Towards CNC Programming Using Haskell[⋆]

G. Arroyo[1], C. Ochoa[2], J. Silva[2], and G. Vidal[2]

[1] CIIDET, Av. Universidad 282 Pte. Centro, Santiago de Querétaro, Qro, Mexico.
garroyo@ciidet.edu.mx
[2] DSIC, Tech. University of Valencia, Camino de Vera s/n, E-46022 Valencia, Spain.
{cochoa,jsilva,gvidal}@dsic.upv.es

Abstract. Recent advances in *Computerized Numeric Control* (CNC) have allowed the manufacturing of products with high quality standards. Since CNC programs consist of a series of assembler-like instructions, several high-level languages (e.g., AutoLISP, APL, OMAC) have been proposed to raise the programming abstraction level. Unfortunately, the lack of a clean semantics prevents the development of formal tools for the analysis and manipulation of programs. In this work, we propose the use of Haskell for CNC programming. The declarative nature of Haskell provides an excellent basis to develop program analysis and manipulation tools and, most importantly, to formally prove their correctness.

1 Introduction

Computerized Numeric Control (CNC for short) machines have become the basis of many industrial processes. CNC machines include robots, production lines, and all those machines that are controlled by digital devices. Typically, CNC machines have a *machine control unit* (MCU) which inputs a CNC program and controls the behavior and movements of all the parts of the machine. Currently—as stated by the standard ISO 6983 [3]—CNC programs interpreted by MCUs are formed by an assembler-like code which is divided into single instructions called *G-codes* (see Fig. 1 below).

One of the main problems of CNC programming is their lack of *portability.* In general, each manufacturer introduces some extension to the standard G-codes in order to support the wide variety of functions and tools that CNC machines provide. Thus, when trying to reuse a CNC program, programmers have to tune it first for the MCU of their specific CNC machines. For example, even though both CNC machines `HASS VF-0` and `DM2016` are milling machines, the G-codes they accept are different because they belong to different manufacturers (e.g., the former is newer and is able to carry out a wider spectrum of tasks).

CNC programming is not an easy task since G-codes represent a low-level language without control statements, procedures, and many other advantages of modern high-level languages. In order to provide portability to CNC programs

[⋆] This work has been partially supported by CICYT TIC 2001-2705, by Generalitat Valenciana GRUPOS03/025, by EU-India Economic Cross Cultural Prog. ALA/95/23/2003/077-054, and by MCYT under grant HU2003-003.

and to raise the abstraction level of the language, there have been several proposals of intermediate languages, such as APL [9] and OMAC [7], from which G-codes can be automatically generated with compilers and post-processors. Unfortunately, the lack of a clean semantics in these languages prevents the development of formal tools for the analysis and manipulation of programs. Current CNC programming languages, such as Auto-Code [6] or AutoLISP [2, 12], allow us to completely specify CNC programs but do not permit to analyze program properties like, e.g., termination, or to formally use heuristics when defining the behavior of CNC machines (as it happens with autonomous robots).

