1 Introduction

This document describes the 3APL ‘prototype’ interpreter system. It is created to experiment with new features of 3APL, like complex goals and agent communication. As a prototype system, it doesn’t implement full 3APL and it lacks many features that the full Java system has.

- no graphical environment.
- just bare-bone prolog support, i.e. no floating point, strings, libraries etc.

However, it does have some interesting features too:

- supports multiple, repeated goal variables in rule bodies and patterns.
- supports multiple (concurrent) goals.
- supports multiple, concurrent, communicating agents.
- supports a most-general goal selection algorithm. There exist a special version where you can write and select your own goal selection algorithm in prolog, i.e. longest versus shortest match, no backtracking beyond a guard etc.
- supports (platform independent) external components written in any language.
- It has a host of demo programs in the demo directory.

2 Environment variables

Before using the system, you need to set two environment variables, APLPATH and PROLOGPATH. The first variable points to a list of paths that contain 3APL (.3apl) files and the second to prolog (.pl) files. The APLPATH should at least contain the demo directory and the PROLOGPATH at least the prolog directory. On my system it looks like:
3 The interpreter

The interpreter is started by typing `triple-apl`. In this document, we assume that the current working directory is the main 3APL directory. The interpreter then resides in the `bin` subdirectory.

```
C:\docs\3apl>bin\triple-apl
loading: c:\docs\3apl\prolog\strategy.p
loading: c:\docs\3apl\prolog\stdlib.pl
>
```

We load a file with `:l` command. The `demo` directory contains some interesting example files.

```
> :l mail3
loading: c:\daan\3apl\demo\mail3.3apl
loading: c:\daan\3apl\prolog\stdlib.pl

We can run the program using `:g` (go).

```
> :g
mail3: Go(room4,room3)
mail3: Go(room3,room1)
mail3: Go(room1,room2)
mail3: Takemail()
mail3: Go(room2,room1)
mail3: Postmail()
done.
```

Ok, let’s reload and step through it. A next step is done by typing a semicolon (;). We can start executing again using the `g` key and break it using any other key.

```
> :r
loading: c:\daan\3apl\demo\mail3.3apl
loading: c:\daan\3apl\prolog\stdlib.pl
>
```

```
> :s
mail3: Go(room4,room3)
[step]
mail3: Go(room3,room1)
break.
```

At this point, we look at the current beliefs and goals of the program.
3 The interpreter

> ?b
current beliefs:
  forbid(door42).
  mail(room2).
  pos(room1).
  self(mail3).

> ?g
current goals:
  Go(room1, room2);
  Takemail();
  Go(room2, room1);
  Postmail()

You can type ?- at the command prompt to evaluate prolog terms. Type ; to find a next solution or any other key to terminate the proof.

> ?-append(X,Y,[1,2]).
X = [], Y = [1, 2] [next]
X = [1], Y = [2] [next]
X = [1, 2], Y = [] [next]
no.

3.1 Commands

Supported interpreter commands are:

:q – quit
:l filename – load a 3APL program
:g – run the program (go!)
:s – start in step mode
:r – reload the program
:n – show command n from the history

?- terms . – prove terms
?p – show the program
?b – show the current beliefs
?g – show the current goals
?h – show the command history
?l – show the prolog library

3.2 Quirks

- The syntax differs from the Java system. Every statement is terminated with a dot (.), while predicates are separated by a comma, just like Prolog. Goals are separated by a semicolon (;). A statement is a capability, belief, clause, goal sequence or rule. The a formal definition of the syntax is partly given in the syntax.pdf document.
• Only core-prolog is supported without any extensions. The standard library consists only of prolog/stdlib.pl. However the following items are supported:
  – Full list syntax.
  – Unnamed structures or tuples – (a,b).
  – The findall(vars, term, results) predicate.
  – Evaluation: is.
  – Comparisons: <, <=, >, >= and ==.
  – Integers, together with: +, -, *, / and abs(i).

4 Multiple agents

Multiple concurrent- and communicating agents are supported. The load (:l) command can take an optional agent name to distinguish between different agents. For example:

