Haski the Robot
Project report
Advanced functional programming 2004

Lars Otten, 810715-6690 <*ottenl@student.chalmers.se*>

December 14, 2004

1 Introduction

Haski the Robot is a simple, yet complete little programming environment aimed at the unexperienced user who wants to acquire some general ideas related to programming. It uses its own little programming language with which the user can maneuver one or more robots around a working area, trying to accomplish diverse tasks. It comes with an extensive user manual providing several examples and detailed explanations.

By keeping the language itself simple and providing visual output of the resulting robot actions, Haski the Robot quickly introduces the user to concepts like functions/procedures and the basic control structures, that are common to most major programming languages.

2 Similar, existing software

The first project of this kind dates back to 1981, when Richard E. Pattis developed "Karel the Robot" [1] which was originally based on Pascal. Since then several other versions have been developed, e.g. for C, C++ and Java.

In particular I would like to refer to "Niki der Roboter", which is also based on Pascal and is quite frequently used for programming courses in German secondary schools – in fact it provided my initial motivation to implement this project.

3 A short overview

To use Haski the user writes programs and saves them in plain textfiles. He can also specify a working area via a textfile. For a complete syntax reference please refer to the Haski user manual which is supplied with the program files.

A simple program file might for instance look like this:

```
main      = While (Not field_has_item :& front_free) go_forward &>
            turn_left &>
            IfThenElse front_free run (turn_left &> run)

run       = go_forward &> run

turn_left = turn_right &> turn_right &> turn_right
```

One can already see most features of Haski in this example, including for example:

- All the common control structures (if-then, while, do-while, if-then-else) are available to the user
- The user can define various functions and call them inside his program
- Functions can call themselves recursively
- Boolean expressions can be constructed via the usual AND, OR and NOT

Again, for a complete description of these things please consult the user manual, where you can also find numerous examples.

4 Putting it into action

In the following I will give some details about the development of the project, decisions I made and problems I encountered.

4.1 Implementation outline

The whole process of implementing the project can roughly be divided into a number of steps:

- Building the basic GUI functionality and its underlying interpreter for the embedded language
- Constructing a lexical and syntactical analyser for reading Haski programs from textfiles
- Likewise creating a lexical and syntactical analyser for working area definitions, reusing the results of the previous point
- Adding more sophisticated error handling and somehow informative error messages to the parsers.
- Putting it all together
- Enhancing the GUI with more features
- Finding and fixing bugs ;-)
- Writing the user manual with comprehensive examples and explanations (important for unexperienced users)

It should be noted that – especially towards the end – the distinct steps often overlapped to quite some extent.

4.2 Design decisions

When I started the project I just had the general idea of "Niki der Roboter" in mind, that I had been using at secondary school. And, as we covered that quite extensively in the lectures and labs, I was mainly considering an embedded language for that.

After three days of 'hacking', however, the main part of the GUI was almost complete, so this seemed too easy to me. Furthermore the main intention behind the project was to create a piece of software that could (in theory) be used by users with no programming experience at all. Therefore requiring those users to install GHC(i) in order to run Haski did not seem that promising after all.

One drawback of a textfile/parser-solution is that one loses many of the things an embedded language (or rather its host language) offers, for example function definitions or if-then-else-contructs. But since controlling the robot is supposed to be simple and allows only a very limited choice of commands, in this special case this was a loss I was willing to take; it just meant that I had to implement corresponding things in my own little 'external' language.

So I remembered our lecture about parsers and the course about compiler construction, that I had taken at my home university last year, and started searching the web for ways to parse Haski programs and working area definitions from text files. As I knew a little bit about `lex` and `yacc` from the afore mentioned course, I ended up with `alex` and `happy` as they seemed similar to me.

After some long nights of unsuccessful attempts and studying the respective user manuals, I finally managed to produce a first, simple working parser for Haski programs from textfiles, which I then integrated into the existing program. Now I was able to repeat the whole thing for working area files, where I mostly could reuse the ideas of the first parser. But one major problem arose with the introduction of these parsers: In case of an error they just call `error "..."` , which is, from the user's perspective, neither convenient (since the whole program 'crashes') nor helpful (due to the static error message).

Therefore I went back to the user manuals of `alex` and `happy` again, looking for ways to catch errors and produce meaningful output. I also remembered our lectures, where we introduced a little monad for handling errors – which is similar to what I eventually chose for my solution. Moreover I made the lexer remember line numbers and saved them into the token stream handed to the parser, so that they could be used for generating somewhat more helpful error messages.

Now that I had brought the basic program to a solid state it was time to consider making some refinements to the GUI in order to ease usage. I wanted the user to be able to dynamically load programs and maps (i.e. working area definitions), so I added a menu with items for this. I also added a "Reset"-button, to make re-executing a program more convenient. Furthermore I added a kind of log-window, where the robots output feedback for the commands they carry out.

Finally I started writing the user manual, which I consider very important, as the program is (at least in theory) intented for unexperienced users. It turned out to be quite a lot of work, more than I had expected, but I wanted it to be quite comprehensive and easy to understand after all.