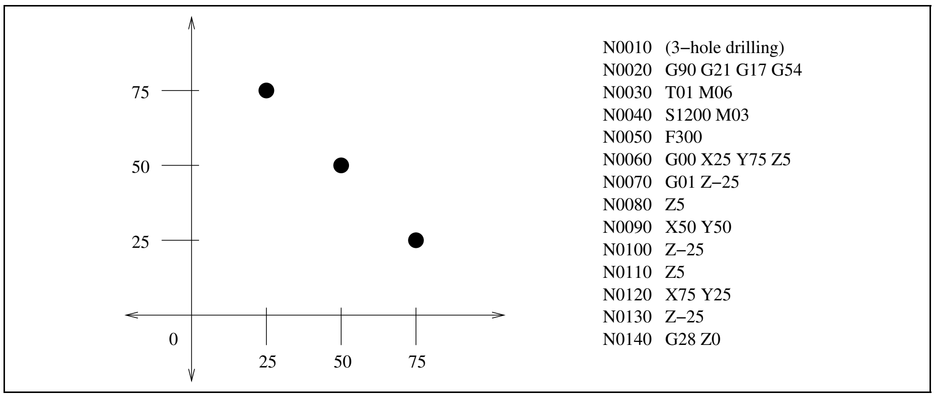
In this work, we propose the use of the *pure* functional language Haskell [11] to design CNC programs. Our choice relies on the fact that Haskell is a modern high-level language which provides a very convenient framework to produce and—formally—analyze and verify programs. Furthermore, it has many useful features such as, e.g., lazy evaluation (which allows us to cope with infinite data structures), higher-order constructs (i.e. the use of functions as first-class citizens, which allows us to easily define complex combinators), type classes for arranging together types of the same kind, etc. This paper presents our proposal for CNC programming using Haskell and shows the main advantages of Haskell over current languages which are used for the same purpose.

This paper is organized as follows. In the next section, we provide a brief review of CNC programming. Section 3 introduces the functional language Haskell. In Sect. 4, we illustrate the use of Haskell to represent CNC programs and, then, in Sect. 5, we enumerate some advantages of choosing Haskell over other existing languages. Some implementation details are discussed in Sect. 6 and, finally, conclusions and some directions for future work are presented in Sect. 7.

2 CNC: A Brief Review

Computer numerical control is the process of having a computer controlling the operation of a machine [17]. CNC machines typically replace (or work in conjunction with) some existing manufacturing processes. Almost all operations performed with conventional machine tools are programmable with CNC machines. For instance, with CNC machines we can perform motion control in linear—along a straight line—or rotary—along a circular path—axes.

A CNC program is composed by series of blocks containing one or more instructions, written in assembly-like format [13]. These blocks are executed in sequential order, step by step. Each instruction has a special meaning, as they get translated into a specific order for the machine. They usually begin with a letter indicating the type of activity the machine is intended to do, like F for feed rate, S for spindle speed, and X, Y and Z for axes motion. For any given CNC machine type, there are about 40-50 instructions that can be used on a regular basis. G words, commonly called G codes, are major address codes for preparatory functions, which involves tool movement and material removal. These include rapid moves, lineal and circular feed moves, and canned cycles. M words, commonly called M codes, are major address codes for miscellaneous functions



```
N0010  (3–hole drilling)
N0020  G90 G21 G17 G54
N0030  T01 M06
N0040  S1200 M03
N0050  F300
N0060  G00 X25 Y75 Z5
N0070  G01 Z–25
N0080  Z5
N0090  X50 Y50
N0100  Z–25
N0110  Z5
N0120  X75 Y25
N0130  Z–25
N0140  G28 Z0
```

Fig. 1. Simple CNC Program

that perform various instructions do not involving actual tool dimensional movement. These include spindle on and off, tool changes, coolant on and off, and other similar related functions. Most G and M-codes have been standardized, but some of them still have a different meaning for particular controllers.

As mentioned earlier, a CNC program is composed by series of blocks, where each block can contain several instructions. For example, `N0030 G01 X3.0 Y1.7` is a block with one instruction, indicating the machine to do a movement (linear interpolation) in the X and Y axes. Figure 1 shows an example of a simple CNC program for drilling three holes in a straight line.

