

AL-Interpreter in Perfect

Ingo Feinerer
Vienna University of Technology
e0125130@student.tuwien.ac.at

July 15, 2003

Introduction

This document forms the documentation to an AL (Assignment Language) Interpreter written in Perfect in the Perfect Developer 2.0 working environment [1]. The AL Interpreter is a project originated from the Vienna University of Technology, where AL is used in one of the computer science basic courses to get students in contact with formal verification.

In this documentation you will find basics needed for the further understanding of this document, the AL syntax and semantics, the modified grammar that the AL Interpreter uses indeed, insights in the implementation and an overview about the problems encountered and solutions to them. Finally I will give a short review and my personal opinion reflected in the final summary.

The formal description of AL is based upon the definitions in the scriptum "Theoretische Informatik 1" from Gernot Salzer [3]. For this project many definitions were shortened or adapted to keep the document as readable and short as possible. For interested it is worth to have a look at it. Especially first order logic definitions have lost axioms for quantors and still are mentioned as PL (Prädikatenlogik = first order logic in German). Also the data type is set to \mathbb{N} (0, 1, 2, 3 ...), whereas the original AL works with other data types as well (stacks ...).

Basics

In this section basic elements are defined, which will be needed for AL syntax and semantics definitions.

An individual variable symbol IVS (Individuenvariablensymbol) starts with x, y or z followed by an optional nonnegative index. A constant symbol KS (Konstantensymbol) is a nonnegative integer. A function symbol FS_2 (Funktionssymbol) is either a + or – or *, where the – has a special meaning: the result has to be positive (including 0) — if a subtraction results in a negative integer it has to yield 0. PS_2 (Prädikatensymbol) is either < or =.

The set of terms T is the minimal set restricted to:

- (T1) $IVS \subseteq T$.
- (T2) $KS \subseteq T$.
- (T3) $\underline{(T_1 FS_2 T_2)}$ with $T_1, T_2 \in T$.

I is called interpretation (a mapping from variables to values) and yields back the actual value of a variable. ENV (environment) is the set of all possible I.

The semantics of terms is determined by the function $M_T : ENV \times T \mapsto \mathbb{N}$. M_T is evaluated according to following rules:

- (MT1) $M_T(I, v) = I(v)$ if $v \in IVS$
- (MT2) $M_T(I, c') = c$ if $c' \in KS$, where c is the constant to the symbol c' .
- (MT3) $M_T(I, \underline{(T_1 FS'_2 T_2)}) = M_T(I, T_1) FS_2 M_T(I, T_2)$, where FS_2 is the function for the symbol FS'_2 .

The set of first order logic formulas is the minimal set restricted to:

- (PL1) $\underline{(T_1 PS_2 T_2)} \in PL$ with $T_1, T_2 \in T$.
- (PL2) $\underline{(F \wedge G)} \in PL$ with $F, G \in PL$.
- (PL3) $\underline{(F \vee G)} \in PL$ with $F, G \in PL$.
- (PL4) $\underline{(F \supseteq G)} \in PL$ with $F, G \in PL$.
- (PL5) $\underline{\neg F} \in PL$ with $F \in PL$.

The semantics of PL formulas is determined by the function $M_{PL} : ENV \times PL \mapsto \{t, f\}$. M_{PL} is evaluated according to following rules:

- (MPL1) $M_{PL}(I, \underline{(T_1 PS'_2 T_2)}) = M_T(I, T_1) PS_2 M_T(I, T_2)$, where PS_2 is the function for the symbol PS'_2 .
- (MPL2) $M_{PL}(I, \underline{(F \wedge G)}) = M_{PL}(I, F) \wedge M_{PL}(I, G)$.
- (MPL3) $M_{PL}(I, \underline{(F \vee G)}) = M_{PL}(I, F) \vee M_{PL}(I, G)$.
- (MPL4) $M_{PL}(I, \underline{(F \supseteq G)}) = M_{PL}(I, F) \supset M_{PL}(I, G)$.
- (MPL5) $M_{PL}(I, \underline{\neg F}) = \neg M_{PL}(I, F)$.

