ListType} = \mathrm{Type} \times \{\ \texttt{listT}\ \} \\
\tau_1 * \cdots * \tau_k & \in & \mathrm{Type}^k \\
 & & \mathrm{TupleType} = \cup_{k>1}\ \mathrm{Type}^k \\
\tau \to \tau & \in & \mathrm{FunType} = \mathrm{Type} \times \mathrm{Type} \\
\sigma & \in & \mathrm{AllType} = \mathrm{Type} \cup \mathrm{FunType} \\
H & \in & \mathrm{TyEnv} = \mathrm{IDENT} \overset{\mathrm{fin}}{\to} \mathrm{AllType}
\end{array}
$$

Figure 3.1: Semantic objects for typing rules

`~`	`intT` → `intT`
`not`	`boolT` → `boolT`
`hd`	τ `listT` → τ
`tl`	τ `listT` → τ `listT`
`noti`	`intT` → `intT`
`explode`	`stringT` → `charT listT`
`implode`	`charT listT` → `stringT`
`ord`	`charT` → `intT`
`chr`	`intT` → `charT`
`/`	`intT * intT` → `intT`
`%`	`intT * intT` → `intT`
`*`	`intT * intT` → `intT`
`+`	`intT * intT` → `intT`
`-`	`intT * intT` → `intT`
`and`	`boolT * boolT` → `boolT`
`or`	`boolT * boolT` → `boolT`
`^`	`stringT * stringT` → `stringT`
`<<`	`intT * intT` → `intT`
`>>`	`intT * intT` → `intT`
`xori`	`intT * intT` → `intT`
`andi`	`intT * intT` → `intT`
`ori`	`intT * intT` → `intT`

Figure 3.2: Types for basic operations

Type Expressions For each *basetype*, $H \vdash basetype \Rightarrow basetypeT$ ($basetypeT$ is a member of the semantic object BaseType).

$$\frac{H \vdash typeexp \Rightarrow \tau}{H \vdash typeexp\ \texttt{list} \Rightarrow \tau\ \texttt{listT}} \tag{3.1}$$

$$\frac{H \vdash typeexp_1 \Rightarrow \tau_1, \ldots, H \vdash typeexp_k \Rightarrow \tau_k \quad k \geq 2}{H \vdash typeexp_1 * \cdots * typeexp_k \Rightarrow \tau_1 * \cdots * \tau_k} \tag{3.2}$$

Expressions

$$\frac{c \neq []}{H \vdash c \Rightarrow \tau_c} \tag{3.3}$$

$$\frac{}{H \vdash [] \Rightarrow \tau\ \texttt{listT}} \tag{3.4}$$

$$\frac{}{H \vdash () \Rightarrow \texttt{unitT}} \tag{3.5}$$

$$\frac{H(x) = \tau}{H \vdash x \Rightarrow \tau} \tag{3.6}$$

$$\frac{H \vdash deflist \Rightarrow H' \quad H \circ H' \vdash exp \Rightarrow \tau}{H \vdash \texttt{let}\ deflist\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau} \tag{3.7}$$

$$\frac{H \vdash unaryop \Rightarrow \tau \rightarrow \tau' \quad H \vdash exp \Rightarrow \tau}{H \vdash unaryop\ exp \Rightarrow \tau'} \tag{3.8}$$

$$\frac{H \vdash binaryop \Rightarrow \tau * \tau' \rightarrow \tau'' \quad H \vdash exp \Rightarrow \tau \quad H \vdash exp' \Rightarrow \tau'}{H \vdash exp\ binaryop\ exp' \Rightarrow \tau''} \tag{3.9}$$

$$\frac{H(x) = \tau_1 * \cdots * \tau_k \to \tau \quad H \vdash exp_1 \Rightarrow \tau_1, \ldots, H \vdash exp_k \Rightarrow \tau_k \quad k \geq 2}{H \vdash x(exp_1, \ldots exp_k) \Rightarrow \tau} \tag{3.10}$$

$$\frac{H(x) = \tau' \to \tau \quad H \vdash exp \Rightarrow \tau'}{H \vdash x(exp) \Rightarrow \tau} \tag{3.11}$$

$$\frac{H(x) = \texttt{voidT} \to \tau}{H \vdash x() \Rightarrow \tau} \tag{3.12}$$

$$\frac{H(x) = \tau_1 * \cdots * \tau_k \to \tau \quad H \vdash exp_1 \Rightarrow \tau_1, \ldots, H \vdash exp_k \Rightarrow \tau_k \quad k \geq 2}{H \vdash |x|(exp_1, \ldots exp_k) \Rightarrow \tau \ \texttt{chunkT}} \tag{3.13}$$

$$\frac{H(x) = \tau' \to \tau \quad H \vdash exp \Rightarrow \tau'}{H \vdash |x|(exp) \Rightarrow \tau \ \texttt{chunkT}} \tag{3.14}$$

$$\frac{H(x) = \texttt{voidT} \to \tau}{H \vdash |x|() \Rightarrow \tau \ \texttt{chunkT}} \tag{3.15}$$

