Deriving Applicative Programs from Formal Specifications

Silvio Lemos Meira
Departamento de Informática
Centro de Ciências Exatas e da Natureza
Universidade Federal de Pernambuco
50739, Recife PE, Brasil.

ABSTRACT

We introduce a notation to write functional (declarative, applicative) programs and consider ways in which to derive applicative implementations of formal specifications written using the Z formalism.
In this paper we consider Z as the utmost level of abstraction, and A, the functional language, as the most operational formalism available to the programmer. Our goal is to consider the properties of a formal programming environment in which to derive functional programs from mathematical specifications of their properties, as a (first step) of building reliable pieces of software. We also consider the intricacies of automating (parts of) the process.

RESUMO

Apresentamos uma notaçao para escrever programas declarativos, e consideramos como derivar programas aplicativos a partir de especificaçoes formais escritas em Z. Aqui, Z é o nível mais abstrato, e A, uma linguagem funcional de alta ordem, o mais concreto. O objetivo é considerar as propriedades de um ambiente formal de desenvolvimento de software, onde o desenvolvimento de programas funcionais seria o primeiro passo na construçao de software confiável.

1. INTRODUCTION

1.0 Essential Terminology

The Z here refers to the Z specification language [Hayes 86], and A is our own in-house applicative language being designed and implemented at UFPE. Both formalisms are described elsewhere [Meira 87b, Meira 87e].
Some of the terminology in this paper comes from [Bjorner 87], and much of it is classified as "common knowledge". We use the terms in the meanings ascribed in the aforementioned article, without any attempt to adopt them as standards.

1.1 The Software Cycle and Models of the Software Development Process

The software development process is part of the software cycle. The phases treated as "development" here are specification, design and implementation. We will consider some of the different approaches taken in modelling the process of software development "in the large". There are a few models, the most commonly used being derived from the waterfall[Cohen 86] and modified versions of it. Recent developments such as RAISE[Bjorner 87] (an acronym for Rigorous Approach to Industrial Software Development), are based in a sub-cycle of that model, encompassing the three steps of the "waterfall", which are called the "software development cycle", on the idea of "project graphs"[Bjorner 87] and in denotational semantic methods[Jones 86, Bjorner 87] to specify, design and deal with some implementation matters of the software in question.

As we have said before, our main interest is in **formal** methods for developing software, and in order to do that, we need first to have a model in which to work with the objects of that class. That means that we may have to accept a "world" model of the software development process, or we may not. We believe that some of the more esoteric models of software development have been used so far for absolute lacking of better models, tools and theories that would support a simpler approach.
Of course, if the "customer" that goes into a software house in need of a program could write fully formal specifications of his software, say, in VDM[Jones 86], that would be good news for all concerned, including himself. As it happens, it is not likely that this is the way things will happen, and we are bound to face an informal, unstructured description of what is needed. We are not going into this matter here, but a very important problem to deal with, when developing software, is getting to know what the user requirements are, exactly. This will be dealt with in section 1.2, and more extensively in a companion paper[Meira 87a].

In what follows we propose a three-step model for the software development cycle, which we call SP|one (for **specification**, design/**prototyping**, and **implementation**). The method is being used to design non-reactive systems, and it is based in a mathematical formalism for specification of problem/program properties, a purely applicative programming language as both prototyping and implementation language, and a method of transforming formal specifications into functional programs, which can be viewed, in a first stage, as an **executable specification**. Following Bjorner's work at DDC, Denmark, the setting for this activity is the idea of **Software Development Graphs**[Bjorner 87], which is explained at length elsewhere[Meira 87].

1.2 Informal Specifications

Informal specifications, unfortunately, are what most programmers are used to see as the source code which they must compile into working and correct programs. In a real life project, one has to deal with the problem of acquiring the right informal specification, which is not trivial, due to the gap that must bridged by programmer and user to understand each other.
After one has the (right) informal specification, there is the problem of formalizing it somehow. The choice of methods and formalisms is wide (VDM, PAISley[Zave 84], obj[Goguen 82] and me too[Henderson 86] are but a few examples). The method to be chosen has to have the **right** level of abstraction, which is something not very easily defined. Also, it must be possible to translate (this can be seen as some form of compilation) the initial (informal) specification into its formal counterpart, in such a way that the intuitive meaning of the former is at least apparent in the latter.
That is to say that one expects an average reader, with a mathematical background, to be able to read the more abstract levels of the formal specification.