```apl
> :l x mail3
loading: c:\daan\3apl\demo/mail3.3apl
loading: c:\daan\3apl\prolog/stdlib.pl
> :l y mail3
loading: c:\daan\3apl\demo/mail3.3apl
loading: c:\daan\3apl\prolog/stdlib.pl
> :s
y: Go(room4,room3)
[step]
x: Go(room4,room3)
[step]
y: Go(room3,room1)
[step]
x: Go(room3,room1)
break.
> :l z mail3
loading: c:\daan\3apl\demo/mail3.3apl
loading: c:\daan\3apl\prolog/stdlib.pl
> :g
z: Go(room4,room3)
y: Go(room1,room2)
x: Go(room1,room2)
z: Go(room3,room1)
y: Takemail()
x: Takemail()
z: Go(room1,room2)
y: Go(room2,room1)
x: Go(room2,room1)
z: Takemail()
y: Postmail()
x: Postmail()
z: Go(room2,room1)
```
Multiple agents

z: Postmail()
done.

You can look at all agents by using the ?a command:

> ?a
agents:
y
x
z

Each agent knows its own identity with the self belief:

> ?b x
current beliefs:
  forbid(door42).
  pos(room1).
  self(x).

Since communicating agents normally need to know about each other and since they are normally set up in some specific way, the interpreter allows agents to instantiate other agents dynamically through the `New(agent, filename, component, beliefs)` action. This action automatically adds the `self(agent)` belief to the newly instantiated agent. The buyerseller demo demonstrates this. Note that the :L command discards all previously loaded agents.

> :L god buyerseller
loading: c:\daan\3apl\demo/buyerseller.3apl
loading: c:\daan\3apl\prolog/stdlib.pl
> ?a
agents:
god

If we now step through the program, we'll see that god creates a buyer and seller agent.

> :s
god: New(seller,seller,"",[customer(customer)])
  [loading agent "seller"]
loading: c:\daan\3apl\demo/seller.3apl
loading: c:\daan\3apl\prolog/stdlib.pl
[step]
god : New(customer,buyer,"",[salesPerson(seller)])
  [loading agent "customer"]
loading: c:\daan\3apl\demo/buyer.3apl
loading: c:\daan\3apl\prolog/stdlib.pl
[step]
break.
> ?a
agents:
seller
god
customer

The New action takes four parameters, an agent name, the 3APL file name that describes the agent, an external component and a list of initial beliefs of the agent. We will describe external components later in this document. The last parameter is normally used to let the agents know about each other.

The agents can now communicate with each other through the primitive $\text{Send}(\text{agent}, \text{message})$ action.

> :g
customer: Send(seller, request(sellPC(customer,C_1,five)))
seller : Send(customer, agree(sellPC(customer,c2,five)))
customer: Agree(c2,five)
seller : SellPC(customer,c2,five)
seller : Send(customer, inform(done(sellPC(customer,c2,five))))
customer: Bought(c2,five)
done.

After an agent sender has executed a successfull $\text{Send}(\text{agent}, \text{message})$ action, the agent automatically believes: $\text{sent}(\text{agent}, \text{message})$, while the receiving agent believes: $\text{received}(\text{sender}, \text{message})$.

4.1 Commands

Extended interpreter commands for multiple agents are:

:1 agent filename – load a 3APL program with a specific name
:L agent filename – discard all loaded agents and load filename with name agent
:r agent – reload agent
:R agent – discard all loaded agents and reload agent

?- agent - terms . – prove terms in the context of agent
?p agent – show the program of agent
?b agent – show the current beliefs of agent
?g agent – show the current goals of agent
?l agent – show the prolog library of agent

5 External components

The interpreter can load external components that can be scripted. As an example, we look at the trace demo that normally just executes some actions in a row:
However, we can also attach the external `msagent` component to this script and all the command will also be send to that component. The component will execute those actions and we will see an nice cartoon character on the screen that says “Hello”, see figure 5. Unfortunately, the `msagent` component only works under windows since it uses the Microsoft Agent component to display the cartoon character.

As you can see, the interpreter output is almost the same – except for the trace message `merlin: Hello world!` that was printed by the external component. Of course, the cartoon character also performed all his actions.
The component was attached by simply adding a command to the :L command. Note that we always need to give an agent name and file name explicitly, or otherwise the command is interpreted as an agent file. The command can also contain extra arguments, or a complete path name:

> :L x trace "c:\3apl\ext\msagent.exe genie"

External components can also return results to the script. A 3apl script can retrieve those results by using an is expression in the capability description. For example, the msagent component returns the new position in the Move method:

\{ pos(X0,Y0)} (X1,Y1) is Move(X,Y) { NOT pos(X0,Y0), pos(X1,Y1) }.

More information about programming the Microsoft agent component can be found on http://www.microsoft.com/msagent. The demo2a, buyersellerYa and agent demos give some examples of how to program these components from 3APL.

5.1 Commands

Extended interpreter commands for external components are:

:1 agent filename command – load a 3APL program with a specific name attached to command
:L agent filename command – discard all loaded agents and load filename with name agent. Also spawn command and communicate the actions with this process.

5.2 The external interface

Communication with external components goes via pipes. An external component is simply a program that reads its input on the standard input channel (stdin) and writes responses to the standard output channel (stdout). When the interpreter loads an external component, it redirects its standard input channel and output channel to some private channels and uses this to communicate with the component. The component doesn’t know, nor care, whether it is run from the command line or within the interpreter – it simply reads commands and sends responses.

All input and output is always terminated with a new-line character. An input line is an action or the special input quit or quit() that urge the component to quit. The output of a component is either the term ok(...) or error(X), where X is some error string. The arguments of ok are matched with the is expression in the capabilities.

Since external components are simple console applications, we can test them on the command line, giving the commands ourselves. Here is for example a session with the agent component:

C:\daan\3apl>ext\msagent
There are two examples of external components delivered with 3APL. They are both written in Java but any language that can read/write on stdin/stdout can be used: C, Haskell, Perl, etc.

- **trace.** The simplest possible external component written in Java. It only supports the `Trace(S)` action, writing S to stderr.

- **msagent.** A realistic external component written in Java. It uses the Microsoft Agent COM component and is thus only available on windows. The `demo2a`, `buyersellerYa` and `agent` demos use this component.

6 Planning and reflection

Sometimes, the default reduction strategy is not the best possible strategy. The interpreter therefore supports two commands, `Plan` and `Exec` that can generate and execute plans. Two demos show how planning works, `mailplan` and `mailplan2`. For example, `mailplan2` contains the following rules for getting mail from room 2:

```
RULEBASE:
getmail <- mail(room2) | go2; Takemail(); Go21(); Postmail().
go2 <- pos(room1) | Go12().
go2 <- pos(room3) | Go31(); go2.
go2 <- pos(room4) | Go43(); Go31(); Go12().
go2 <- pos(room4), not(forbid(door42)) | Go42().
```

The last two rules give a path from room 4 to room 2. The last one is an optimized version of the previous rule but it only works when the door is open (`not(forbid(door42))`). Unfortunately, the default search strategy will always choose the first rule. One solution is to add an extra guard to the first rule (`forbid(door42)`) but we can also *dynamically* examine all possible ways to reach a goal and choose the shortest sequence. This can be written as:

```
RULEBASE:
```
The primitive Plan(Goals, MaxReductions, MaxCount, Plans), returns at most MaxCount reduction plans in Plans that reach the goals Goals in at most MaxReductions (rule) reductions. The Plans can subsequently be examined by the following primitives:

- listPlans(Ps,Qs). Convert the plans Ps in a list of plans Qs.
- emptyPlan(Ps). Provable when the plans Ps are empty.
- nextPlan(Ps,P,Qs). Provable when the plans Ps are not empty, P is unified with one of the plans and Qs with the rest.
- cost(P,N). Returns the cost of a plan P in N.

A plan Plan can be spliced into the current goals using the Exec(Plan) primitive.

Note that the first argument of Plan is a prolog term. A list of goals can be spliced into a prolog term by enclosing it with curly braces ({}).

A Strategies

Beside the normal distribution, there exists a special 3APL version that supports hand written goal selection strategies.

You can use different reduction strategies using the :ls command. The move, ab and bomb demos show how different strategies can have different results.

- strategy. The default strategy, general backtracking.
- strategyE. An Eager strategy that doesn’t backtrack through guards.
- strategyEL. The eager strategy combined with longest match.

A.1 Writing strategies

Let’s have a look at the default reduction strategy:

```prolog
% general pattern matching, shortest match first (switch last 2 clauses to get longest match % match( [in] Goals, [in] Pattern ).
match([],[]).
match(Xs,[[|Yss]) :- match(Xs,Yss).
match([X|Xs],[Y|Ys]|Yss]) :- match(Xs,[Y|Yss]).
```

% general matching
% selectrule([in] Goals, [out] Goals)
selectrule(Goals,Body) :- rule(Pattern,BodyX),match(Goals,Pattern),concat(BodyX,Body).

% single step reduction: a rule is only selected when its guard also matches
% step([in] Goals, [out] (Goals | Hnf))
step([],hnf(done,[])).
step([test(Ts)|Goals],Goals) :- all(Ts).
step([action(Action)|Goals],hnf(Action,Goals)) :- Action.
step(Goals,Body) :- selectrule(Goals,[guard(Gs)|Body]),all(Gs).

% reduce goals and allow backtracking
% reduce([in] Goals, [out] Hnf )
reduce(Goals,hnf(Action,Body)) :- step(Goals,hnf(Action,Body)).
reduce(Goals,Hnf) :- step(Goals,NewGoals),reduce(NewGoals,Hnf).

The 3APL interpreter will call reduce with the current goals as a first argument and it will expect a head normal form as a result. A head normal form consists of an action and new list of goals – hnf(Action,Goals).

As we can see from the above program, reduce uses step to reduce the goals. A step either delivers a head normal form or it returns a new list of goals (NewGoals).

How is a single step performed? If there are no goals left ([[]]), we are done. When a test test(Ts) is encountered, we need to prove all the predicates in the list Ts. When an action action(Action) is seen, we need to prove the pre conditions of the action, which can simply be done by proving Action since the 3APL system has already compiled all actions with their pre conditions. You can see those action by typing ?a in the interpreter.

If this all fails, we can also try to rewrite the goals by using a practical reasoning rule. This is done by selecting a rule that matches a certain pattern with selectrule, and proving its guards. Note that the rules are already compiled by the system as rule predicates, containing a list of goal lists as pattern and a list of goals lists as body. You can see the compiled rules using ?r.

You can write your own strategies using this template. For example, an eager strategy (strategyE) is written by never backtracking through guards. We can express this as:

reduce(Goals,hnf(Action,Body)) :- step(Goals,hnf(Action,Body)).
reduce(Goals,Hnf) :- step(Goals,NewGoals),!,reduce(NewGoals,Hnf).

A strategy that uses longest match first for goal variables is written as:

match([],[]).
match([X|Xs],[[X|Ys]|Yss]) :- match(Xs,[Ys|Yss]).
match(Xs,[[]|Yss]) :- match(Xs,Yss).

And finally, a strategy that always commits to the first result (depth-first), can be written as:
reduce(Goals,hnf(Action,Body)) :- step(Goals,hnf(Action,Body)),!.
reduce(Goals,Hnf) :- step(Goals,NewGoals),reduce(NewGoals,Hnf).