$$\frac{H(x) = \texttt{exn}}{H \vdash \texttt{raise}\ x \Rightarrow \tau} \tag{3.16}$$

$$\frac{H \vdash exp_1 \Rightarrow \tau \quad H \vdash exp_2 \Rightarrow \tau \quad H(x) = \texttt{exnT}}{H \vdash \texttt{try}\ exp_1\ \texttt{handle}\ x{=}{>}exp_2 \Rightarrow \tau} \tag{3.17}$$

$$\frac{H \vdash exp_1 \Rightarrow \texttt{boolT} \quad H \vdash exp_2 \Rightarrow \tau \quad H \vdash exp_3 \Rightarrow \tau}{H \vdash \texttt{if}\ exp_1\ \texttt{then}\ exp_2\ \texttt{else}\ exp_3 \Rightarrow \tau} \tag{3.18}$$

$$\frac{H \vdash exp_1 \Rightarrow \tau_1, \ldots, H \vdash exp_k \Rightarrow \tau_k}{H \vdash (exp_1; \ldots; exp_k) \Rightarrow \tau_k} \tag{3.19}$$

$$\frac{H \vdash exp_1 \Rightarrow \tau, \ldots, H \vdash exp_k \Rightarrow \tau}{H \vdash [exp_1; \ldots; exp_k] \Rightarrow \tau\ \texttt{listT}} \tag{3.20}$$

$$\frac{H \vdash exp_1 \Rightarrow \tau_1, \ldots, H \vdash exp_k \Rightarrow \tau_k}{H \vdash (exp_1, \ldots, exp_k) \Rightarrow \tau_1 * \cdots * \tau_k} \tag{3.21}$$

$$\frac{\begin{array}{c} H \vdash exp_1 \Rightarrow \tau \ \texttt{chunkT} \quad H \vdash exp_2 \Rightarrow \texttt{hostT} \\ H \vdash exp_3 \Rightarrow \texttt{intT} \quad H \vdash exp_4 \Rightarrow \texttt{hostT} \rightarrow \texttt{hostT} \end{array}}{H \vdash \texttt{OnRemote}(exp_1, exp_2, exp_3, exp_4) \Rightarrow \texttt{unitT}} \tag{3.22}$$

$$\frac{\begin{array}{c} H \vdash exp_1 \Rightarrow \tau \ \texttt{chunkT} \quad H \vdash exp_2 \Rightarrow \texttt{hostT} \\ H \vdash exp_3 \Rightarrow \texttt{intT} \quad H \vdash exp_4 \Rightarrow \texttt{devT} \end{array}}{H \vdash \texttt{OnNeighbor}(exp_1, exp_2, exp_3, exp_4) \Rightarrow \texttt{unitT}} \tag{3.23}$$

$$\frac{\begin{array}{c} H \vdash exp_1 \Rightarrow \tau \ \texttt{chunkT} \quad H \vdash exp_2 \Rightarrow \texttt{hostT} \\ H \vdash exp_3 \Rightarrow \texttt{intT} \quad H \vdash exp_4 \Rightarrow \texttt{devT} \end{array}}{H \vdash \texttt{OnNeighbour}(exp_1, exp_2, exp_3, exp_4) \Rightarrow \texttt{unitT}} \tag{3.24}$$

$$\frac{H \vdash exp_1 \Rightarrow \tau * \tau^{'} \rightarrow \tau \quad H \vdash exp_2 \Rightarrow \tau \quad H \vdash exp_3 \Rightarrow \tau^{'}\texttt{listT}}{H \vdash \texttt{foldl}(exp_1, exp_2, exp_3) \Rightarrow \tau} \tag{3.25}$$

$$\frac{H \vdash exp_1 \Rightarrow \tau^{'} * \tau \rightarrow \tau \quad H \vdash exp_2 \Rightarrow \tau^{'}\texttt{listT} \quad H \vdash exp_3 \Rightarrow \tau}{H \vdash \texttt{foldr}(exp_1, exp_2, exp_3) \Rightarrow \tau} \tag{3.26}$$

Comment: In rule 3.7, and elsewhere that it is used, $\circ$ means composition of environments; the second argument shadows bindings from the first.

Declarations

$$\frac{H \vdash exp \Rightarrow \tau \quad H \vdash typeexp \Rightarrow \tau}{H \vdash \texttt{val}\ x : typeexp \ = \ exp \Rightarrow x \mapsto \tau} \tag{3.27}$$

$$\frac{H \vdash t_1 \Rightarrow \tau_1, \ldots, H \vdash t_k \Rightarrow \tau_k \quad H \vdash typeexp \Rightarrow \tau \quad H \vdash exp \Rightarrow \tau}{H \vdash \texttt{fun}\ x(x_1 : t_1, \ldots, x_k : t_k) : typeexp = exp \Rightarrow x \mapsto \tau_1 * \cdots * \tau_k \to \tau} \tag{3.28}$$