1.3 Formal Specifications

Formal specifications are a need in software development as drawings and formulae in engineering. No one in their right mind would fly in an airplane that they knew had been built using the techniques that we use to build computer programs most of the time. As we are now using computer programs to fly airplanes, it is about time we start to think about using design and construction methods for our programs which are at least as reliable as the ones used in (non computer aided) engineering.
Here we are to face fierce opposition in our way to enforce **formal** software development. Most people's attitude to the subject is that we can do it without mathematics. This is like saying that Marco Polo went to China using nothing but intuition and senses. That is perfectly fine for his time, but I bet he could not take a supertanker anywhere, nowadays, with the same apparatus he used then. That is indeed our problem in software engineering today.

We ought to be using the appropriate tools to solve the problems we have...

1.4 Prototyping and Programming

We should establish the formal relationship between the three activities (specification, design/prototyping and programming) that we assume are software development. It is widely accepted that the developer profits from a prototype instead of going all the way and writing a full implementation. In SPI[one], the aim is to derive functional, executable specifications (i.e., prototypes) from formal specifications written in Z. A small example of this is given in Section 3.

2. Z

Why Z instead of obj, VDM or whatever? A (partial) answer to that is that Z is simpler, open, and it is not a standard, which means we can experiment with it and (may be) have our Z, as it is the case of some groups around the world. Z is a model based formalism like VDM, but the aim is not proof-theoretical as in the latter. The general ideal is to have modular mathematics to communicate the specification, the design and the implementation, at several levels, to the various groups involved in the construction of a large piece of software.

2.1 The Mathematical/Computational Notation

A small problem here is to choose a notation for the mathematical symbols and diagrams in Z, as one would certainly want to have a parser and type checker that could treat specifications written in the formalism. This would be a first step in the direction of having transformation systems that could take Z as input and (most probably with user cooperation) produce runnable programs that conform to the specification. In the long run, this is the aim of this project. Our group's first attempt to define such notation is discussed in [Meira 87e]. Due to space constraints, we are going to use a notation akin to that one here without any further explanation.

2.2 An Example

The example we have chosen to specify is a portuguese version of Unix's spell (Orto[Melo Neto 87]) being developed at UFPE.
The (very) sketchy informal specification goes as follows:

0) given a sorted dictionary of (portuguese) words, we want a program that can
1) insert new words in the dictionary;
2) check whether a given word occurs in the dictionary;
3) given an input text formatted by a word processor, filter the wp's commands and give as result a sorted list of the input's words which do not occur in the dictionary.

The specification goes thus

```
SCHEMA Basics;
DEFINE
  letter = ['A'..'Z','a'..'z'];     (* all letters, accented ones as well *)
  Word   = list Letter;             (* word is a list -or seq- of letters *)
  File   = list Word;               (* file is -just- a list of words     *)
DECLARE
  Dict   : File;
PREDICATE
  ordered Dict                    & (* the dictionary is ordered, and *)
  mk_file (mk_set Dict) = Dict      (* words occur just once          *)
END Basics.
```

The "modules" of formal specifications in Z are called "SCHEMA". The schema above declares a few types, such as word and file, and a variable, the dictionary (Dict). The predicate states that

the dictionary is ordered, and also that words occur just once. The auxiliary functions (eg. mk_set) used in the predicate are assumed defined elsewhere, say in a "system" module.
The schema above states the invariant of the specification, which is imported by all other schemas. Readers familiar with *usual* Z will notice that we give a very liberal interpretation of the formalism here, which puts some extra stress on the specification of functions (one could call it fizZ - a functional interpretation of Z!). Only one *variable* will be talked about, representing the dictionary.

```
SCHEMA Insert;
INCLUDE Basics, Basics';          (* the ' primes all of Basics     *)
DECLARE
  insert : Word -> File -> File;
PREDICATE
  FORALL x : Word .
  ( Dict' = insert x Dict &
    Dict' = Dict $$ x       )    (* $$ means include or substitute *)
                                 (* right into left                *)
END Insert.

SCHEMA Find;
INCLUDE Basics;
DECLARE
  find : Word -> File -> Boolean;
PREDICATE
  FORALL x : Word .
  ( IF x IN (mk_set Dict) THEN find x Dict = TRUE
                          ELSE find x Dict = FALSE &
    Dict' = Dict                                    )
END Find.

SCHEMA Filter;
INCLUDE Basics, Basics';
CONST
  Wpcmds = [".au", ".pp", ".bf", ".ti"]
DECLARE
  filter : File -> File;
PREDICATE
  FORALL f : File.
  (  filter f = mk_file (mk_set f - Wpcmds)
     Dict' = Dict )
END Filter.
```

