The Bla Language:
Extending Functional Programming with First Class Environments[1]

Wouter van Oortmerssen

May 1, 1996

[1] A big "thank you" to my supervisors Marcel Beemster, Remko Scha and Hugh Glaser for their patience with me and for steering me in the right direction, additional "thank you"'s for various reasons (also in no particular order) go to Pieter Hartel, Henkjan Honing, Vicky Syvess, Peter Henderson, Rob Verver and Jason Hulance.

Abstract

We investigate an (impure) functional language whose concept of environment is not implicit as in traditional languages, but made available explicitly as a first class value. This results in a semantics for environments that is best known from the Object Oriented paradigm, and gives us a united function / class concept in an orthogonal way. We also look at the language as a real-world, general purpose language, considering semantics (of type inference, for example), implementation issues, and practical experience of using the compiler.

Contents

1 An Introduction **3**

2 Design of the language **4**

2.1 Uniting Paradigms . . . 4

2.2 Environments vs. Objects . . . 4

2.3 Equational Style of Programming . . . 5

2.4 Syntax . . . 5

2.5 Type System . . . 5

2.6 Pattern Matching vs. Data-Hiding . . . 5

2.7 Extendibility . . . 6

2.8 Exception Handling . . . 6

3 Comparison to related languages **7**

3.1 First Class Environments in programming language design . . . 7

3.1.1 Simula . . . 7

3.1.2 Symmetric Lisp . . . 7

3.1.3 Beta . . . 8

3.1.4 Self . . . 9

3.1.5 Remotely related languages . . . 9

3.2 Type Inference and Type Systems . . . 9

3.2.1 Haskell . . . 9

3.2.2 Eiffel and Sather . . . 10

3.2.3 C++ . . . 10

4 Language report **11**

4.1 Source Syntax . . . 11

4.2 Functions . . . 11

4.2.1 Introduction . . . 11

4.2.2 Components of a Function Declaration . . . 11

4.2.3 Function Name and Modifiers . . . 11

4.2.4 Arguments (Defaults / Pattern Matching) . . . 12

4.2.5 The Function Right-Hand Side . . . 12

4.2.6 Genericity . . . 12
4.2.7 Inheritance and Subtyping . . . 12
4.2.8 Multiple Declarations . . . 13
4.2.9 Lambda's . . . 13
4.2.10 Object Interface: Data-Hiding . . . 13
4.3 Declarations and Types . . . 13
4.3.1 Scopes . . . 13
4.3.2 Built-In Types . . . 13
4.3.3 Tuples . . . 14
4.3.4 Objects . . . 14
4.3.5 Parametrised Types . . . 14
4.4 Expressions . . . 14
4.4.1 "where" and "do" suffixes . . . 14
4.4.2 Function Calls and Instance Variables . . . 15
4.4.3 Assignment . . . 15
4.4.4 Control Structures . . . 15

5 Example **17**

6 Implementation & Semantics **22**
6.1 The implementation . . . 22
6.2 Bla Type Inference . . . 22
6.2.1 Limitations on OO type inference . . . 23
6.2.2 Bounded unification . . . 23
6.2.3 Type parameter binding stacks . . . 23
6.3 The Bla virtual machine: "Emmer" . . . 24
6.3.1 File Format Layout . . . 24
6.3.2 Data Movement Instructions . . . 24
6.3.3 Integer operations . . . 24
6.3.4 Lists, Tuples and Objects . . . 25
6.3.5 Closures and Application . . . 25
6.3.6 Branching and Exceptions . . . 26
6.3.7 Stack Manipulation and Misc. . . . 26
6.3.8 Example Emmer Code . . . 26

7 Conclusions **28**

A Glossary **29**

B Bla Grammar **34**

Chapter 1

An Introduction

This chapter tries to give the reader a general feel of what Bla is about, starting with core concepts[1].

Almost all languages have separate abstraction mechanisms for code (functions) and data structures, even though both are often very much alike. Also, functional languages in particular have missed out on powerful ways of creating data structures in the *Object Oriented* sense, as the tools were available (function *environments*) but not fully reachable. Bla addresses these issues.

The core of Bla is basically a *functional language*. A key feature of most functional languages is the concept of a *higher-order function*: the ability of functions to handle other functions as values, i.e. get them as arguments, return them, and generally pass them around. In the full generality of a *closure*, function values are passed around together with the environment of the enclosing function, since *free variables* may still refer to it. Functional programmers have exploited this feature by using the environment of enclosing functions as "objects".

A slight problem with using these environments as objects is that only the inner function has access to it: the object is anonymously captured "in" the function. Implementing the message-send model of OO languages this way is not possible (or an extremely clumsy emulation at best). The change that Bla makes to this model is very simple, but has great consequences: *it makes the environment object explicitly available as a value.*

In Bla, every function has available the value `self` which denotes that function's environment object. One of the simplest things that can be done with this value is return it: the caller of that function then receives an object as the result. The definition of a function `f` introduces not only a new function, but also a type `f`, being the type of its environment object. A call to `f()` creates an object of type `f` and executes the code of the function within that environment.

Functions local to other functions are actually functions that are required to be evaluated within the environment of a certain function. Since this environment is now accessible as an object, local functions can be invoked wherever such an object is around as a value: they have the same role and semantics as methods in OO languages. Other Object Oriented features come (almost) for free as well.

Using this concept of "1st class environments", Bla integrates functions and Object Orientation in a tiny language, in an elegant and natural way. However, on top of this core language, Bla adds quite a few concepts to make programming practical.