$$\frac{H \vdash typeexp^{'} \Rightarrow \tau^{'} \quad H \vdash typeexp \Rightarrow \tau \quad H \vdash exp \Rightarrow \tau}{H \vdash \texttt{fun}\ x(x' : typeexp^{'}) : typeexp = exp \Rightarrow x \mapsto \tau' \to \tau} \tag{3.29}$$

$$\frac{H \vdash typeexp \Rightarrow \tau \quad H \vdash exp \Rightarrow \tau}{H \vdash \texttt{fun}\ x() : typeexp = exp \Rightarrow x \mapsto \texttt{voidT} \to \tau} \tag{3.30}$$

$$\frac{}{H \vdash \texttt{exception}\ x \Rightarrow x \mapsto \texttt{exn}} \tag{3.31}$$

$$\frac{H \vdash typeexp \Rightarrow \tau}{H \vdash \texttt{svc}\ x : typeexp \Rightarrow x \mapsto \tau} \tag{3.32}$$

$$\frac{H \vdash def_1 \Rightarrow H^{'}, \ldots, H \circ H_1 \circ \ldots \circ H_{k-1} \vdash def_k \Rightarrow H_k}{H \vdash def_1 \cdots def_k \Rightarrow H \circ H_1 \circ \ldots \circ H_k} \tag{3.33}$$

Comment: The notation $x \mapsto \tau$ represents a singleton type environment with a mapping from x to τ.

Chapter 4

Evaluation semantics

4.1 Semantic objects

For each $c \in$ CON, there is a val_c that is the concrete value represented by the syntactic term. Thus for *integer constants*, it is the integer represented by the sequence of digits in the normal arithmetic sense, and so on.

We also assume a function called *Apply* that defines the application of unary and binary ops to the values corresponding to elements of CON.

Figure 4.1 defines semantic objects that will be used to define the evaluation semantics, apart from the ones already defined in the previous chapter.

We also define a set MSG of messages as

$$\text{msg} \quad ::= \quad \texttt{emit}\ (\textit{dev, packet})$$

and a set DUMP of dumps (activation frames) as

$$\begin{array}{rcl} D & ::= & \textbf{nil} \\ & & (S, E, C, D) \end{array}$$

We associate an environment E_h with each host h. Also, there is a function called $\textit{SvcApply}_h$ that applies services to values and returns a value.

4.2 Abstract machine

This section defines an abstract machine that will be used to evaluate PLAN programs and expressions. The machine is a variation on the well known SECD machine.

A state of the machine is the tuple $(S, (E, E_P), C, D, M, rb)$. The transition rules in figures 4.2, 4.3, 4.4, 4.5, 4.6 and 4.7 define the abstract machine. The definitions use shorter syntax wherever possible; in particular, E_P is only used in rule 4.19 and M and rb are only used in rules 4.35 and 4.37. All other rules omit these elements of the state of the machine. The definitions also abuse the

$$
\begin{array}{rcl}
rb,\ i & \in & \text{Int (the set of integers)} \\
v, val & \in & \text{Val} = \text{BaseVal} \cup \text{TupleVal} \cup \text{ListVal} \cup \text{ChunkVal} \cup \text{ExnVal} \\
 & & \text{BaseVal} = \{ val_c \mid c \in \text{CON} \} \cup \text{Host} \cup \text{Dev} \\
h & \in & \text{Host (finite set of hosts in the network; type } \texttt{hostT}\text{)} \\
dev,\ (h_1, h_2) & \in & \text{Dev} = \text{Host} \times \text{Host} \\
(v_1, \ldots, v_k) & \in & \text{Val}^k \\
 & & \text{TupleVal} = \cup_{k>1} \text{Val}_k \\
[v_1; \ldots; v_k] & \in & \text{ListVal} = \{\texttt{List}\} \times \text{Val} \\
 & & \text{ExnVal} = \{\texttt{Exn}\} \times \text{IDENT} \\
ch & \in & \text{ChunkVal} = \text{IDENT} \times \cup_{k \geq 0} \text{Val}^k \times \text{Env} \\
f & \in & \text{FnVal} = \cup_{k \geq 0} \text{IDENT}^k \times \text{Exp} \times \text{Env} \\
svc & \in & \text{SvcVal} = \{\texttt{Svc}\} \times \text{IDENT} \\
 & & \text{AllVal} = \text{Val} \cup \text{FnVal} \cup \text{SvcVal} \\
rf & \in & \text{RoutFun} : \text{Host} \rightarrow (\text{Dev} \times \text{Host}) \\
\text{E} & \in & \text{Env} = \text{IDENT} \stackrel{\text{fin}}{\rightarrow} (\text{AllVal} \times \text{Type}) \\
packet & \in & \text{ChunkVal} \times \text{Host} \times \text{IntVal} \times \text{RoutFun} \\
C & \in & \text{Code}^* \\
 & & \text{Code} = \text{Exp} \cup \text{Program} \cup \text{Unaryop} \cup \text{Binaryop} \cup \\
 & & \{\mathbf{ap}_i : i \geq 0\} \cup \{\mathbf{ch}_i : i \geq 0\} \cup \{\mathbf{tup}_i : i \geq 0\} \cup \{\mathbf{list}_i : i \geq 0\} \cup \\
 & & \{\mathbf{if}, \mathbf{foldl}, \mathbf{foldr}, \mathbf{remote}_1, \mathbf{remote}_2, \mathbf{neighbor}, \mathbf{bind}\ (\tau)\} \\
S & \in & \text{Stack}^* \\
 & & \text{Stack} = (\text{AllVal} \times \text{Type}) \cup \{\mathbf{raised}\ \text{x}\}
\end{array}
$$

