# Generating CNC Code From a Domain Specific Language [*]

Gustavo Arroyo[1], J. Guadalupe Ramos[2], and Josep Silva[3]

[1] Centro Interdisciplinario de Investigación y Docencia en Educación Técnica, Av. Universidad 282, CP 76000, Santiago de Querétaro, México, garroyo@ciidet.edu.mx,
[2] Instituto Tecnológico de La Piedad, Av. Tecnológico 2000, CP 59310, La Piedad, Michoacán, México jgramos@pricemining.com
[3] Universidad Politécnica de Valencia, Camino de Vera s/n, C.P. 46022, Valencia, España jsilva@dsic.upv.es

*(Paper received on November 28, 2010, accepted on January 28, 2011)*

**Abstract.** Computerized Numeric Control (CNC) is an industrial language for the manufacturing of products. CNC programs are series of code that consist of assembler-like instructions and, consequently, they are low-level programs that require specialized developers in order to gain productivity in programs writing. In this work we introduce a domain specific language for the generation of CNC programs. The DSL itself has been developed in Curry, a declarative functional-logic language. Our DSL includes a set of functions that encapsulate CNC instructions raising the abstraction level, and therefore, improving productivity. It is designed in such a way that non-expert users can write CNC programs. We show how the use of a DSL allows us to perform the requirements capture of CNC systems and to reduce the gap between the requirements and the prototype. Finally, from our domain specific language we generate real CNC code in order to produce real world applications.

**Keywords:** CNC Programming; Domain Specific Languages

## 1 Introduction

The implementation of programs by employing computer programming languages is a technical task which is frequently delegated to programmers. Each programming language is founded on a set of technical features that influence the writing style of programs. Moreover, a program is profusely composed by technical sentences whose behavior strongly depends on a particular paradigm.

[*] This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant ACOMP/2010/042, by the *Universidad Politécnica de Valencia* (Program PAID-06-08), by SES-ANUIES and by the Mexican *Dirección General de Educación Superior Tecnológica* under grant 2369.09-P.

Hence, a programmer is a person with a solid preparation in the technical aspects of a particular programming language.

Historically, rising the abstraction level of programming languages has been a common goal in all paradigms. This change of abstraction level allows the hiding of difficult and cumbersome details closer to the machine, and it allows non-expert programmers to construct solutions. A simple method to rise the abstraction level of programs is to produce interfaces that hide low-level instructions that are grouped into modules composing the so-called libraries of the language. A more sophisticated method consists in developing a new programming language that encloses principles and abstractions in a consistent way that is inspired on the target domain of application. These programming languages are known as *Domain Specific Languages* (DSL's) [6] and they provide a powerful solution to construct abstractions of higher level. A DSL incorporates the most common abstractions of a domain and it offers combinators which permit to construct programs and to produce interactions among abstractions.

Orthogonally, when companies develop large (or even medium) new software systems, it is relatively common to discover that the prototype does not fit the requirements which were contracted (by the customer) at the beginning of the project. This produces a critical situation that implies additional costs and time, and that could be avoided by performing an adequate requirements capture.

Indeed the design of domain specific languages is considered an activity for requirements capture [1]. However, although some DSL's are good for resembling the abstractions of the domain, sometimes they do not produce the final code of the tool, and thus they are only useful as languages for specification.

In this work we focus on the development of a domain specific language for Computer Numerical Control (CNC). Nowadays, CNC 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 components of the machine. We consider that rising the abstraction level of the CNC language will allow us (1) to be able to develop more friendly CNC programs since we use abstractions of higher level, (2) to be more productive since we propose a library of functions, each of them encapsulating many simple CNC instructions, and (3) to demonstrate that DSL's are useful in order to produce real code and hence they permit to capture executable requirements.

We present our language as a set of functions that encapsulate CNC instructions, and that can generate real CNC code in order to produce real world applications.