AL syntax

The set AL – Assignment Language – of *statements* is the minimal set restricted to:

- (AL1) If $v \in IVS$ and $t \in T$, then $v \leftarrow t \in AL$.
- (AL2) If $\alpha, \beta \in AL$, then $\underline{\text{begin}} \alpha ; \beta \underline{\text{end}} \in AL$.
- (AL3) If $B \in PL_0$ and $\alpha, \beta \in AL$, then $\underline{\text{if}} B \underline{\text{then}} \alpha \underline{\text{else}} \beta \in AL$.
- (AL4) If $B \in PL_0$ and $\alpha \in AL$, then $\underline{\text{while}} B \underline{\text{do}} \alpha \in AL$.

AL semantics

The semantics of AL statements is determined by the function $M_{AL} : ENV \times AL \mapsto ENV$. M_{AL} is evaluated according to following rules:

- (MAL1) $M_{AL}(I, v \leftarrow t) = I'$ with $I'(v) = M_T(I, t)$, $I'(w) = I(w)$,
 $\forall w \in IVS, w \neq v$.
- (MAL2) $M_{AL}(I, \underline{\text{begin}} \alpha ; \beta \underline{\text{end}}) = M_{AL}(M_{AL}(I, \alpha), \beta)$
- (MAL3) $M_{AL}(I, \underline{\text{if}} B \underline{\text{then}} \alpha \underline{\text{else}} \beta) = \begin{cases} M_{AL}(I, \alpha) & \text{if } M_{PL}(I, B) = t \\ M_{AL}(I, \beta) & \text{if } M_{PL}(I, B) = f \end{cases}$
- (MAL4) $M_{AL}(I, \underline{\text{while}} B \underline{\text{do}} \alpha) = \begin{cases} M_{AL}(M_{AL}(I, \alpha), \underline{\text{while}} B \underline{\text{do}} \alpha) & \text{if } M_{PL}(I, B) = t \\ I & \text{if } M_{PL}(I, B) = f \end{cases}$

AL-Interpreter Grammar

For technical reasons the grammar was changed at a few points and enriched with some information for the interpreter. The most important issue was to pass the environment on to the interpreter. This is done by a `envbegin ENV envend` sequence. Next follows the AL program construct tagged with `albegin AL alend`. \leftarrow is replaced by `=`, \neg by `!`, \wedge by `&`, \vee by `|` and \supset by `>>`.

The complete grammar with the starting symbol START can be written as shown here:

START \Rightarrow `envbegin ENV envend albegin AL alend`

ENV \Rightarrow `[IVS \equiv CONST]*`

AL \Rightarrow IVS \equiv T
 | begin AL ; AL end
 | if PL then AL else AL
 | while PL do AL

IVS \Rightarrow [xyz][0 – 9]*

T \Rightarrow IVS
 | CONST
 | (T FS T)

CONST \Rightarrow [0 – 9]+

FS \Rightarrow [± _ *]

PL \Rightarrow ! PL
 | (T PS T)
 | [PL OP PL]

PS \Rightarrow [≤ ≡]

OP \Rightarrow [& | ≥]

The AL Perfect parser only accepts a grammar if it is wellformed in a way shown above and every IVS used outside the ENV block has already received a value in the ENV block (this means every IVS has been declared and has a valid value).

Implementation

The AL-Interpreter is split up into various files, but mainly consists of three tasks, represented in the three files scanner.pd, parser.pd and interpreter.pd. If AL-Interpreter is invoked at first the parser is called who coordinates the whole job. He starts the scanner, who delivers the tokens and augments them with additional information — every token is assigned to a tokentype — then checks whether the input fullfills all grammar criterias and then and only then the interpreter is called, who evaluates the program and returns the final result — a mapping from variables to values: the socalled environment. In more specific terms the interpreter is implemented as a top-down recursive descent parser calling the methods of lexical analysis and afterwards the evaluation methods to receive the final environment. The parser and scanner are based upon a small compiler-interpreter developed at the Vienna University

of Technology [2] on the one side and upon a simple scanner example from David Crocker [1].