Figure 4.1: Semantic objects for evaluation semantics

syntax of programs to write $exp_1, \ldots, exp_k$ instead of a *tuplist* and similarly for *arglist* etc.

The following functions help to start computations on the machine and unload the results once the computation is finished:

getEnv : Program $\rightarrow$ Env
getEnv (*program*) = E
iff $(\mathbf{nil}, (\phi, \phi), program, \mathbf{nil}, \{\}, 0) \longrightarrow^* (\mathbf{nil}, (E, E'), \mathbf{nil}, \mathbf{nil}, \{\}, 0)$

getVal : Exp $\rightarrow$ Val
getVal $(program, exp, E) = val$
iff $(\mathbf{nil}, (E, E), \texttt{let}\ program\ \texttt{in}\ exp\ \texttt{end}, \mathbf{nil}, \{\}, 0) \longrightarrow^* ((val : \tau) :: \mathbf{nil}, (E, E), \mathbf{nil}, \mathbf{nil}, \{\}, 0)$

evalPacket : Packet $\rightarrow$ Val $\times$ MSG*
evalPacket $((x, (val_1, \ldots, val_k), E_P), h, rb, x) = (val, M)$
iff $(\mathbf{nil}, (E_P \circ E_h, E_P), x(val_1, \ldots, val_k) :: \mathbf{nil}, \mathbf{nil}, \{\}, rb)$
$\longrightarrow^* ((val : \tau) :: \mathbf{nil}, (E, E_P), \mathbf{nil}, \mathbf{nil}, M, rb')$

4.3 Reduction rules

Reduction rules are of the form

$$msg_1, \ldots, msg_k \mapsto \{msg_1^{'}, \ldots, msg_l^{'}\} \ (l \geq 0, k > 0)$$

and the effect is to remove msg_1 etc. from the multiset (the "sea" of messages) and add $msg_1^{'}, \ldots, msg_l^{'}$ into the multiset.

$$\begin{aligned}&\texttt{emit}((h,h^{'}),((x,(exp_1,\ldots,exp_k),\rho_P),h^{'},rb,rf)) \mapsto \{msg_1,\ldots,msg_n\}\\ &\quad if\ \rho_{h'} \circ \rho_P, rf, rb-1, \rho_{h'} \vdash x(exp_1,\ldots,exp_k) \Rightarrow val, \{msg_1,\ldots,msg_n\}\end{aligned} \tag{4.42}$$

expln.: Given a message $\texttt{emit}(dev,\ packet)$, if the evaluation destination of *packet* is the same as the host that *dev* belongs to $(h^{'})$, then evaluate the packet and add the set of messages that resulted as side effects of the evaluation to the multiset of messages.

$$\begin{aligned}&\texttt{emit}((h,h^{'}),(ch,h^{''},rb,rf)) \mapsto\\ &\qquad \{\texttt{emit}(dev,(ch,h^{''},rb-1,rf))\} \quad \text{where } dev = rf(h^{'},h^{''})\\ &\qquad\qquad if\ h^{'} \neq h^{''}\end{aligned} \tag{4.43}$$

expln.: On the other hand, if the evaluation destination has not been reached yet, then forward the packet onto the destination by emitting it on the interface given by the routing function.

4.4 Evaluation model

A PLAN computation is started by *injecting* a PLAN *program* into the network along with an *initial invocation*) of the form $x(exp_1, \ldots, exp_k)$. Along with the *program* and the initial invocation, the evaluation destination *h*, initial amount of resource bound *rb*, and a routing function *rf* are also specified. An initial PLAN packet is constructed as follows:

1. The *program* is massaged into an environment E using the function *getEnv* described in section 4.2.
2. The arguments in the initial invocation $(exp_1, \ldots, exp_k)$ are evaluated at the node where injection happened into values $(val_1, \ldots, val_k)$ using the function *getVal* again defined in section 4.2.
3. The packet constructed is

$$((x,(val_1,\ldots,val_k),E),h,rb,rf)$$

Evaluation of PLAN packets is divided into two modes - *stationary* and *transient*. PLAN packets start out in a transient mode, travelling in the network from the *injection point* to the first *evaluation destination*. Once at the eval dest, they change to stationary mode. They may give rise to other PLAN packets that again start out in transient mode and travel to another eval dest.

Transient mode evaluation is represented in the semantics as a multiset of messages. The reduction rules in section 4.3 specify how the multiset changes (through term rewriting).

