The PLAN Tutorial*

for PLAN version 3.2

Michael Hicks and Pankaj Kakkar

July 25, 2001

1 Introduction

The purpose of this document is to provide a short introduction to PLAN and the basic usage of the 3.2 OCaml implementation. It will be of help to people just starting with PLAN, although experienced PLAN programmers might use this as a quick reference.

PLAN 3.2 is implemented in OCaml version 2.02. Previous versions of PLAN (up to 2.22) were written in Java [2] (and an extension, Pizza [6]), which has its own documentation. This document assumes that installation (of both PLAN and its required components) has completed successfully; refer to the Installation document for that.

2 Uses of PLAN

The PLAN distribution will build a number of different executables. Of interest are:

- *pland* : the PLAN daemon implementing an Active Router
- *plan_loader* : the PLAN daemon merged with the Active Loader [1] allowing switchlet-loading capability. This won't be described in this document.
- *inject* : sends a PLAN program to the local pland to be interpreted
- *plan* : a non-network PLAN interpreter in *read-eval-print* mode used to test out basic programs

A number of shellscripts are provided to simplify the use of these programs. Additionally, a number of other executables exist, but they won't be described here.

*This work was supported by DARPA Contract N66001-96-C-852

2.1 Example programs

The examples presented in this tutorial as well as many others may be found in the directories `interp_tests` and `rout_tests` of the distribution. The former contains programs which do not make use of the network primitives (and may thus be run using the non-network interpreter).

3 Starting up an Active Router

During this discussion the terms *Active Router* and *PLAN node*, both of which refer to the `pland` program, will be used interchangeably.

The first step towards being part of an active network is to set up a local machine as an active router. The `pland` program serves as the entry point for the router. For more detailed usage information, please refer to The PLAN Active Router [3] document. Try typing[1]:

```
% bin/pland -help
```

You will get a message about the usage of pland :

```
usage : bin/pland [-router] [-firewall d1,d2,...dn] [-l log]
        [-ip port] [-rf rout_tab_file] [-hf host_file] [-key key_file]
        [-policy policy_file] [-authlist h1,h2,...,hn] ifc_spec_file

  -router Turn on router mode.
  -firewall Specify devices to firewall (default=None)
  -l Set log file.
  -ip Specify local port number (default=3324).
  -rf Specify static route table file.
  -hf Specify host name file (default=EXP_IP_ADDRS).
  -key Specify key file (default=./key.<local portnum>).
  -policy Specify policy specification file (depends on policy engine in use)
  -authlist Specify nodes to authenticate with (default=None)
```

The `-router` option specifies that `pland` should execute in *router mode* which means that no local connections to host applications will be accepted; this should help performance. The `-firewall` option takes a list of comma-separated device names as an argument and specially processes packets received on those devices; see the Security Guide for more on the firewall. The argument given to `-l` indicates the file that all messages should be logged to (if none is specified, the file "log" in the current directory will be used; a "`-`" indicates `stdout`).

The `-ip` option specifies the implicit port number that PLAN will receive local connections on; any program which *injects* packets to `pland` must specify

[1] Note that all executable programs in the PLAN distribution are stored in the `bin` directory in the PLAN directory-tree. These examples assume that you are typing the commands from the `plan` directory.

this port. If no port number is given, a default of 3324 is used. Using different port numbers allows several PLAN nodes in the same PLAN network to reside on the same machine. Only one Active Router can be using any port at any time.

The default is that a RIP-style protocol will be used to dynamically determine the routing tables of the PLAN node. Alternatively, the `-rf` flag can used to specify a file used by this node to form its routing table. The `-hf` flag specifies the name of the file used to resolve hostnames; this file is in the same format as `/etc/hosts`. If no file is specified, `./EXP_IP_ADDRS` is used.

To make use of service-level security, the active router must be able to determine its public and private keys. By default, these keys are read in from a file in the current directory called `key.`*portnum* where *portnum* is the node's implicit port. If this file does not exist, then a default pair of keys is used (essentially meaning that you don't intend to make use of the security features). Additionally, the `-policy` flag is used to specify the node's security policy file, which depends on the policy manager compiled into `pland`. Additionally, security relationships will (attempt to be) estabilshed for the comman-separated list of hosts given to `-authlist`. For this tutorial, we won't be worried about security; for more details, see the PLAN Security Guide [4].