3 An Overview of Haskell

Haskell is a general-purpose, purely functional programming language that provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern matching, list comprehensions, a monadic input/output system, and a rich set of primitive datatypes [11]. In Haskell, functions are defined by a sequence of rules of the form

$$f\ t_1\ \ldots\ t_n\ \ =\ \ e$$

where $t_1, \ldots, t_n$ are *constructor* terms and the right-hand side e is an *expression.*

The left-hand side must not contain multiple occurrences of the same variable. Constructor terms may contain variables and constructor symbols, i.e., symbols which are not defined by the program rules. Functions can also be defined by *conditional equations* which have the form

$$f\ t_1\ \ldots\ t_n \mid c\ =\ e$$

where the condition (or *guard*) c must be a Boolean function. Function definitions have a (perhaps implicit) declaration of its type, of the form

$$f\ ::\ a_1 \rightarrow\ a_2 \rightarrow \ldots \rightarrow a_n \rightarrow b$$

which means that f takes n elements of types $a_1, \ldots, a_n$ and returns an element of type b. For example, the following function returns the length of a given list:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Note that in this example, "a" is a *type variable* which stands for any type.

Local declarations can be defined by using the `let` or `where` constructs. For example, the following function returns `True` if the first argument is smaller than the length of the list being passed as a second argument, or `False` otherwise:

```
indexChecker :: Int -> [a] -> Bool
indexChecker n xs = n <= l where l = length xs
```

A Haskell function is *higher-order* if it takes a function as an argument, returns a function as a result, or both. For instance, `map` is a higher-order function that applies a given function to each element in a list:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Haskell provides a static type semantics, and even though it has several primitive datatypes (such as integers and floating-point numbers), it also provides a way of defining our own (possibly recursive) types using `data` declarations:

```
data Bool   = False  | True
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

For convenience, Haskell also provides a way to define *type synonyms*; i.e., names for commonly used types. Type synonyms are created using a `type` declaration. Here are some examples:

```
type String = [Char]           type Name    = String
type Person = (Name,Address)   data Address = None | Addr String
```

Type classes (or just classes) in Haskell provide a structured way to introduce *overloaded* functions that must be supported by any type that is an *instance* of that class. For example, the `Equality` class in Haskell is defined as follows:

```
class Eq a where
   (==) :: a -> a -> Bool
```

A type is made an instance of a class by defining the signature functions for the type, e.g., in order to make `Address` an instance of the `Equality` class:

```
instance Eq Address where
   None      == None      = True
   Addr st1  == Addr st2  = st1 == st2
   _         == _         = False
```

where `_` is a wildcard, used to introduce a default case.

We refer the interested reader to the report on the Haskell language [11] for a detailed description of all the features of the pure functional language Haskell.

G01: Moves the turret chuck along the XYZ axes. It can be followed by XYZ codes.
G90: Indicates that absolute positioning is being used.
G91: Indicates that incremental positioning is being used.
X(-)nn: used to move the turret chuck along the X axis
Y(-)nn: used to move the turret chuck along the Y axis
Z(-)nn: used to move the turret chuck along the Z axis

where `nn` indicates:
- the new absolute position in the corresponding axis, where (0,0,0) is a given reference point over the table (*absolute* positioning).
- the number of units in the current axis that the tool is being shifted (*incremental* positioning).

Fig. 2. Instructions set for simple CNC drilling machine

4 Using Haskell for CNC Programming

In this section, we illustrate the use of Haskell for CNC programming. By lack of space, we consider a *simple* CNC drilling machine which can just move the turret chuck in the X and Y axes (in order to position the drill bit), and in the Z axis (in order to make the hole). The machine also handles absolute and incremental positioning of the turret chuck.[3].

A CNC program for this machine consists of a header and a body. The header is optional and is usually a short comment, whilst the body is a list of blocks, where each block is identified by a number (`Nnnnn`) and can contain either one or more instructions or a comment, where comments are always parenthesized.

An instruction can contain one of the CNC codes shown in Fig. 2. In the following, we consider millimeters as measurement units.

For instance, a CNC program for drilling two holes at positions (10,10) and (15,15) is as follows:

```
N0010 (two-hole drilling)
N0020 G90
N0030 G01 Z05
N0040 X10 Y10
N0050 Z-25
N0060 Z05
N0070 X15 Y15
N0080 Z-25
N0090 Z05
```

In this example, the block `N0010` denotes a comment, `N0020` instructs the CNC machine to use absolute positioning, `N0030` moves the turret chuck 5mm over the table in the Z axis, `N0040` positions the turret chuck in the (10,10) coordinate (note that `X10 Y10` is a shortcut for `G01 X10 Y10`), `N0050` moves the turret chuck 25mm under the table in the Z axis, thus making a hole, `N0060` moves the turret chuck 5mm over the table in the Z axis, in order to be able to move it again in the XY axes, `N0070` positions the turret chuck in the (15,15) coordinate, and finally `N0080` and `N0090` create the second hole.

[3] A full example with the implementation of complete data structures needed to represent CNC programs can be found at `http://www.dsic.upv.es/~jsilva/cnc`.

```
data CNCprogram  = Header Body
data Header      = Maybe Comment
type Comment     = String
data Maybe a     = Nothing | Just a
type Body        = [Block]
data Block       = Com Comment | Code Command
type Command     = [Instruction]
data Instruction = X Int | Y Int | Z Int | G String
```

Fig. 3. Haskell data structures for representing a CNC program

It should be clear from this example that, when a big number of holes should be done, the amount of lines of code we have to write also increases considerably. The Haskell data structure intended to hold a CNC program is shown in Fig. 3.

For simplicity, our data structure does not contain information about block numbering. Nevertheless, given a list of blocks, it is straightforward to build a function that adds such numbering to each block. In our context, a CNC program is composed of a header and a body. A *header* is an optional comment (a `String` or the constructor `Nothing` when missing). A *body* is a set of blocks, each block being a comment or a set of instructions.

Figure 4 shows a function for drilling n holes in a straight line implemented in Haskell. Note that `nHolesLine` returns a list of blocks, instead of a complete CNC program as defined in Fig. 3 (the header is missing). Far from being a shortcoming, this is due to the fact that `nHolesLine` is integrated in an environment with many other different functions; therefore, there is a master function which builds the whole program by prompting the user for a header comment, if any, and integrating the code obtained from the different functions.

The function `nHolesLine` receives as parameters the number of holes, `n`, the initial XY coordinate and the corresponding increments in each axis. Then, this function generates a list of blocks by creating a list of `n` elements containing the *absolute* XY coordinates of the holes; it uses the higher-order function `map` to apply the function `makeHole` to each XY coordinate.

Note that, even though the list of `n` elements is created by first calling `createLine`—which generates an *infinite* list—only `n` elements of such a list are actually built. This is due to the lazy evaluation of Haskell (see Sect. 5). For instance, in order to make 50 holes, starting at position (10,10) and increasing 5mm in each axis per hole, we simply call function `nHolesLine` as follows:

```
nHolesLine 50 10 10 5 5
```

The same example using G codes requires about 150 lines of code. However, with Haskell functions it is very simple to change any of the parameters to achieve any straight line of holes to be drilled. In the same way, a lot of helpful functions can be created, in order to make different shapes like ellipses, grids, etc, that are able to work with other CNC machines like lathes and milling machines.

```
-- creates a [Block] containing the CNC instructions for
-- making n holes in a straight line
nHolesLine :: Int -> Int -> Int -> Int -> Int ->  [Block]
nHolesLine n x y incX incY = [posit,init] ++ concat (map makeHole nLine)
  where line    = createLine x y incX incY
        nLine   = finiteLine n line
        posit   = Code [G "90"]
        init    = Code [G "00",Z 5]

-- creates an infinite list of absolute coordinates
createLine :: Int -> Int -> Int -> Int -> [(Int,Int)]
createLine x y incX incY = (x,y):createLine (x+incX) (y+incY) incX incY

-- takes a list and returns a sublist containing the first n elements
finiteLine :: Int -> [a] -> [a]
finiteLine _ []     = []
finiteLine 0 _      = []
finiteLine n (x:xs) = x : finiteLine (n-1) xs