When a PLAN packet arrives at its eval dest, the *evalPacket* function is used to evaluate the packet in the abstrct machine defined in section 4.2. The result of this evaluation is some messages and either a value, an uncaught exception or a type error. The messages are added to the multiset of messages that represent transient mode evaluation; the second result is discarded if it is a value, and bundled in an error message to the source of the packet in the other two cases.

$$((val : \tau) :: S, E, \mathbf{nil}, (S^{'}, E^{'}, C^{'}, D^{'})) \longrightarrow ((val : \tau) :: S^{'}, E^{'}, C^{'}, D^{'}) \tag{4.1}$$

$$(S, E, c :: C, D) \longrightarrow ((val_c : \tau_c) :: S, E, C, D) \tag{4.2}$$

$$(S, E, x :: C, D) \longrightarrow (E(x) :: S, E, C, D) \tag{4.3}$$

$$(S, E, \texttt{let}\ deflist\ \texttt{in}\ exp\ \texttt{end} :: C, D) \longrightarrow (S, E, deflist :: exp :: C, D) \tag{4.4}$$

$$(S, E, unaryop\ exp :: C, D) \longrightarrow (S, E, exp :: unaryop :: C, D) \tag{4.5}$$

$$((val : \tau) :: S, E, unaryop :: C, D) \longrightarrow (Apply(unaryop, val) :: S, E, C, D) \tag{4.6}$$

$$\begin{aligned}&(S, E, exp_1\ \ bnaryop\ \ exp_2 :: C, D)\\ &\quad \longrightarrow (S, E, exp_1 :: exp_2 :: binaryop :: C, D)\end{aligned} \tag{4.7}$$

$$\begin{aligned}&((val_2 : \tau_2) :: (val_1 : \tau_1) :: S, E, binaryop :: C, D)\\ &\quad \longrightarrow (Apply(binaryop, val_1, val_2) :: S, E, C, D)\end{aligned} \tag{4.8}$$

$$\begin{aligned}&(S, E, \texttt{if}\ exp\ \texttt{then}\ exp_1\ \texttt{else}\ exp_2 :: C, D)\\ &\quad \longrightarrow (S, E, exp :: \mathbf{if} :: exp_1 :: exp_2 :: C, D)\end{aligned} \tag{4.9}$$

$$((\texttt{trueV} : \texttt{boolT}) :: S, E, \mathbf{if} :: exp_1 :: exp_2 :: C, D) \longrightarrow (S, E, exp_1 :: C, D) \tag{4.10}$$

$$((\texttt{falseV} : \texttt{boolT}) :: S, E, \mathbf{if} :: exp_1 :: exp_2 :: C, D) \longrightarrow (S, E, exp_2 :: C, D) \tag{4.11}$$

$$(S, E, (exp_1; \ldots ; exp_k) :: C, D) \longrightarrow (\mathbf{nil}, E, exp_1 :: \cdots :: exp_k, (S, E, C, D)) \tag{4.12}$$

$$(S, E, (exp_1, \ldots , exp_k) :: C, D) \longrightarrow (S, E, exp_1 :: \cdots :: exp_k :: \mathbf{tup}_k :: C, D) \tag{4.13}$$

$$\begin{aligned}&((val_k : \tau_k) :: \cdots :: (val_1 : \tau_1) :: S, E, \mathbf{tup}_k :: C, D)\\ &\quad \longrightarrow (((val_1, \ldots , val_k) : \tau_1 * \cdots * \tau_k) :: S, E, C, D)\end{aligned} \tag{4.14}$$

Figure 4.2: Basic Expressions

$$(S, E, x(exp_1, \ldots, exp_k) :: C, D)$$
$$\longrightarrow (S, E, exp_1 :: \cdots :: exp_k :: x :: \mathbf{ap}_k :: C, D) \tag{4.15}$$

$$((((x_1, \ldots, x_k), exp, E^{'}) : (\tau_1 * \cdots * \tau_k \to \tau)) :: (val_k : \tau_k) :: \cdots :: (val_1 : \tau_1) :: S, E, \mathbf{ap}_k :: C, D)$$
$$\longrightarrow (\mathbf{nil}, E^{'}[(val_1 : \tau_1)/x_1] \cdots [(val_k : \tau_k)/x_k], exp, (S, E, C, D)) \tag{4.16}$$

$$((\mathtt{Svc}\ x : (\tau_1 * \cdots * \tau_k \to \tau)) :: (val_1 : \tau_1) :: \cdots :: (val_k : \tau_k) :: S, E, \mathbf{ap}_k :: C, D)$$
$$\longrightarrow (SvcApply_h(x, val_k, \ldots, val_1) :: S, E, C, D) \tag{4.17}$$

$$(S, E, |x|(exp_1, \ldots, exp_k), D)$$
$$\longrightarrow (S, E, exp_1 :: \cdots :: exp_k :: \mathbf{ch}_k :: x :: C, D) \tag{4.18}$$