Finally, the only required argument is the interface specification file. This file is used to indicate which link layers `pland` shall run on top of. A sample file will be given below for expositional purposes. More details on the interface file format, as well as a more detailed desription of all of these arguments may be found in [3].

3.1 An Example

Let us construct an active network of two nodes on our machine m (where m is the DNS name of the machine). We will first have to construct two interface files; we'll call the first $m1$ and the second $m2$[2]. The contents of the two files:

$m1$:
`1 ip0 ip` m`:3324` m`:3325`

$m2$:
`1 ip0 ip` m`:3325` m`:3324`

These two files describe a virtual network of two nodes, $m1$ and $m2$. The network is virtual because both nodes will run on the same machine, m. The first file indicates that the node $m1$ has one IP interface named `ip0` with PLAN-network address m`:3324`. The network neighbor of this interface has address m`:3325`. The second file indicates that the node $m2$ has one IP interface named `ip0` (ip device names are arbitrary) with PLAN-network address m`:3325`. The network neighbor of this interface has address m`:3324`.

[2]The actual names of the interface files are not important, we just chose these for convenience.

Before you can start a node, you must add your machine to the DNS resolution table, by default stored in the file `EXP_IP_ADDRS` in the plan directory. This file has the same format as the file `/etc/hosts` (and you can in fact provide `/etc/hosts` as an argument to `-hf` if you choose). Looking at the file, you will see entries like:

```
10.0.1.1 pride.cis.upenn.edu pride
10.0.0.1 wrath.cis.upenn.edu wrath
10.0.0.2 bootsy.cis.upenn.edu bootsy
```

Each line of the file consists of at least two fields, separated by whitespace. The first field is the address of the node, in octet-format, the same as used by IP. If you are running PLAN on top of IP, then you should specify your machine's actual IP address here. The second field is the domain name of the machine. Thus, for our example, we would add the following line to the table:
$w.x.y.z \quad m$
where $w.x.y.z$ is the IP address of the machine you are running on, m. Note that any fields following the name of the machine on the same line are considered aliases, and can be used interchangeably with the name itself to refer to the machine. In particular, any calls to the service `getHostByName` (explained later) will return the same address when given any of the aliases mapped to it.

Typing in a terminal window of m in the `plan` directory[3], we can start node $m1$:

`% bin/pland -l log3324 -ip 3324` $m1$

or alternatively

`% bin/pland -l log3324` $m1$

to make use of the default port setting. In another terminal window, we can start node $m2$:

`% bin/pland -l log3325 -ip 3325` $m2$

Notice that the argument to `-ip` for both nodes matches the port portion of the network address for each node. This is necessary, for reasons we shall see below (and further explained in [3]).

If you have any trouble starting up the nodes as described here, please first refer to the PLAN FAQ, which is present in this distribution as an HTML file. The most up-to-date version can be found at

`http://www.cis.upenn.edu/~switchware/PLAN/FAQ.html`.

[3]`pland` will look for the file `EXP_IP_ADDRS` in the current directory. If you want to run `pland` from a different directory, you will have to specify to `pland` where to find this file by using the `-hf` option. This file must be present and must *at least* contain your machine's IP address mapping.

4 Getting Started

The quintessential first problem solved in any programming language is the "Hello world" program.

4.1 The Hello world program

A PLAN program is a list of definitions, which could be function definitions, exception declarations, or variable bindings (in the ML style). In the case of the Hello world program, there is just one function :

```
svc print : 'a -> unit
svc thisHost : void -> host list
fun doit () =
    (print (thisHost ()); print(" says : Hello world!\n"))
```

4.2 Run the Hello world program : The `inject` program

Before we try and understand the program, let us run it to see if it works!

The `inject` program is provided as a means to inject PLAN programs into the network directly from the command line. Get the usage information as follows:

```
% bin/inject -help
usage : bin/inject [-v] [-val] [-p port] [-ed evalDest] [-rf routFun]
        [-hf host_file] [-o outfile] <codefile>[,codefile2]...] <RB>

  -v Turn on verbose mode.
  -val Unmarshall and print responses as values (rather than strings).
  -p Set default port. (default=3324)
  -ed Set initial evaluation destination.
  -rf Specify routing function.
  -fid Specify flow ID. (default=0)
  -hf Specify file for hostname resolution. (default=EXP_IP_ADDRS)
  -o Specify file to output marshalled packet.  (default=None)
```

The mandatory arguments are (at least one) codefile, and an integer resource bound, or RB. The codefile argument is the name of the file (or files) that contain the PLAN program. The RB argument specifies the initial amount of resource (more on this later) to hand to the packet going into the network.