Here note that the specification for filter just says that it removes all word processor commands from the input text. No mention is made to the order of the output text relative to the input.

```
SCHEMA Spell;
INCLUDE Basics, Basics';
DECLARE
  spell : File -> File;
PREDICATE
  FORALL f : File.
  (  spell f = sort (mk_file (mk_set (filter f) - Dict))
     Dict'   = Dict )
END Spell.
```

It should be said that Z's usual schemas deal with variables, which are part of a state, the object of the formal specification. Our reading is functional, so we tend to say much more about the functions, in this case, that will use the only *variable*, the dictionary.

3. A

3.0 Why a New Applicative Language?

There is a number of applicative languages in the run already. It is a risk, in a sense, to design yet a new one, and it might also be a fruitless exercise to do so.

We have decided to design and implement a new language for a number of reasons. First, because some features that we'd like to see in ONE language are spread over several. These include comprehensions (Miranda and Orwell), non- and controlled strictness(Miranda), polymorphic strong typing (Miranda, ML and Hope), functional unification (Hope), modules and separate compilation(LML and Miranda), algebraic and abstract data types (Miranda and ML) and inheritance (Pebble and FOOPS), and more exotic capabilities such as Darlington and While's modal "restrictions to evaluation" (currently implemented in Hope).
If we had to choose a new language, it would have been Miranda [Turner 86], but that was not possible because of implementation rights and problems in porting it to Brazilian versions of ATT's Unix operating system. Also, we could not get a copy of Wadler's [Wadler 86] Orwell system for more or less the same reason.

Apart from having a go at language design, we also wanted to have a testbed in which to sort out our ideas in developing functional "executable specifications" (or prototypes) from formal specifications, i.e., the language might have influence of Z in its basis. Another possible aim is to try some clever implementation ideas by Rafael Lins [Lins 86], from his work in categorical combinators.

At last, to our best knowledge, there is no previous experience in the implementation of purely functional languages with the features described above in this country. The exercise proposed herein will also fulfill some of the lack of experience in this subject, which is of greater importance by the day.
In what follows, we give an informal description of A, which is to be thought as a very preliminary partial description of its features. The language's formal syntactic, pragmatic and semantic descriptions will follow[Meira 87b].

3.1 The Data Objects

In A, types are sets of values, like in Fun[Cardelli 85]. We do not enter into the subject here, but it is necessary to say this minimum in order to account for lists, tuples and other structured types, polymorphism, type parameterization and inference. It is planned that A will have an ML-like type system.
The basic data types in A will be **bool**, **int**, **real** and **char**, which are the language types corresponding to truth values, integers, reals and characters, respectively. Examples of each are

```
TRUE, FALSE,    -1, 1987,    1.0, 3.1415926,    'a', 'z'.
```

One basic structure is the list, created by the type constructor **list**, and one can have lists of any type of object (of course including lists). Lists are homogeneous data structures. In particular, a **string** is a list char, abbreviated that way for convenience. Examples of lists are

```
[1,2,3]                          which has type list int
[[1],[2,3],[4,5,6]]              list (list int)
['d','i','a','n','a']            list char, the same as string.
```

There is an alternative representation for the last list, as "diana".
To start with, we are not going to bother about arrays in A. Arrays can be represented as lists, bar access time, which is constant for single access on arrays, and linear on lists. A might have (applicative, as in [Meira 85]) arrays in the future. There is no promise.

The way of grouping objects of different types in a structure is to put them together in a tuple, like

```
11, "good", 1.0!
```

whose type is

```
tuple [int, string, real].
```

The argument to the constructor tuple is a list of types. tuple is a constructor of product types, and tuples could be defined, of course, in terms of the usual domain operators. Note that we can use A's tuples as Pascal's records, so we do not bother with the record mechanism here.