-- takes an XY coordinate and return a block containing
-- all instructions needed to make a hole at such position
makeHole :: (Int,Int) -> [Block]
makeHole (x,y) = [posXY,down,up]
   where up    = Code [Z 5]
         down  = Code [Z (-25)]
         posXY = Code [X x,Y y]
```

Fig. 4. Example program `nHolesLine`

5 Some Advantages of Haskell

In the following, we summarize the most significant advantages of Haskell for CNC programming:

Data structures and recursion. Haskell allows the definition of complex data structures (such as 3D geometric pieces) or iterative ones (such as hole meshes) that can be later manipulated and reused. A common way of defining and manipulating such data structures is to use recursion. In Fig. 4, two lists, `line` and `nLine`, are used to describe a set of specific positions of a piece. We recursively apply a defined function to them by using a simple command.

Polymorphism. Functions in Haskell can be polymorphic, i.e., they can be applied to different types (compare function `length`, which can be applied to lists of any kind). In our context, this means that some functions can be reused in many parts of the CNC program with different input data.

Higher-order functions. Higher-order facilities [4] are one of the main advantages of Haskell over the rest of languages currently used for CNC programming, incorporating a big amount of predefined higher-order functions allowing us to optimize and minimize the size of the code with a high expressiveness.

Laziness. Haskell follows a *lazy* evaluation model [1, 16], which means that functions are evaluated *on demand*. This is particularly useful when dealing with infinite data structures. For instance, consider a robot hand which performs a specific movement each time a piece is under it. Thanks to laziness, we can define the behavior of the robot hand by this infinite movement since it will only be evaluated as much as needed. To the best of our knowledge, all languages used in CNC programming lack of lazy evaluation (i.e., they are strict languages with *call by value* evaluation).

Type checking system. Haskell includes a standard type inference algorithm during compilation. Type checking can be very useful to detect errors in a CNC program, e.g., to detect that a drilling tool has not been separated from the piece surface after its use. Since CNC programs are usually employed in mass-production processes, program errors are very expensive. Therefore, having built-in type error checkers provided by a high-level compiler represents a significant advantage over usual CNC programs written by hand.

Type classes. Type classes in Haskell provide a structured way to introduce *overloaded*[4] functions that must be supported by any type that is an *instance* of that class [18]. This is a powerful concept that allows us, e.g, to arrange together data structures (representing CNC machines) having a similar functionality and to define standard functions for such types. When introducing a new data structure representing a CNC machine having a functionality similar to an existing one, we can just derive it from such a class, applying the existing functions to this data structure without re-writing any code.

Verification and heuristics. Haskell is a formal language with many facilities to prove the correctness of programs [14]. This represents the main advantage of our proposal compared with current languages being used for the same purpose. Thus, formal verification of CNC programs can be performed to demonstrate its termination, correctness, etc. Moreover, it makes possible the application of heuristics to define the behavior of CNC machines (as it happens with autonomous robots). This is subject of ongoing work and justifies our choice of Haskell for CNC programming.

Furthermore, Haskell is amenable to formal verification by using theorem provers such as Isabelle [10] or HOL [8], verification logics such as P-Logic [5], etc.

6 Implementation Remarks

The implementation of a Haskell library to design and manipulate CNC programs has been undertaken. This library currently contains:

- an XML DTD properly defined to completely represent any CNC program,
- a specific Haskell data structure equivalent to the DTD, and
- a set of functions to build and test CNC programs.

[4] While polymorphic functions have a *single* definition over many types, overloaded functions have *different* definitions for different types.