$$((val_k : \tau_k) :: \cdots :: (val_1 : \tau_1) :: S, E, \mathbf{ch}_k :: x :: C, D)$$
$$\longrightarrow ((x, (val_1, \ldots, val_k), E_P) :: S, E, C, D) \tag{4.19}$$

$$\tag{4.20}$$

Figure 4.3: Functions, services and chunks

$$(S, E, \mathtt{try}\ exp\ \mathtt{handle}\ x => exp^{'}, D)$$
$$\longrightarrow (\mathbf{nil}, E, exp, (S, E, \mathbf{handle}\ (x, exp^{'}) :: C, D)) \tag{4.21}$$

$$(val :: S, E, \mathbf{handle}\ (x, exp) :: C, D) \longrightarrow (val :: S, E, C, D) \tag{4.22}$$

$$(S, E, \mathtt{raise}\ x :: C, (S^{'}, E^{'}, C^{'}, D^{'})) \longrightarrow (\mathbf{raised}\ x :: S^{'}, E^{'}, C^{'}, D^{'}) \tag{4.23}$$

$$(\mathbf{raised}\ x :: S, E, \mathbf{handle}\ (x, exp) :: C, D) \longrightarrow (S, E, exp, D) \tag{4.24}$$

$$(\mathbf{raised}\ x :: S, E, \mathbf{handle}\ (y, exp) :: C, D) \longrightarrow (S, E[(\mathtt{Exn}\ x : \mathtt{exnT})/y], exp, D) \tag{4.25}$$

$$(\mathbf{raised}\ x :: S, E, C, (S^{'}, E^{'}, C^{'}, D^{'})) \longrightarrow (\mathbf{raised}\ x :: S^{'}, E^{'}, C^{'}, D^{'}) \tag{4.26}$$

Figure 4.4: Exception handling

$$(S, E, [exp_1; \ldots; exp_k] :: C, D) \longrightarrow (S, E, exp_1 :: \cdots :: exp_k :: \mathbf{list}_k :: C, D) \tag{4.27}$$

$$\begin{aligned}&((val_k : \tau) :: \cdots :: (val_1 : \tau) :: S, E, \mathbf{list}_k :: C, D) \\ &\quad \longrightarrow (([val_1; \ldots; val_k] : \tau\ \texttt{listT}) :: S, E, C, D)\end{aligned} \tag{4.28}$$

$$(S, E, \texttt{foldl}\ (x, exp_{base}, exp_l) :: C, D) \longrightarrow (S, E, exp_l :: exp_{base} :: \mathbf{foldl} :: x :: C, D) \tag{4.29}$$

$$\begin{aligned}&(val_{base} :: [val_1; \ldots; val_k] :: S, E, \mathbf{foldl} :: x :: C, D) \\ &\quad \longrightarrow (S, E, x(\ldots x(val_{base}, val_1), \ldots, val_k) :: C, D)\end{aligned} \tag{4.30}$$

$$(S, E, \texttt{foldr}\ (x, exp_l, exp_{base}) :: C, D) \longrightarrow (S, E, exp_l :: exp_{base} :: \mathbf{foldr} :: x :: C, D) \tag{4.31}$$

$$\begin{aligned}&(val_{base} :: [val_1; \ldots; val_k] :: S, E, \mathbf{foldr} :: x :: C, D) \\ &\quad \longrightarrow (S, E, x(val_1, \ldots, x(val_k, val_{base}) \ldots) :: C, D)\end{aligned} \tag{4.32}$$

Figure 4.5: Lists

$$(S, E, \texttt{OnRemote}\ (exp_1, exp_2, exp_3, x) :: C, D)$$
$$\longrightarrow (S, E, exp_1 :: exp_2 :: exp_3 :: \mathbf{remote}_1 :: x :: C, D) \tag{4.33}$$

$$((val_{rb} : \texttt{intT}) :: (val_h : \texttt{hostT}) :: (val_{ch} : \texttt{chunkT}) :: S, E, \mathbf{remote}_1 :: x :: C, D)$$
$$\longrightarrow ((val_h : \texttt{hostT}) :: (val_{rb} : \texttt{intT}) :: (val_{ch} : \texttt{chunkT}) :: S, E, x :: \mathbf{ap}_1 :: \mathbf{remote}_2 :: C, D) \tag{4.34}$$

$$(((val_d : \texttt{devT}), (val_{nh} : \texttt{hostT})) :: (val_{rb} : \texttt{intT}) :: (val_{ch} : \texttt{chunkT}) :: S, E, \mathbf{remote}_2 :: C, D, M, rb)$$
$$\longrightarrow (\texttt{unitV} :: S, E, C, D,$$
$$M \cup \texttt{emit}(val_d, (val_{ch}, val_h, val_{rb}, x)), rb - \texttt{intVal}(val_{rb})) \tag{4.35}$$

$$(S, E, \texttt{OnNeighbor}\ (exp_1, exp_2, exp_3, exp_4) :: C, D)$$
$$\longrightarrow (S, E, exp_1 :: exp_2 :: exp_3 :: exp_4 :: \mathbf{neighbor} :: C, D) \tag{4.36}$$

$$((val_d : \texttt{devT}) :: (val_{rb} : \texttt{intT}) :: (val_h : \texttt{hostT}) :: (val_{ch} : \texttt{chunkT}) :: S, E, \mathbf{neighbor} :: C, D, M, rb)$$
$$\longrightarrow (\texttt{unitV} :: S, E, C, D,$$
$$M \cup \texttt{emit}(val_d, (val_{ch}, val_h, val_{rb}, \phi)), rb - \texttt{intVal}(val_{rb})) \tag{4.37}$$