3.2 Abstract Data Types

Much of the power of today's programming languages is in the provisions they make for data abstractions. It is, of course, much easier to talk about the true properties of a tree, than about the details of its implementation in assembler. Like Miranda, A provides two sorts of abstractions: the first are the **algebraic** data types, where the programmer builds algebras to describe her data types, which are visible, at constructor level, from the outside. Trees, for once, can be described as

```
ALGEBRA tree;
  tree x = _nil | _node x (tree x) (tree x)
END tree.
```

Note that the previous equation describes parameterized trees (`tree x`) which can be used to represent trees of any data object. The underscored names are used (by definition) as constructors of the data type, hence also valid in pattern matching objects of that type. The type described above is a free algebra, i.e., there are no restrictions to the behaviour of the constructors. We could define an ordered binary tree algebraic type using laws to enforce a desired behaviour of the constructors, such as in

```
ALGEBRA otree x;
  otree x = _onil | _onode x (otree x) (otree x) | _insert x (tree x);
LAWS
  _insert a _nil           =>  _onode a _nil _nil;
  _insert a (_onode b l r) =>  _onode b (_insert a l) r, a <= b;
  _insert a (_onode b l r) =>  _onode b r (_insert a r), a >  b;
END otree.
```

Here, if you use the `_insert` constructor, the tree is ordered on its own! Normal trees can be built using the ordinary `_node` constructor. Note that although one can build trees using the `_insert` constructor, all instances of it are rewritten, by the laws, into instances of `_node`.

Abstract types, on their turn, have hidden implementation parts, and are used according to the "exported" definition. A balanced tree type could be defined as an **abstype** by

```
ABSTYPE baltree x;
VISIBLE
  nil    :      baltree x;
  empty  :      baltree x -> bool;
  insert : x -> baltree x -> baltree x;
  delete : x -> baltree x -> baltree x;
  find   : x -> baltree x -> bool;
HIDDEN
TYPE
  baltree x = tree x;  (* balanced trees are algebraic trees *)
END;
DEFINE nil;
  nil = _nil;
END nil;
DEFINE empty;
  empty x = x = _nil;
END empty;
DEFINE insert : a -> baltree a -> baltree a;
  insert x t = balance (put x t);
DEFINE insert;
  (* et coetera *)
END baltree.
```

3.3 Functions, Comprehensions and Sections

Functions are defined in A by writing their defining equations, just as we have done in the HIDDEN part of the abstract data type above. Ideally, there would be no order in the definitions, as the guards in the equations should be mutually exclusive. To simplify writing, we assume that the equations are compiled from top to bottom. All function definitions are curried, higher order and maybe polymorphic. A embodies the concepts of sections (partially parameterized operators as in Orwell) and comprehensions (ZF-expressions in Miranda), and has a call-by-need semantics throughout. It is planned (but there is no guarantee) that it will have unification as in extended Hope[Darlington 86]. We now show a few standard definitions in A (a whole set of definitions is available from StdLib):

```
MODULE Standard;
(*
  An A MODULE is made up by IMPORT and EXPORT lists, followed by
  definitions of types (TYPE, ALGEBRA and ABSTYPE) and functions (DEFINE).
  Each function is introduced by a DEFINE...END pair. Giving the type
  of the function in the DEFINE line is optional, as the compiler will
  deduce its type. Semicolons are terminators
*)
EXPORT filter, map, reduce, sort;
DEFINE map : (x -> y) -> list x -> list y;
  map f x = [ f a ; a <- x]
  (* read as " list of f applied to a, for all a taken from x " *)
END map;
DEFINE filter;
  filter c x = [ a ; a <- x ; c a ];
  (* no need to declare types, as they can be deduced by the compiler *)
END filter;
DEFINE reduce;
  reduce op id [] = id;
  reduce op id (a:x) = op a (reduce op id x);
END reduce;
DEFINE sort; (* this implements bubble sort... *)
  sort = fix swap;
END sort;
DEFINE fix;  (* visible only in this module, it is not exported *)
  fix f x = x, x = fax;
          = fix f fax;
WHERE fax = f x; (* this is local to fix, not visible outside *)
END fix;
DEFINE swap;
  swap x       = x                , #x <= 1;
  swap (a:b:x) = a : swap (b:x), a <= b
               = b : swap (a:x), a >  b
END swap;
END Standard.
```

3.4 Modules

As it can be deduced by the informal presentation above, A's module mechanism is very much like that of Modula2[Wirth 83]. Furthermore, algebras and abstypes are defined inside modules, as it is the case for everything else in the language. For the purpose of this paper, we can assume that this is indeed the case. A more detailed explanation is given in [Meira 87b].

3.5 A Simple Implementation of A

True to the main aim of this project, we are starting it by creating a formal description of A. The method being used is to write a "program" which implements the denotational semantics of the language, as in [Allison 83]. Due to the features proposed for A, a compiler is a must. At the time of writing, we only have an interpreter for KRC[Croft 84], under VMS on the VAX, to which the A definitions have to be hand translated in order to be executed.