As the host language of our DSL, we propose Curry [5], a multi-paradigm functional-logic language. Our choice relies on the fact that Curry is a high-level language which provides a very convenient framework to produce and (formally) analyze and verify programs. Moreover, it has many modern 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. Part of this work was inspired on [9] where authors present a DSL approach for routers specification in Curry. Some ideas in our designs are based on [3].

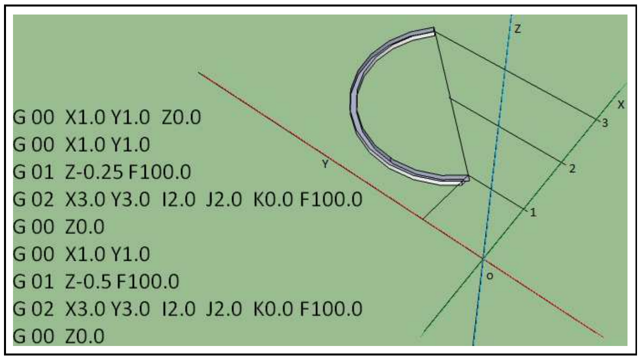
## 2 CNC: A brief review

Currently, as stated in standard ISO 6983 [4], CNC programs interpreted by MCUs are formed by an assembler-like code which is divided into single instructions called *G-codes* (see an example in Figure 1).

One of the main problems of CNC programming is the lack of portability. In general, each manufacturer of CNC machines introduces some extension to the standard G-codes due to the wide variety of functions and tools that CNC machines provide. Thus, when trying to reuse a CNC program, programmers have to first tune it 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 [3] (e.g., the former is newer and is able to carry out a wider spectrum of tasks).

CNC programming is a hard task because 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 and to raise the abstraction level of the language, there have been several proposals of intermediate languages, such as APL [8] and OMAC [7], from which G-codes can be automatically generated with compilers and post-processors.

Computer numerical control is the process of having a computer controlling the operation of a machine [11].



```
G 00  X1.0 Y1.0  Z0.0
G 00  X1.0 Y1.0
G 01  Z-0.25 F100.0
G 02  X3.0 Y3.0  I2.0  J2.0  K0.0 F100.0
G 00  Z0.0
G 00  X1.0 Y1.0
G 01  Z-0.5 F100.0
G 02  X3.0 Y3.0  I2.0  J2.0  K0.0 F100.0
G 00  Z0.0
```

**Fig. 1.** Simple CNC Program

A CNC program is a series of blocks containing one or more instructions, written in assembly-like format [10]. 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.

## 2.1 G and M Codes

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 that perform various instructions 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 a series of blocks, where each block can contain several instructions. For instance,

```
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 cutting a half circle by means of linear and circular interpolation G codes (G 01 and G 02).

## 2.2 An example

In this section, we illustrate a CNC programming example. Due to the lack of space, it is not possible to show a real example of a complete CNC program—explaining the meaning of all involved G and M codes. Instead, we consider a simple CNC milling machine which can move the turret chuck in the X, Y and Z axes. The machine also handles absolute and incremental positioning of the turret chuck.

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). This number can be optional, and can contain either one or more instructions or a comment, where comments are always parenthesized. An instruction will contain one of the CNC codes shown in Figure 2.

A CNC program for cutting a circular arc from (1,1) to (3,3) with center in (2,2) is as follows:

```
N0010 (cutting a circular arc XY plane) N0070 G00 X1.0 Y1.0
N0020 G90                               N0080 G01 Z − 0.5 F50.0
N0030 G00 X1.0 Y1.0 Z0.0                N0090 G02 X3.0 Y3.0 I2.0 J2.0 K0.0 F50.0
N0040 G01 Z − 0.25 F50.0                N0095 G00 Z0.0
N0050 G02 X3.0 Y3.0 I2.0 J2.0 K0.0 F50.0
```

In this example, the block N0010 denotes a comment; N0020 instructs the CNC machine that absolute positioning is being used; N0030 moves the turret

G01: moves the turret chuck along the XYZ axes. It can be followed by X, Y and Z codes, it represents a linear interpolation.
G02: moves the turret chuck along circular span lying in a plane parallel to one of the three principal planes of reference. It represents a clock wise circular interpolation.
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` is a number whose meaning depends on the type of positioning we are handling:

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

**Fig. 2.** Instructions set for a simple CNC milling machine

chuck at (1,1,0), 0 in the Z axis represents the surface of the piece to be machined; N0040 moves the turret chuck 0.25 mm under the table in the Z axis, it makes a hole in (1,1); N0050 starts the circular interpolation from the last position to the end position (3,3). The I,J,K words define the center of the arc. F represents the feed motion; N0060 moves up the turret chuck; N0070 moves the turret chuck at (1,1) position (the beginning of the arc); N0080 moves down the turret chuck, this time 0.5 millimeters down in the Z axis; N0090 starts again the circular arc 0.5 mm under the table in the Z axis; finally N0095 places the turret chuck at Z0.0. So we get a circular arc cut 0.5 mm depth (see Figure 1).

## 3 A DSL for CNC programming embedded in Curry

In this section we introduce a domain specific language for CNC code generation, which is based on the standard ISO 6983 [4]. We focus particularly in design aspects (taking into account that the DSL is developed in Curry), explaining the main functions developed and showing examples in order to illustrate the applicability of our DSL.

### 3.1 The functional-logic language Curry

Curry is a functional-logic language that inherits the best properties of the most important declarative programming paradigms, i.e., logic and functional programming paradigms. Curry combines features of both paradigms like: nested expressions, lazy evaluation, higher-order functions from the functional programming and logical variables, partial data structures, built-in search from the logic programming. Curry also has another powerful properties such as concurrent programming and besides, compared with functional programming: search, computing with partial information; compared with logic programming: more efficient evaluation due to the deterministic evaluation of functions. We refer the reader to [5] for an introduction to Curry.

## 3.2 Using Curry as the host language of the DSL

The Curry data structure that we define to hold a CNC program is shown in Figure 3.

```
data CNCprogram = Header Body
data Header  = Maybe Comment
type Comment = String
data Maybe a = Nothing | Just a
type Body    = [Command]
type Command = [Instruction]
data Instruction = N Int | G String | X Float | Y Float | Z Float
| U Float | V Float | W Float | P Float | Q Float | R Float | A Float
| B Float | C Float | I Float | J Float | K Float | F Float | S Float
| T Float | M String
```

**Fig. 3.** Curry data structure for representing a CNC program

In our setting, a CNC program is compound of a header and a body [2]. A `Header` is an optional comment—a `Comment` is a String—, when it is missing the `Nothing` constructor is used. A `Body` is a set of blocks, each block being a `Command` or a set of instructions. Finally, an `Instruction` is defined as one of the possible codes defined in the ISO 6983 standard [4].

For instance, we invoke the function `DrawHole` as follows:

```
DrawHole (1.0,1.0,0.0,1.5,10.0,0.5)
```

This command drills a hole in (1,1,0) with a 1.5 units depth, and a 0.5 units wide tool. It produces the following code:

```
[(G "00"), (X 1.0), (Y 1.0), (Z 0.0)]
[(G "01"), (Z (−0.25)), (F 10.0)]
[(G "01"), (Z (−0.5)), (F 10.0)]
[(G "01"), (Z (−0.75)), (F 10.0)]
[(G "01"), (Z (−1.0)), (F 10.0)]
[(G "01"), (Z (−1.25)), (F 10.0)]
[(G "01"), (Z (−1.5)), (F 10.0)]
```

The code moves the turret chuck to (1,1,0), then makes a hole with linear interpolation -0.25 units in Z axis to feed speed 10.0 units per minute.

Roughly analyzing the `DrawHole` function, it starts with the *InitPOS* procedure, it calls the *writefile* function definition which takes the *GotoXYZ* function as parameter. Note that, because Curry is a higher-order language, it can accept functions as arguments. *GotoXYZ* yields the first CNC command. Then it executes the *Perforate* procedure; because the width of the tool is less than the depth of the hole, the program control executes *FAUX_DrawHole* producing the rest of CNC commands until the depth is reached (details in the URL indicated at section 4).

It is worth to note that function *writefile* appends each CNC command in a file each time it is executed. Auxiliary functions are shown in Figure 4.