The `-v` option produces a verbose output. The `-val` option prints output received as the string representations of PLAN values rather than as strings; this is useful for printing the results of the `deliver` service (see the Programmer's Guide). The `-p` option allows you to specify the PLAN port number of the router to receive the message. This must match the number given to `-ip` for some `pland` running on the local machine. This also serves to set the port portion of the identity should the `thisHost` local service routine be invoked during the initial invocation (see the ping example below).[4] If no `-p` is given, the default is 3324. The options `-ed` and `-rf` specify the *evaluation destination* and *routing function* of the packet you are injecting. By default, the evaluation destination (that is, the node on which the program should evaluate) is set to the local `pland`[5]. For this reason, the default is for this field (that is, the function that is used to help the packet get to its evaluation destination) to be empty. Should you decide that the first location that your program should execute is *not* the local node, you would indicate where (and how to get there) with these options. The flow id is set with `-fid`; this option isn't important for now. The `-hf` option is used to specify the hostname file. This file is used by the `getHostByName` function to map domain names to network addresses. Finally, `-o` may be used to write the resulting PLAN packet to a specified file rather than to the local interpreter. This may be useful for 'pre-creating' PLAN packets to be injected into the network by a means other than the local PLAN interpreter.

The Hello world program may be found in the file `interp_tests/Helloworld.plan`, and that you want to use the host $m : 3324$ as the network entry point. Let us also give this program an initial resource amount of 10.[6] Now type (again, from the plan directory):

```
% bin/inject -p 3324 interp_tests/Helloworld.plan 10
```

or alternatively

```
% bin/inject interp_tests/Helloworld.plan 10
```

The program waits for you to give it the next input : the initial invocation. Recall that a PLAN program is a list of definitions - when the program arrives at an active host, you must specify which function to start executing and with what arguments. In the case of the Hello world program, there is only one function to execute, and it takes no parameters, so you type:

```
doit ();
```

followed by a carriage return.[7] Out comes the response:

```
[(some.ip.address,3324);<loopback>] says : Hello world!
```

Notice that you must press Ctrl-C to get back to the command line. This is because `inject` waits for possible responses; since it does not know how many

[4] This is why the argument given to `pland`'s `-ip` option must match the port portion of some network address in the interface spec file – otherwise, calls to `thisHost()` from `inject` will not have the proper network address.

[5] In particular, the evaluation is set to your machine's name, as returned by the Unix call `gethostname`, concatenated with the given inject port

[6] We'll see later that the resource bound is irrelevant for this program!

[7] Earlier versions of PLAN required that you use Ctrl-D rather than a semicolon. This was due to a poor parsing definition. We allow the use of Ctrl-D rather than a semicolon for backward compatibility

responses to expect, it has to wait indefinitely until told explicitly to stop. Notice that a *list* of names is returned as a result of the `thisHost` call, which differs from PLAN 2.x. This is because each PLAN router, like an Internet router, may have multiple network interfaces with one address assigned per interface. All routers will have at least two addresses: a network address and a loopback address.

As a final note, had you included the `-v` option, you would have seen the following (the initial invocation as if you had typed it is included):

```
% bin/inject -v -p 3324 interp_tests/Helloworld.plan 10
Using port 3324 on localhost.
Initial resource bound of 10.
Consuming host resolution file EXP_IP_ADDRS.
Attempting to read code from files interp_tests/Helloworld.plan...done
Parsing ...
Enter initial invocation:
doit()
Initial invocation parsed.
1.1-1.6: Warning: service doit not declared.
Initial invocation typechecked.
Packet is 139 bytes
[127][0][0][1][12][252][0][0][0][10][127][0][0][1][12][252][0][0][0][0]
[0][0][0][0][0][12]defaultRoute[0][10]defHandler[0][0][0]U[0][4]doit[0]
[0][0][1][2][0][4]doit[0][0][0][11][0][0][0][2][2][0][5]print[0][1][2][
0][8]thisHost[0][0][2][0][5]print[0][1][0][3][0][0][0][21] says : Hello
 world![10]
[(127.0.0.1,3324);<loopback>] says : Hello world!
```

This is useful in debugging errors.