The following sections will give additional explanations of every component, nevertheless it is necessary to have a quick look at the source code to receive a comprehensive overview. It is also worth mentioning that these components are explicitly designed to work together and may only be separated under the conditions defined by pre- and postconditions to guarantee bug-free software as far as Perfect can prove this. For further details see the chapter "Provability".

Scanner

The class Scanner reads in its input and splits it up into tokens and saves them (see the function tokenize). After that the scanner investigates every token and maps a type to it so that the parser can easily check whether a token fulfills specific criteria (see tokentypize). The scanner has also the ability to deliver the next actual token from the input inclusive type to the parser (see next).

Parser

The class parser calls the scanner every time a token is needed, checks if the the input passes by the restrictions set up by the AL grammar, builds up a tree to allow the interpreter to forget the original input and only has to work on a wellformed tree. If the parsers comes to an end with a positive check the interpreter is called with the starting environment and the in the parser phase constructed tree. The interpreter evaluates the input tree and returns the modified final environment, which is the result of the whole AL-Interpreter.

Interpreter

The class Interpreter consists only of three methods which are called from the parser. They implement quite exactly the behaviour of the semantics (meaning) of AL, PL and terms, because the original definitions were given as recursive definitions. As the methods are also implemented in a recursive way it is obvious to see that these methods evaluate in the same way as wanted in the semantics section defined.

Provability

In this section an overview about the provability of the AL-Interpreter will be given. With Perfect Developer 2.0 it was possible to prove almost the whole AL-Interpreter in a way of minimizing warnings of the Perfect prover. To achieve this there were some tricks necessary. So PD 2.0 needs some axioms in order that the prover has some definitions for operations on strings in its scope (like `isDigit` and `isLetter`) — later versions will probably be able to check this. In addition it was necessary to guarantee a lot of preconditions for the interpreter functions because they rely on a wellformed tree produced in the parser phase. The fact that the interpreter is only called on positive investigation of the parser was not enough. Nevertheless with those preconditions this interpreter ensures now that there are no failures on misbuilt trees. Another aspect are the warnings about missing variants — these are left out on purpose because sometimes it is not possible to give a variant (AL allows infinite loops!), sometimes it was just convenience to get rid of them because the prover could not prove them (like the amount of unprocessed tokens decreases on every recursive call).

An important point is that AL is defined in a recursive way, what is a big advantage in solving this problem. Especially this is a point where PD had to show its capabilities, usability and efficiency. On the other side we may not forget that a interpreter with a scanner and parser is a sophisticated task, although this AL-Interpreter is a simple and tiny example of a real world computer science problem.

Summa summarum it can be said that Perfect was very straightforward to program with. Validation was a more complex topic — but if you get in contact with the Perfect prover you soon now what is important to write a provable program.

Summary

In this document you have found AL syntax and semantics and some words about the AL-Interpreter. In my personal point of view I am sure that Perfect is quite useful for a set of programs, especially programs related to what universities normally experiment with. For a business project I do not dare to give a judgement, it would be definitely necessary to have a deep look at this side of Perfect Developer. I am sure that the few problems described in the Provability chapter soon can be solved automatically by PD in the near future because these are only syntactic sugar. Finally I would like to thank David Crocker [1] for code samples and a lot of useful tips and Gernot

Salzer [3] for mentoring this project.

References

- [1] David Crocker. Perfect Developer 2.0. *Escher Technologies*, 2003. www.eschertech.com — Perfect: a language to guarantee bug-free software.
- [2] Institut für Computersprachen. Mini-Compiler in Java. *TU Wien*, 2003. <http://www.complang.tuwien.ac.at/lehre/uebersetzerbau-vo.html> — Mini.java is a compiler-interpreter for Mini written in Java.
- [3] Gernot Salzer. Theoretische Informatik 1. *Institut für Computersprachen, TU Wien*, June 2002. www.logic.at/lvas/thinf1/ — An introduction to formal languages, mathematical logics and formal verification.