```
-- erasefile cleans CNC file
erasefile :: IO()
erasefile = writeFile "CNCCapture.txt" "%\n"
-- Go to X, Y, Z
GotoXYZ :: (Float, Float, Float) -> Command GotoXYZ
(x,y,z) = [G "00", X x, Y y, Z z]
-- Drill Z with a feedrate
DrawZ :: (Float, Float) -> Command
DrawZ (z,feedrate) = [G "01", Z (0-.z), F feedrate]
-- writefile writes CNC commands in a text file
writefile :: Command -> IO()
writefile(NewCommand) =appendFile "CNCCapture.txt" (show(NewCommand)++"\n")
```

**Fig. 4.** `writefile` and other auxiliar functions

## 3.3 Functions of the DSL

In this section we specify the function definitions of our DSL that is currently under development and we show the CNC code that it generates from a list of simple parameters. The intention of such functions is to simplify the way in which programming is performed for a piece of machining. Generally all the parts that take a manufacturing process can be formed from the combination of simple geometric figures, such as linear, circular and parabolic functions.

From this idea we are designing a DSL whose functions generate CNC code based on the ISO 6983 standard [4]. These DSL instructions produce: linear and circular cuts, make simple geometric figures (rectangles, circles), and canned (circular or rectangular). In the following we describe some representative functions and their parameters.

**Configuring functions.** These functions are useful for specifying general parameters during the programming process. For instance, they indicate whether the positioning is absolute or relative, whether they require measures in inches or millimeters and so on. Some Curry definitions of this functions are shown in the following chart:

```
—Selecting the kind of movement
Movement :: (String “Absolute” | “Relative”) –> IO()

—Selecting the kind of compensation
Compensation :: (String “Center” | “Left” | “Right”) –> IO()

—Selecting measurement unit
Unit :: (String “Millimeters” | “Inches”) –> IO()
```

**Function for making a hole in a specific point**

```
DrawHole :: (Float,Float,Float,Float,Float,Float) –> IO()
DrawHole (InitX,InitY,InitZ,Height,Feedrate,dx)
```

where *InitX, InitY, InitZ* represent the initial position of the CNC machine tool, *Height* is the depth of the drilling, *Feedrate* is the speed rate of the tool and *dx* represents the width of the cutting tool.

**Function for lineal cutting**

```
DrawLine :: (Float, Float, Float, Float, Float, Float, Float, Float) –> IO()
DrawLine (InitX, InitY, InitZ, EndX, EndY, Height, Feedrate, dx)
```

where *InitX, InitY, InitZ* indicate the initial position of the CNC machine tool, *EndX, EndY* represent the final position for the cutting, *Height* is the depth of the drilling, *Feedrate* is the cutting speed for the CNC machine tool and *dx* represents the width of the cutting tool.

**Function for making an arch of a circle**

```
DrawArc :: (Float, Float, Float, Float, Float, Float, Float, Float, Float, String, Float,
            Float) –> IO
DrawArc (Initx, Inity, Initz, Endx, Endy, i, j, k, Height, TSpin, Feedrate, dx)
```

where *InitX, InitY, InitZ* indicate the initial position of the CNC machine tool, *Endx, Endy* represent the X,Y final coordinates of the arch, *i, j, k* represent the coordinates of the circle, *Height* represents the depth of the drilling, *TSpin* represents the orientation of the spin, *Feedrate* represents the cutting speed of the CNC machine tool and *dx* represents the width of the cutting tool.

**Function for making a rectangular canned**