4.3 Understanding the program

This very simple program defines a function called `doit`, which takes no parameters, and returns a `unit` result, as inferred by the PLAN type-inference algorithm.

The body of the function calls a service routine called `print`. The `print` routine takes a single argument, which is the datum to print. The second `print` call is invoked on a string, while the first argument is yet another service routine call; the `thisHost` service routine returns a value of type `host list` which is a representation of the active router that the program is currently executing on. Both service routines are declared as prototypes using the `svc` keyword.

To illustrate how the `print` routine is working, let us briefly look at how PLAN programs are *injected* by a host into the active network, and how the host obtains their output. Figure 1 illustrates a hypothetical active network and a host with an application that desires to inject a PLAN program into that network. To do this, the application opens a connection to the local PLAN

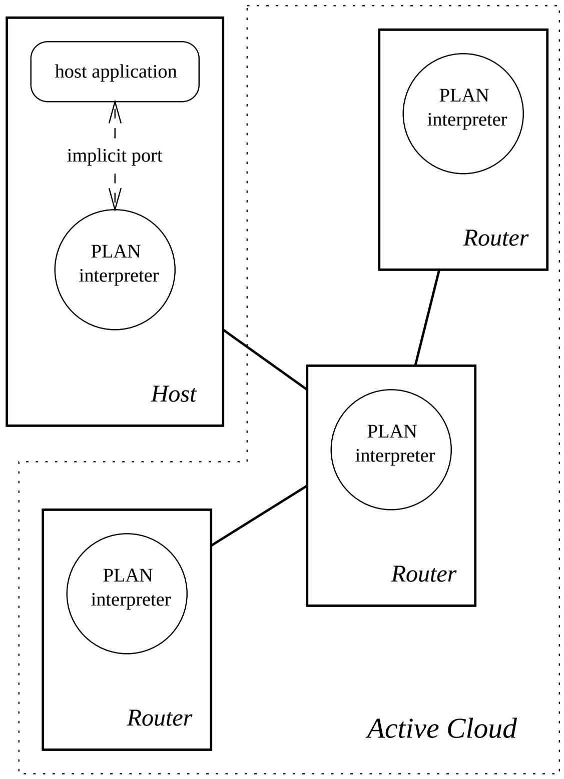

Figure 1: The PLAN network environment

interpreter, and constructs and submits a PLAN packet via this connection. The PLAN packet contains the PLAN code to execute, which in our example is the "Hello World" code, as well some other things that we need not worry about yet. This connection also serves to pass output from the PLAN program back to the host. It is via this connection that `inject` receives the output generated by the `print` command, and it is `inject` that displays this output on your screen.

In PLAN, this connection is referred to as a *PLAN port*. PLAN ports are duplex connections between host applications and active routers. The active router side of the PLAN port always receives PLAN packets, while the application side receives output in a number of formats, depending on the service command used; for the `print` routine, the output format is a string. When a PLAN packet is first injected into the network onto a particular node, it is associated with its injection PLAN port on that node. That way, output routines such as `print` have a default PLAN port upon which to send their output. These routines will only work on the injection node, because that is where the PLAN port resides. For example, in Figure 1, the PLAN program in the packet submitted by *Host* will have its `print` output sent back to *Host* via the injection

PLAN port so long as it is executing on *Host*. If the program were to move to another node and then execute the `print` routine, an exception would be raised.

For a precise description of what the various service routines do, and a greater exposition on PLAN ports, please refer to the PLAN Programmer's Guide [5].

5 Moving on

Now that we are familiar with `inject` and the basic format of a PLAN program, we can write more useful programs. PLAN is meant specifically for writing programs to be used in an active network—so let us run a very simple program that uses the active network.

5.1 Ping

The Ping program tries to reach a remote host, and once that task is accomplished, it sends the Ack program back to the original source which prints "Success."

```
svc print : 'a -> unit

fun ack() = print ("Success")

svc thisHostIs : host -> bool
svc getRB : void -> int
svc defaultRoute : host -> host * dev

fun ping (source, destination) =
  if (thisHostIs (destination)) then
    OnRemote (|ack| (), source, getRB (), defaultRoute)
  else
    OnRemote (|ping| (source, destination),
              destination, getRB (), defaultRoute)
```

5.2 Understanding the program : the OnRemote primitive

The **OnRemote** primitive is one of the two primitives[8] provided in the language for specifying computation at a node across the network. It takes four arguments:

What The first argument specifies what to execute at the remote node. It must be of the form of a function call.[9] Note that the arguments to the

[8]There is actually a third primitive, **RetransOnRemote**, but it is not described here.