Figure 4.6: Message transmission

$$((val : \tau) :: S, E, \mathbf{bind}(\tau) :: x :: C, D) \longrightarrow (S, E[val : \tau/x], C, D) \tag{4.38}$$

$$(S, E, \texttt{val}\ x : typeexp\ =\ exp :: C, D) \longrightarrow (S, E, exp :: \mathbf{bind}(\tau) :: x :: C, D) \tag{4.39}$$

$$(S, E, \texttt{fun}\ x(x_1 : typeexp_1, \ldots, x_k : typeexp_k) : typeexp\ =\ exp :: C, D)$$
$$\longrightarrow (S, E[(((x_1, \ldots, x_k), exp, E) : (\tau_1 * \cdots * \tau_k) \to \tau)/x], C, D) \tag{4.40}$$

$$(S, E, \texttt{exception}\ x :: C, D) \longrightarrow (S, E[(\texttt{Exn}\ x : \texttt{exnT})/x], C, D] \tag{4.41}$$

Figure 4.7: Declarations

Chapter 5

Notes

The preceding chapters specify almost a complete language. However, a few ambiguities have been left deliberately, these are noted here.

Type checking: dynamic v/s static While the separation of typing rules from evaluation semantics implies that type checking in PLAN is done statically, this is not necessarily true; in fact, the Penn implementation does type checking while executing a program. However, the separation does show that static type checking is possible. Implementations of PLAN at Penn or elsewhere, might include tools to statically type check PLAN programs.

Notice however that the freedom to type check either way means that the result of injecting a PLAN packet that contains a type error into the network is not defined.

Evaluation: source v/s byte code The evaluation semantics specify only the meaning of a program and not how the program is executed. The program might be interpreted from source, or compiled into an intermediate language program which is then interpreted, as long as it can be shown that the translation from the PLAN source into the intermediate language together with the semantics of the intermediate language obeys the evaluation semantics specified in this document.

Appendix A

Exceptions

The following exceptions are available to a PLAN program to raise without being explicitly defined. These may also be raised by the interpreter, as specified:

- *ServiceNotPresent* is raised when a service call is attempted for which the service is not resident on the active node. Note that because service calls are syntactically identical to function calls, the evaluator assumes that *any* unbound function name in an application is a service, thus raising this exception.
- *DivByZero* is raised upon an attempted division by zero.
- *Hd* is raised when `hd` is applied to an empty list.
- *Tl* is raised when `tl` is applied to an empty list.
- *NotEnoughRB* is raised when the current packet runs out of resource bound, or if it attempts to allocate to a child packet more resource bound than it has available.
- *HostNotLocal* is raised when a router is asked to forward a packet to a node which is not one of its link-layer connections.
- *NoRouteEntry* is raised when the invoked routing function does not know of a route for the specified destination node.

Appendix B

Services

The service functions form a library that must be included with a standard PLAN implementation. The following specifies the names of the services that must be included, together with the required types and a short description of each:

- $getRB : () \rightarrow int$
  Returns the current amout of resource bound held by the packet.

- $getSource : () \rightarrow host$
  Returns the node specified in the packet's source field. Assuming that nobody has tampered with the packet on the way to the current node, this is the node that the packet's oldest ancestor (which may be the packet itself) originated from.

- $getSrc : () \rightarrow host$
  Same as $getSource$.

- $thisHost : () \rightarrow hostlist$
  Returns a list of all network addresses for devices on the current node.

- $thisHost : dev \rightarrow host$
  Returns the network address which corresponds to the given device. If the argument is not a proper device, the exception *InvalidDevice* would be raised.

- $thisHostIs : host \rightarrow bool$
  Returns *true* if the given address matches any of the addresses of interfaces on the current node.

- $gettimeofday : () \rightarrow int \times int$
  Return the current time since 00:00:00 GMT, Jan. 1, 1970, as a (seconds, microseconds) pair.

- *timeStamp* : () $\rightarrow$ *string*
  Returns the current time of day as a string.

- *length* : α *list* $\rightarrow$ *int*
  Returns the length of the given list.

- *member* : (α, α *list*) $\rightarrow$ *bool*
  Checks whether the given element is in the list or not.

- *remove* : (α, α *list*) $\rightarrow$ α *list*
  Removes the specified element from the given list.