```
DrawBox :: (Float, Float, Float,Float, Float, Float, Float,Float) –> IO()
DrawBox (Initx, Inity, Initz, Length, Width, Height, Feed, dx)
```

where *InitX, InitY, InitZ* indicate the initial position of the CNC machine tool, *Length* represents the length of the box, *Width* represents the width of the box, *Height* represents the depth of the drilling, *Feedrate* represents the cutting speed and *dx* represents the width of the cutting tool.

**Function for making a circular canned**

```
DrawCylinder :: (Float, Float, Float, Float, Float, Float, Float)->IO()
DrawCylinder (i, j, k, Height, Radius, Feedrate, dx)
```

where *i, j, k* indicate the center of the circle, *Height* represents the depth of the drilling, *Feedrate* represents the cutting speed and *dx* represents the width of the cutting tool.

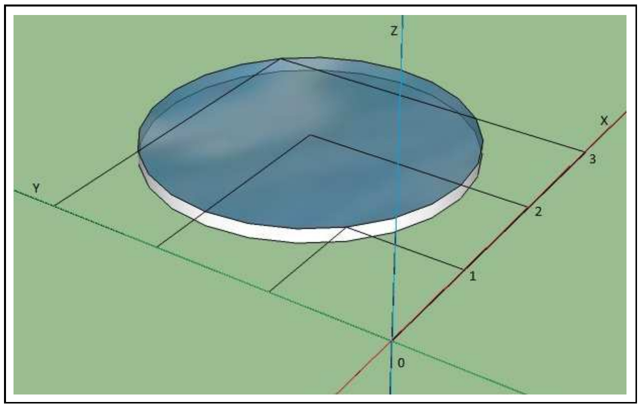


**Fig. 5.** Circular canned

## 3.4 Using DSL Functions

The following DSL Curry function

```
DrawCylinder (2.0,2.0,0.0,0.25,1.0,0.5)
```

makes a circular canned (see Figure 5). It defines a center of the circular sector in (2,2,0) with a 0.25 units depth with 1.0 units radius and a tool 0.5 units wide. This DSL function produces de following CNC code:

```
[(G "00"),(X 2.0),(Y 2.0),(Z 0.0)]
[(G "01"),(Z (-0.25)),(F 40.0)]
[(G "01"),(X 2.5),(F 120.0)]
[(G "03"),(X 2.5),(I 2.0),(J 2.0),(K 0.0),(F 120.0)]
[(G "01"),(X 3.0),(F 120.0)]
[(G "03"),(X 3.0),(I 2.0),(J 2.0),(K 0.0),(F 120.0)]
[(G "00"),(X 2.0),(Y 2.0),(Z 0.0)]
```

`G 03` makes a circular interpolation counter clock wise, because the tool is 0.5 wide with a hole in the center and just two `G 03` commands are enough to make the circular canned.

Another example is shown in Figure 1, where CNC code to produce a circular arc can be generated with a single call to the following DSL function:

```
DrawArc (1.0,1.0,1.0,3.0,3.0,2.0,2.0,0.0,0.5,"CW",100.0,0.5)
```

It generates the following CNC code:

```
[(G "00"),(X 1.0),(Y 1.0),(Z 1.0)]
[(G "00"),(X 1.0),(Y 1.0)]
[(G "01"),(Z (-0.25)),(F 100.0)]
[(G "02"),(X 3.0),(Y 3.0),(I 2.0),(J 2.0),(K 0.0),(F 100.0)]
[(G "00"),(Z 1.0)]
[(G "00"),(X 1.0),(Y 1.0)]
[(G "01"),(Z (-0.5)),(F 100.0)]
[(G "02"),(X 3.0),(Y 3.0),(I 2.0),(J 2.0),(K 0.0),(F 100.0)]
[(G "00"),(Z 1.0)]
```

Clearly, the DSL improves the readability of programs, and reduces the size of the code. In this example, using the DSL allows us to perform all the work with a single DSL instruction. Figure 6 shows the Curry code that implements the `DrawArc` DSL function. Observe that an important advantage of the DSL is that we can re-implement all the functions for another specific CNC machine but keeping the same signature. This means that the DSL provides a portability that allows us to use the same DSL programs in different CNC machines.