[9]Notice those funny |'s around the function call name? These are syntax for defining a value of type $\alpha chunk$. More will be said about chunks a little later.

function call will be evaluated on the machine invoking OnRemote, while the function will be evaluated on the remote machine.

Where The second argument specifies where to execute the packet next. Note that the packet may go through many other active nodes in the process of getting to the specified host, but it will not be executed at any intermediate nodes.

How much The third argument specifies how much of the current packet's resources to give to the outgoing packet.

How to The fourth argument specifies which routing function to use to get to the remote host.

The other network primitive provided is **OnNeighbor**. It requires that the second argument be *adjacent* to the current host; adjacency is defined in terms of the underlying link layer. When PLAN is running on top of IP, adjacency is defined by neighbors specified for each virtual IP device in the interface specification file. In our example in Section 3.1, interface file $m1$ specifies that m:`3324` is adjacent to m:`3325`. When PLAN is running on top of Ethernet, two nodes are adjacent if they are on the same Ethernet. When two nodes are adjacent, no routing function is required, since routing is only necessary between physical networks. Instead, OnNeighbor needs the network interface through which to send the packet (which is of PLAN type `dev`).

The program consists of two function definitions. The first function, called `ack` simply prints "Success." The second function, `ping`, is more interesting. It takes two arguments of type `host` and returns a unit result. The arguments name the source and destination[10] of the computation. The body of the function first checks to see if the packet has arrived at the destination (the `thisHostIs` call). If so, it sends the ack function back to the source where "Success" will be printed. Otherwise, the function resends itself towards the destination. Both of these actions make use of the OnRemote primitive, and in both cases give all of current resources to the outgoing packet, and specify `defaultRoute` as the routing function to use.

5.3 Run this program

We'll use `inject` again, and we'll use the active network we set up at the end of Section 3, so our two nodes are m:3324 and m:3325. Suppose that you want to ping m:3325 from m:3324. The ping program may be found in the file `rout_tests/ping.plan`. We'll give this program a resource bound of 60 (will be explained later).

Then you invoke the program (from the plan directory) by the following :

```
% bin/inject -p 3324 rout_tests/ping.plan 60
```

[10] Note that these are independent of the packet header

And the initial invocation would be :

```
ping (getHostByName ("m:3324"), getHostByName ("m:3325"))
```

Notice that `inject` just provides a way from doing an OnRemote from the shell: the initial invocation specifies the first argument to OnRemote, the second and fourth arguments are specified with `-ed` and `-rf` options, and the third argument is specified by the RB.

Recall the initial invocation for the Hello World program – simply the function call `doit()`. Here, the initial invocation is a bit more complex. We first specify that we want to execute the `ping` function when we reach our injection node, and then we specify the arguments to give to that function. Here, the first two arguments are themselves function calls. The interesting thing is that these arguments are evaluated *immediately* before being sent to the injection point. Therefore, the `getHostByName` function is executed locally, in the `inject` program, in order to resolve hostnames into PLAN values of type `host`. It is these PLAN values that are actually sent in the PLAN packet. `getHostByName` is one of a number of *local services* that are made available to programs using `inject`. These local service calls may only appear, and will only be evaluated, in the initial invocation provided to `inject`. For a complete list of local services, please refer to the PLAN Programmer's Guide [5]. Here, `getHostByName` looks up the name of the machine that you gave it, m, in the DNS table and retrieves the address for that machine which is then coupled with the provided port number to complete the virtual address of the node (having PLAN type `host`).

After typing this in, you should see a message on your screen :

```
Success
```

6 Chunks

A fundamental part of PLAN computation is defined by entities known as *chunks*. These are essentially three-tuples consisting of code (possibly empty), a function name to execute (possibly a service), and a list of values to apply to the given function name. Essentially a PLAN packet, which is created by a call to OnRemote or OnNeighbor, consists of a chunk to be evaluated and location to evaluate it (the other fields in the packet assist in this process).