In order to guarantee the portability of the CNC programs produced in our setting, we have defined an XML DTD which is able to represent any CNC program since it contains all the syntactic constructs specified in the standard ISO 6983 [3], as well as the extensions proposed in [15, 17]. With this DTD, we can properly define any CNC program and, with some Haskell translation functions, automatically convert it to/from an equivalent Haskell data structure.

We have also implemented a library of functions which allows us to build and transform the data structure representing CNC programs in a convenient way. We provide several basic testing and debugging functions. Preliminary experiments are encouraging and point out the usefulness of our approach. More information (the implementation of Haskell library, the XML DTD and some examples) are publicly available at `http://www.dsic.upv.es/~jsilva/cnc`.

7 Conclusions and Future Work

This work proposes Haskell as a high-level language for the design and implementation of CNC programs. We have clarified its advantages over existing languages for CNC programming. Besides typical high-level features—such as control sequence, recursion, rich data structures, polymorphism, etc.—Haskell provides several advanced features like higher-order combinators, lazy evaluation, type classes, etc. Furthermore, Haskell offers a clean semantics which allows the development of formal tools.

We have defined a Haskell data structure which is able to represent any CNC program, allowing us to properly manipulate it by using Haskell features. Furthermore, we implemented an XML DTD which is fully equivalent to the Haskell data structure. This DTD ensures the portability of our programs among applications and platforms.

Preliminary experiments are encouraging and point out the usefulness of our approach. However, there is plenty of work to be done, like augmenting our library with other useful functions for making geometric figures, defining functions for other CNC machines (lathes, milling machines, etc), defining libraries for assisting the user in the post-processing of CNC programs, defining a graphical environment for simplifying the task of designing CNC programs, etc.

References

1. R. Bird. *Introduction to Functional Programming Using Haskell, 2nd Ed.* Prentice Hall Press, 1998.
2. H. Carr and R. Holt. The AutoLISP Platform for Computer Aided Design. In *40th Anniversary of Lisp Conference: Lisp in the Mainstream*, Berkeley, California, November 1998.
3. International Standardization for Organizations. Technical committee: ISO/TC 184/SC 1. Numerical control of machines – Program format and definition of address words, September 1982.
4. J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

5. R. Kieburtz. P-logic: Property Verification for Haskell Programs, 2002. The Programatica Project, http://www.cse.ogi.edu/PacSoft/projects/programatica/.
6. B. Kramer. *The AutoCADET's Guide to Visual LISP*. CMP Books, 2001.
7. J. Michaloski, S. Birla, C.J. Yen, R. Igou, and G. Weinert. An Open System Framework for Component-Based CNC Machines. *ACM Computing Surveys*, 32(23), 2000.
8. T.F. Melham M.J.C. Gordon. *TIntroduction to HOL*. Cambridge University Press, 1993.
9. T.P. Otto. An apl compiler. In International Conference on APL, editor, *Proceedings of the international conference on APL-Berlin-2000 conference*, pages 186–193. ACM Press - New York, NY, USA, 2000.
10. L.C. Paulson. The Foundation of a Generic Theorem Prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
11. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
12. R. Rawls and M. Hagen. *AutoLISP Programming: Principles and Techniques*. Goodheart-Willcox Co, 1998.
13. W. Seames. *CNC: Concepts and Programming*. Delmar Learning, 1994.
14. S. Thompson. Formulating Haskell. Technical Report 29-92*, University of Kent, Computing Laboratory, University of Kent, Canterbury, UK, November 1992.
15. J. Richard W. Maeder, V. Nguyen and J. Stark. Standardisation of the Manufacturing Process: the IMS STEP-NC Project. In *Proc. of the IPLnet Workshop*, 2002.
16. P. Wadler. The Essence of Functional Programming. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '92), Albequerque*. ACM Press, 1992.
17. M. Weck, J. Wolf, and D. Kiritsis. Step-nc The STEP Compliant NC Programming Interface: Evaluation and Improvement of the Modern Interface. In *Proc. of the ISM Project Forum 2001*, 2001.
18. M. Wenzel. Type Classes and Overloading in Higher-Order Logic. In E. Gunter and A. Felty, editors, *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, pages 307–322, Murray Hill, New Jersey, 1997.