```
 -- DrawArc draws an arc where i,j,k are the origin of the radio
 -- TSpin:  CW=ClockWise CCW=CounterClockWise
 DrawArc :: (Float, Float, Float, Float, Float,Float,Float,Float,Float,
             String, Float, Float) -> IO()
 DrawArc (Initx, Inity, Initz, Endx, Endy, i, j, k, Height, TSpin,
           Feedrate, dx) = do InitPOS
                              Perforate
 where
  InitPOS = writefile(GotoXYZ(Initx, Inity, Initz))
  Perforate =  if (Height<dx) then do
      writefile(DrawZ(Height,Feedrate))
      writefile([Spin(TSpin),X Endx, Y Endy, I i, J j, K k, F Feedrate])
-- if tool width is less than depth
       else do FAUX_DrawArc(Initx,Inity,Initz,Endx,
               Endy,i,j,k,Height,TSpin,Feedrate,(0.5*.dx),(0.5*.dx))
```

**Fig. 6.** The Curry code of the DSL function `DrawArc`

## 4 Conclusions

In this work we introduced a DSL developed in Curry as a high-level language for the design of CNC programs. We have shown that using the DSL provides important advantages over pure CNC programming.

The DSL provides two main advantages in the development of CNC programs. The first advantage is the possibility of programming at a high abstraction level. This improves the productivity, i.e., to provide more commands to the CNC machine (more CNC code) with less instructions and readability. This goal has been reached since the DSL functions encapsulate procedures that automatically produce CNC code for a specific and non-simple task.

The second advantage is reducing the gap between user requirements and the developed prototypes. This situation is a common trouble in the software engineering field because the analyst that captures the requirements and the programmer that programs the prototype are different persons. The use of DSLs during the analysis allows the analyst to produce rapid prototypes without the need of being a CNC programmer. This can be done thanks to the higher abstraction level of the DSL that allows requirements to become the so-called executable requirements.

Preliminary experiments are encouraging and point out the usefulness of our approach. However, there is plenty of work to be done, such as augmenting our library with other useful functions for making geometric figures, defining functions for other CNC machines (lathes, milling machines, etc), defining a graphical environment for simplifying the task of designing CNC programs, etc. Some other examples and the source code of our Curry DSL library is publicly available at the following URL: http://www.ciidet.edu.mx/Sitio_DSL/CNC_DSL.htm

## Acknowledgment

We are grateful to Antonio Ávalos Olguín, Professor of the Technological Institute of Querétaro for his comments and support in the validation of CNC programming. Also to Marco Antonio Llanes Rodríguez for his valuable cooperation in the development of the Curry library.

## References

1. C. Wohlin A. Aurum. *Engineering and Managing Software Requirements*. Springer-Verlag, 2005.
2. G. Arroyo. Diseño de un Lenguaje de Especificación para CNC. IP Report (in Spanish), DSIC-UPV, 2004.
3. G. Arroyo, C. Ochoa, J. Silva, and G. Vidal. Towards CNC Programming Using Haskell. In Lemaitre Christian, Reyes Carlos A., and Gonzalez Jesus A., editors, *Advances in Artificial Intelligence-IBERAMIA 2004*, pages 386–395. Springer LNCS 3315, 2004.
4. International Standardization for Organizations. Technical committee: ISO 6983-1/TC 184/SC 1. Numerical control of machines – Program format and definition of address words, September 1982.
5. M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at http://www-ps.informatik.uni-kiel.de/currywiki/, 2010.
6. P. Hudak. Modular Domain Specific Languages and Tools. In *Procedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.
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.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.
9. J. G. Ramos, J. Silva, and G. Vidal. An Embedded Language Approach to Router Specification in Curry. In *SOFSEM*, pages 277–288, 2004.
10. W. Seames. *CNC: Concepts and Programming*. Delmar Learning, 1994.
11. 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.