4.2.1 The Haski language

After moving from an embedded language to an 'external' one, I could have redesigned the whole thing completely. But I rather chose to stick to the notation of the embedded

language and 'simulate' it in a way (so that it at least looks like Haskell), e.g. by keeping the names for operators like `&>` and `:&` and `:|` . After all, it's supposed to be a Haskell project. ;-)

But as the Haski language itself it pretty simplistic, there is actually no real functional aspect in it, one could also regard it as an imperative language (as it was originally based on Pascal).

4.3 Implementation details

As mentioned above I introduced a little monad for error handling. It contains either the (intermediate) parsing result or an error message as a string:

```
data LexE a = Ok a | Failed String

instance Monad LexE where
 return a = Ok a
 m >>= k  = case m of
              Ok a     -> k a       -- continue with parsing
              Failed e -> Failed e  -- just pass the error message along

failE :: String -> LexE a           -- creates an error message
failE err = Failed err
```

The lexical and syntactical analysers for Haski programs are then functions as follows:

```
lexer  :: String  -> LexE [Token]
parser :: [Token] -> LexE [Function]
```

`Token` is the data type for the tokens the lexical analyser identifies, which is needed by the syntactical parser. `Function` is the datatype used for representing one function parsed from the file, it contains the function's name and the actual instructions.

In this context it is perhaps interesting to look at how the (possibly) multiple functions parsed from a file are handled – also remember that each robot can be assigned a different program file. Internally a robot is represented by a state of the following type:

```
data State = State{
 numItems :: Int,       -- no. of items the robot is currently carrying
 facing :: Facing,      -- direction robot is currently facing
 pos :: Position,       -- current position
 prog :: Program,       -- commands to carry out, initially the main-function
 funcs :: [Function],   -- all functions returned by the lexer/parser
 file :: FilePath       -- the hs-file which was parsed for that robot
}
```

We notice that each robot always 'carrys' all function definitions with him, so that it can, when a function is called during execution, look up the respective commands and put them on top of the command stack.

The concept for the working area definition is very similar to the one for robots, with a similar, suitable datatype.

5 Possible future enhancements

- Unfortunately the current way of graphical output is quite slow, as the whole working area is redrawn after each step. One could try to change this by using several small wxHaskell-panels, one for each field, instead of just one big one, and then update only those that actually changed since the last step. But this would probably involve some serious rewriting of major parts of the GUI-code and as this idea has emerged quite late I didn't find the time to do that.

- On older computers with slow CPUs the log field is not updated properly. This is most probably related to the first point, but maybe there is a way to fix that without changing the drawing procedure. I have already experimented around a little with this, but I haven't found a satisfying solution.

- One could try to think of some new features for the robot. One idea, for instance, I have in mind is some kind of basic memory inside a robot, where each robot can save and modify one integer. But it is not yet clear to me what this could be used for after all.

- As the definition of the Haski language is now quite flexible, one could create modified parsers by slightly changing the `alex` and `happy` definitions. Thereby it should for example be possible to 'emulate' Pascal or C.

- Perhaps it would be nice to create some special fields for the working area, for example fields where the robot 'dies' and that he therefore has to avoid or where all the items a robot is carrying are 'stolen' from him.

At least the last point would not provide any additional learning aspect to the user, but could still be fun to tamper with.

6 Using Haskell

I have to admit that, at the beginning of the course, I was quite lost with regard to the labs and Haskell in general. After all I had only taken the standard course in functional programming in the first quarter, the level of which was considerably lower. But with some effort I managed to catch up and I got to appreciate the features of Haskell.

What I liked most when developing my project was probably the compactness of Haskell code; one can write complex things with very little code. On the other hand this can sometimes make understanding existing code quite strenuous. Another helpful feature is the strong type-checking. This often helps finding errors in your code rather early, it's also a good tool to think through certain processes on an abstract level.

What I did not like that much and found rather disturbing is the lack of some kind of global variables (at least there seems to be no straightforward way to do that) which are easily accessible after creation – although I am aware that this would probably collide with the purity of Haskell. But everytime, for example, I needed to access one of the wxHaskell-elements in a function, I had to add it as parameter to the function, changing the signature and updating every call to that function. This was especially unpleasent when I kept adding features to the working program, which often required a number of small changes to existing code.

Another drawback of using Haskell for this project is low speed. The program is, related to what it does, awfully slow. But on the one hand this might be related to wxHaskell, on the other hand perhaps there is a way to bypass this, as I described above.

Finally, as as last, minor point I would like to point out that working with wxHaskell is often quite wearisome, as it is mostly difficult to find the functionality you are looking for – but I can't really blame that on Haskell in general. . . ;-)

7 Conclusion

Altogether this project was fun to carry out and I can claim to have learned quite a bit, although, on the other hand, it was a lot of work and a number of obstacles emerged and had to be circumvented.

Perhaps this project will eventually be used in a basic programming course somewhere; it would be very interesting to see how well it actually 'performs', for example with respect to usability, and how understandable the user manual is. After all, I believe Haski the Robot should, in principle, be well-suited for that challenge.

References

[1] Richard E. Pattis. *Karel the Robot.* John Wiley and Sons, 1981.