4. From Z to A

The starting point here is that we can easily transform the requirements stated in Z into function definitions in A which satisfy the stated requirements.
Looking at a Z schema, what we see is a number of type definitions, variable declarations and predicates that must be satisfied. On the core of the predicates, we find functions (yet to be defined), which, given the truth of the pre-conditions, their application will result in the truth of the post-conditions. The job to be done when programming is to define the functions which are true to the specifications.
A is a very high level language. The denotational character of the language makes it easier to match specification blocs with language modules and function definitions.
For example, given the specification of sorting as

```
SCHEMA Sort;
INCLUDE Basics;
DECLARE
  sort : File -) File;
PREDICATE
  FORALL f : File .
  ( sort f IN permutations f &
    ordered (sort f)          )
END Sort.
```

Assuming the usual specifications for *permutations* and *ordered*, the following module implements the specification above, modulo the correctness of the definitions of permutation and ordered, given that sort is correct by construction:

```
MODULE Sort;
FROM StdLib IMPORT head;
EXPORT File, sort, Word;
TYPE
  File = list Word;
  Word = string;
END;
DEFINE sort : File -) File;
  sort x = head [ x (- permutations x; ordered x ];
END sort;
DEFINE ordered;
  ordered []      = TRUE;
  ordered [a]     = TRUE;
  ordered (a:b:x) = a (= b & ordered (b:x);
  (* Note that the definition is polymorphic, of type list a -) bool.
     Used above, the general type is instantiated to File -) bool      *)
END ordered;
DEFINE permutations;
  permutations [] = [[]];
  permutations x  = [ a:p ; a (- x; p (- permutations (x - [a]) ];
END permutations;
END Sort.
```

The reader has certainly noticed that the definition of sort is not very efficient. This was not the aim of our first definition. The process of transforming specifications into definitions goes exactly the way we have tried to show above. First one defines the data types that will be used in the definitions, and they will be isomorphic to the types used in the specifications. Next one writes what could be properly called executable specifications, i.e., definitions in which one has only concerned oneself with correctness in view of the specification.

4.1 A Programming Exercise: Orto in A

The definitions below have been derived from the formal specification in section 2.2. We feel they need not much of an explanation here, given that the "derivation" in question is straightforward.

```
MODULE Orto;
FROM StdLib IMPORT or, read;
FROM Sort   IMPORT Word, File, sort;
EXPORT find, insert, spell;
DEFINE find : Word -> File -> bool;
  find x fil = or [ x = a; a <- x ];
END find;
DEFINE insert : Word -> File -> File;
  insert a []    = [a];
  insert a (b:y) = a:b:y,          a <= b;
                 = b:insert a y, a >  b;
END insert;
DEFINE spell : File -> File;
  spell = (- dict) . mkset . sort . (- wpends)
WHERE
  dict = read Word "dictionary";
  (* the type of read is a -> string -> list a,
     the operational semantics of the function is obvious *)
  wpends = [".au", ".pp", ".bf", ".ti"];
END spell;
DEFINE mkset : File -> File;
  mkset []      = [];
  mkset [a]     = [a];
  mkset (a:a:x) = mkset (a:x);
  mkset (a:b:x) = a:mkset (b:x);
END mkset;
END Orto.
```

mkset does not really makes sets. It takes an ordered list and produces one where each element occurs at most once. A sorting function could take care of that as well.

5. Conclusions

It is expected that one can indeed produce some of the deliverables advertised here, in a form that can be used by industry. If not, at least we can put some hard questions about the technology of programming in the large and see how a particular pair of paradigms, Z and A in this case, is suited to answer them.

In view of what we know a priori, we do not expect great progress, initially, in the area of automated generation of programs from specifications, and that is not our aim here. We expect to produce some generic tools that can verify whether a program, derived from a formal specification, is true to it. We are well prepared to accept that even this is not a simple thing to do, and we are not going to try it first. Hand tranformation of specifications in definitions must be tried first, to discover the "natural" approach. There might not be something that is, at the same time, natural and simple.

However, we feel that there are a number of things that can and must be done, here and now.

The software technicians and engineers that we have at work, nowadays, are most of the time working in skyscrapers which they have no plan of, when they do the structural engineering, building, or the maintenance, and they do not know what they are supposed to get out of their work. As a result, most bugs found in "modern" systems end up being "features", for lacking of a proper specification of the system and proper design of its implementation.