In PLAN, chunks are *first-class*, which means they can be manipulated as data (bound to variables, passed to functions, etc.) The syntax for creating a chunk is to surround a function call name with `|`'s, as in the ping example above. This has the effect of first evaluating the arguments of the function call (since they must be values to be transmitted over the wire), grabbing the necessary code, and finally packaging both results with the function name into a single entity. Conceptually, the function name is not resolved locally, and this is indicated syntactically by surrounding it with `|`'s. More details about chunks can be found in [5].

7 The resource bound

7.1 Rationale

Consider the following PLAN program (found in `rout_tests/ping_pong.plan`):

```
svc print : 'a -> unit
svc getRB : void -> int
svc defaultRoute : host -> host * dev

fun ping_pong(pingHost, pongHost) =
 (print("Current RB is "); print(getRB()); print("\n");
  OnRemote (|ping_pong| (pongHost, pingHost),
            pongHost, getRB (), defaultRoute))
```

A very simple program to write and run, but if this program were to be left loose on the network (without any resource bound control), it would keep shuttling between two hosts and eat up costly bandwidth.

7.2 Managing the resource bound

The concept of a resource bound is provided as a solution to this problem. Each packet has associated with it a certain amount of "resources". The packet can only access this value through the `getRB` service routine; it can specify the amount of resource bound it wants to give out of its own bank to an outgoing packet through the third argument of OnRemote and OnNeighbor.

Each hop that a packet traverses decreases its resource bound by one. Additionally, whenever a packet causes a new packet to be sent out into the network (through OnRemote or OnNeighbor) its resource bound is decremented by the amount it allocated to the outgoing packet.

If at any point the packet does not have enough resource bound left, either to traverse the next hop or to give out to an outgoing packet, the *NotEnoughRB* exception is raised, which can either be handled by the program or ignored; in the latter case a message is sent back to the source and printed on the console. No further hops may be traversed, and no additional packets may be sent when the resource bound reaches 0.

7.3 The two examples and resource bound

In the first example that we saw, there were no network primitives, no packets going out. The resource bound was totally irrelevant in that context.

In the second example, the program always gave all of its current resource bound to an outgoing packet. Between the local and remote hosts, the packet might traverse a number of hops, and the resource bound will be decremented before each hop. The initial amount of resource bound given to the packet must be enough to take it to the remote host and get it back. In the example, we assumed that the packet would not take more than 30 hops on each leg, so we gave it a total of 60 units of resource bound initially.

8 The Plan tool

In addition to `inject`, the distribution also provides a tool for running your programs in a read-eval-print loop, called `plan`. This program takes a single, optional argument, which is the PLAN program file to read in at the outset. After startuip, you will be prompted with a `-->` and you can respond with a valid PLAN expression to compute, followed by semi-colon (or Ctrl-D). For example:

```
% bin/plan interp_tests/Helloworld.plan
--> 1 + 1;
2
--> doit();
1.1-1.6: Warning: service doit not declared.
[(158.130.6.143,3324);<loopback>] says : Hello world!
()
--> "hello "^"world";
hello world
-->
```

Note that because expressions are typechecked in isolation, you will get warnings like

```
1.1-1.6: Warning: service doit not declared.
```

for services and functions defined in your included code files. The network primitives are not available in this tool; if an expression evaluation tries to use OnRemote or OnNeighbor the evaluation is aborted. Also, the Router Services (as described in the PLAN Programmer's Guide [5]) are not available on the read-eval-print loop.

Press Ctrl-C to exit the loop.

9 Conclusion

Thus ends the short tutorial. For more details on the actual semantics and specs, refer to the PLAN Programmer's Guide [5].

References

[1] D. Scott Alexander, Marianne Shaw, Scott Nettles, and Jonathan M. Smith, "Active Bridging," *Proceedings, 1997 SIGCOMM Conference*. 1997.

[2] K. Arnold and J. Gosling, "The Java Programming Language," Addison-Wesley, 1996.

[3] M.W. Hicks, "The PLAN Active Router," PLAN 3.2 documentation, October 1998.

[4] M.W. Hicks, "The PLAN Security Guide," PLAN 3.2 documentation, October 1998.

[5] J.T. Moore and M.W. Hicks, "PLAN Programmer's Guide," PLAN 3.2 documentation, October 1998.

[6] M. Odersky and P. Wadler, "Pizza into Java: Translating theory into practice," *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.