- *staticRoute* : *host* $\rightarrow$ *host* $\times$ *dev*
  PLAN provides a static routing system, based on local routing tables read in from files at startup. This service returns the nexthop and interface from a specified destination according to the static routing table. The format of the static routing table file is given by the following:

```
let
  X = a single integer character
  Int = an integer
  Id = an identifier
in
  IP_ADDR = XXX.XXX.XXX.XXX
  DNS_NAME = Id(.Id)*
  ADDR = (IP_ADDR|DNS_NAME)
       |  (IP_ADDR|DNS_NAME)':'Int
  ADDRDEV = ADDR'/'Id
  DEV = '/'Id
  DEFAULT = "default"
end

GRAMMAR:
File := Entry-list
Entry-list := Entry | Entry Entry-list
Entry := Dest Hop
Dest := ADDR | DEFAULT
Hop := ADDR | ADDRDEV
```

- *getStaticRoutes* : *unit* $\rightarrow$ (*activehost* $\times$ *activehost* $\times$ *dev*)*list*
  Returns the static routing table as a list of three tuples; the first element of the tuple is the destination, and the next two elements are the next hop given as the host and the device to send through to get there. The default route will have destination *any_addr*.

- *getHostByName* : *string* $\rightarrow$ *host*
  Takes as input a *name*, according to the following grammar:

$$
\begin{array}{lcl}
\textit{name} & ::= & \textit{host} \mid \textit{host} : \textit{port-num} \\
\textit{host} & ::= & \textit{ip-addr} \mid \textit{domain-name} \\
\textit{port-num} & ::= & \textit{int-literal} \\
\textit{domain-name} & ::= & \textit{string-literal}
\end{array}
$$

where *ip-addr* has the form $n.n.n.n$ where each n is an integer from 0 to 255. If the *port-num* argument is not provided, then it is assumed to be the implicit PLAN port number of the invoking node. *getHostByName* attempts to resolve this *name* into a value of the PLAN *host* type. This mapping is currently determined by the PLAN DNS, which obtains mappings from a file indicated at node startup time. Raises an $UnknownHost$ exception on failure.

- *defaultRoute* : $host \rightarrow host \times dev$
  The default routing function. By the default, this is the same as the staticRoute function.

- *getNeighbors* : $() \rightarrow (host \times dev)\ list$
  Using the default routing function's table, this function determines the "neighbors" of the current node; that is, those nodes that are on one of the physical networks to which the current node is attached.

- *print* : $\alpha \rightarrow unit$
  If this call occurs on the node on which this PLAN application originated, then the value will be converted to a string (see `toString`, below) and the output will be made available to the initiating process. The underlying mechanism for print is the PLAN port created by the injecting host program between it and the originating node.

- *getSrcDev* : $() \rightarrow dev$
  Returns the interface on which the packet arrived.

- *getDevs* : $() \rightarrow dev\ list$
  Returns a list of all interfaces on the current node.

- *getMTU* : $dev \rightarrow int$
  Returns the **M**aximum **T**ransmission **U**nit for the physical network that the given device connects to.

- *toString* : $\alpha \rightarrow string$
  Converts the argument (a PLAN value) into a string.

- *eval* : $chunk \rightarrow \alpha$
  Subtracts 1 from the current packet's resource bound, and evaluates the given chunk. The environment is first reinitialized to be the default for the node, and one is subtracted from the packet's resource bound. The result of evaluating the chunk is returned from `eval`. Notice that you could indirectly simulate recursion using this service (which must terminate due to loss of resource bound):

```
fun f (x:int):int =
    ...
    eval (|f| (x-1))
    ...
```

- *evalBlob* : $blob \rightarrow \alpha$
  Same as `eval`, except the argument is of type *blob*. The argument is first cast to a chunk, which results in a type error on failure.

- *zeroRB* : $() \rightarrow unit$
  Sets the resource bound of the packet to zero. This is used for error packets (for example, the ones sent by the `abort` service routine) to ensure that error reporting packets do not spawn off any new packets.

- *abort* : $chunk \rightarrow unit$
  This service routine can be used by the packet to stop executing on detection of unforeseen problems and send an error report to the source. On a call to *abort*, execution of the packet is terminated, and a special packet is created and sent to the source. On arrival at the source, this packet sets its resource bound to zero, and evaluates the given chunk.

  This error reporting packet is given a resource bound of 32. Thus even if a packet has run out of resource bound (and thus cannot even send an error report back normally) it would be able to report the problem to the source.

- *broadcast_addr* : *host* is a value representing a broadcast address. Packets sent to this address will be received by all neighbors of the current node.

- *any_addr* : *host* is a network address representing all interfaces on the current node.

- *loopback_addr* : *host* represents the address of the current node; packets sent to this address will be received by the current node.

Bibliography

[1] R. Milner, M. Tofte, R. Harper and D. MacQueen '*The Definition of Standard ML (Revised)*, The MIT Press, 1997

[2] J.T. Moore, M.W. Hicks and Pankaj Kakkar, "PLAN Programmer's Guide" PLAN 3.0 documentation, October 1998.