We would like to contend that there is no way in which to specify, informally, a huge building and get it right in the end, and the same is valid for large pieces of software.
As there are drawings, equations, cross-sections and the like for buildings, there must be equations, specifications and drawings for software. It is not for being abstract that software is less complex than buildings or mechanical artifacts. **If anything, it is more complex.**

The "execution" of the program of work hinted here, applied to some real systems that programmers would do in practice (such as the Hypertext project described in [Meira 87d]), would most certainly generate a wealth of knowledge in the fields of software engineering it deals with (language design and implementation, formalisms and methodologies for software specification, design and implementation, and more), and it is a long term programme of work. It will not be fulfilled with one or two papers, it is more likely to take years of work to grasp the real complexity of the systems we are deemed to deal with. Furthermore, there is not a solution, but many.

At last, but not at least, readers are warned that this paper skims the surface of work in progress. It is not the case that everything will change in the future, but we have not entered steady state yet.

6. Acknowledgements

This work owes much to the seminars of the Functional Programming group at UFPE (Augusto Sampaio, Carmen Salazar, Maria Liege Sombra, Nelson Asfora and Roberto Souto Maior). Special thanks go to Augusto Sampaio, for forcing me to explain some of "my" Z to him, which has cleared the presentation of Section 2.

7. References

[Allison 83]
Programming Denotational Semantics. Computer J., 1983.

[Bjorner 87]
On the Use of Formal Methods in Software Development. 9th. Intl. Conf. on Softw. Eng., Monterey, CA, Apr. 1987.

[Cardelli 85]
On Understanding Types, Data Abstraction and Polymorphism. Comp. Surv. 17 (4), Dec. 1985.

[Cohen 87]
Cohen, B., Harwood, W. T., Jackson, M. I.: *The Specification of Complex Systems.* Addison-Wesley, Wokingham, (UK), 1986.

[Croft 84]
Croft, S.: *Implementation of Functional Languages.* MSc Thesis, Rutherford College, University of Kent at Canterbury, 1985.

[Darlington 86]
Darlington, J, Field, A. J., Pull, H.: *The Unification of Functional and Logic Languages.* IC-CS Internal Report, Imperial College, London, (UK), 1986.

[Goguen 82]
Goguen, J., Meseguer, J. A.: *Rapid Prototyping in the OBJ Executable Specification Language,* SRI Intl., TR CSL-137, Aug. 1982.

[Hayes 87]
Hayes, I. (ed.): *Specification Case Studies.* Prentice Hall Intl (UK), 1987.

[Henderson 86]
Henderson, P., Minkowitz, C.: *The* me too *method of software design.* ICL Tech. J., May 1986.

[Jones 86]
Jones, C. B.: *Systematic Software Development Using VDM*. Prentice Hall Intl (UK), 1986.

[Lins 86]
Lins, R. D.: *Categorical Multi-Combinators*. Computing Lab Report 41, University of Kent at Canterbury, 1986.

[Meira 85]
Meira, S.: *On The Efficiency of Applicative Algorithms*. PhD Thesis, Rutherford College, University of Kent at Canterbury, 1985.

[Meira 87a]
Meira, S.: *Gretas de Desenvolvimento de Software, Z e a Especificacao do Arco-Iris*. In portuguese. To appear.

[Meira 87b]
Meira, S.: *A Linguagem de Programacao Funcional*. In portuguese. To appear.

[Meira 87c]
Meira, S.: *From Z to A: Deriving Applicative Programs from Formal Specifications*. Outline of the SPI^{one} project. To appear.

[Meira 87d]
Meira, S.: *Hiperlexdo*. Projeto de Pesquisa Submetido ao CNPq. Personal Communication.

[Meira 87e]
Meira, S., Sampaio, A., Salazar, C., Sombra, M. and Souto Maior, R.: *Z: Notacao Computacional e Especificacoes*. In portuguese. To appear.

[Melo Neto 87]
Melo Neto, C.: *Orto, Um Revisor Ortografico*. Proc. Semish 1987, SBC, Salvador, 1987.

[Turner 86]
Turner, D. A.: *Miranda: a non strict functional language with polymorphic types*. Proc. Semish 1986, SBC, Recife, 1986.

[Wadler 86]
Wadler, P.: *An Introduction to ORWELL*. PRG Internal Report, Oxford University, (UK), 1986.

[Wirth 83]
Wirth, N.: *Programming in Modula-2*. Springer-Verlag, Heidelberg 1983.

[Zave 84]
Zave, P.: *The Operational versus the Conventional Approach to Software Development*. CACM, Feb. 1984