The type system used by Bla is a variation on the *Hindley-Milner* system, and provides very practical *parametric polymorphism*, extensions mostly to support OO constructs, providing *inclusion polymorphism*. The compiler uses a powerful *type inferencer* and Bla is *compile-time type-safe*.

Bla's syntax is equipped with very comfortable *pattern-matching* facilities, and is targeted towards a dominantly functional style of programming.

Bla breaks with *referential transparency* and allows use of assignment as well, which is essential for the OO concept of an "object" to work.

[1] Familiarity with functional / Object Oriented languages and associated concepts is assumed. There is a Glossary in the appendix which tries to explain some concepts, often the ones printed in *italics*. For a more in-depth introduction to functional programming I can advise [BW88] or [Hen80]; Object Oriented concepts are explained well in [Hat93], or else [Mey92] will do too.

Chapter 2

Design of the language

There were quite a few reasons for designing Bla as it is now, and some of them are investigated in this document[1].

2.1 Uniting Paradigms

Bla unites the Functional and Object Oriented programming paradigms. It was never really a goal as such to unite the two, rather it came from the realisation that both could be united very well, and most of the features explained here contribute to that.

Needless to say, combining two paradigms is a very obvious thing to do (looking at the history of programming language design), and often results in huge bulky languages that have "forced" semantics. The design strategy taken in Bla was actually the reverse: once a semantic construct was invented (the Bla "1st class environment") it became clear that it covered almost all abstraction mechanisms present in both paradigms. As a result, the semantic core of Bla is much smaller than any "single-paradigm" language it might resemble.

2.2 Environments vs. Objects

The semantic feature that constitutes the core of the language, and that provides the starting point for many other features, is the Bla function. It unites functions (or: methods / lambdas / procedures) and classes (or: structures / terms). This comes from the realisation that a function's environment[2] is very similar to an object. At the same time functions may be seen as *constructors* for environments.

For example, in C++ [Str91][3]:

```
class X {
  int a;
  int b() { return a+1; };
  X() { a=1; };
};

void Y {
  int a=1;
  int b() { return a+1; };
};
```

Both declarations `X` and `Y` introduce a *name space* with an integer `a` and a function `b`, which may use `a`. For both, an environment is created upon evaluating the expressions `new X()` and `Y()` respectively, and code executed (`a` is set to `1`, etc.). The difference is that for the former the environment object is returned as value (it may be allocated anywhere, in this case dynamically), for the latter it is automatically destroyed upon termination of the function (it is allocated stack-only). In Bla we could write[4]:

```
X() = self where
  a = 1
  b() = a+1
```

This too gives us an environment with `a` and `b`. The programmer chooses to hand it over to the caller by returning `self`. To get the behavioural equivalent of `Y`, we could return some other value instead of `self`, or leave it out altogether.

What we end up with in Bla is that a function may or may not decide to do something with its environment as a *first class value* (by accessing the `self` value, the function may even return it for use elsewhere after the function has terminated). From this view of environments as first-class values, closures suddenly become the most natural thing in the world[5], and locally defined functions (possible in most modern functional languages) suddenly become *methods* when the environment is approached as an object "from the outside". And this collection of possibilities comes from just one very simple generalisation of the classical function: making the environment a first-class value[6].

[1] No time will be spent on explaining how the usual design goals of *orthogonality*, expressiveness, simplicity etc. were reached, rather we look at goals specific to Bla.

[2] Speaking from an implementation perspective: activation-record (AR).

[3] Functions local to other functions are allowed only in the GNU implementation, as far as I know.

[4] For a more complete description of Bla syntax, read chapter 4.

[5] They are tuples consisting of an environment and a function to be evaluated within that environment, after all.

[6] Much like the generalisation of the classical function call mechanism into a *continuation* in Scheme [WC91] was very simple, but with far reaching consequences.

Functional languages have utilised environments for a long time (returning function values which use variables from enclosing scopes [WC91] [HPW92]), but haven't used this to their full potential: environments are available only *implicitly*. Recent languages such as Self [Ung91] and Beta [KMMN87] have unified objects and functions in some way, though their semantic details differ from Bla (see chapter 3).

2.3 Equational Style of Programming

Not entirely orthogonal to the abstraction mechanism in a language (as we'll see below) is the style of writing code. Though tastes may differ, it was concluded that a functional / equational style of writing code is far superior to the alternative of writing imperative code.

Especially in Object Oriented languages, of which almost all have adopted the imperative style of writing code, this becomes painfully obvious. As most code these days is written in a very modular style, the bulk of an object's methods often do very little actual computation. Still, since a flexible object may be in quite a number of states, the net result is that you often have nested if-thens selecting code that doesn't do very much. The *interface* to the object may be very clear, but the code that implements it isn't.

Bla supports quite a few mechanisms familiar from functional languages to support a functional style of programming. Care was taken that this is also possible for impure code[7].

2.4 Syntax

Simplicity is of course an important design goal in a language, but it shouldn't be taken to extremes. In Bla simplicity is mainly targeted towards semantics: a lot of Bla syntax is "redundant". There are very good reasons for this, though. Making semantics simpler by uniting concepts often results in more obscure syntax for frequently used code patterns. In Bla, quite a few syntactic constructs have been added that might be considered superfluous from a purist's point of view, but serve as essential support for an equational style of writing code: this is pure *syntactic sugaring*, as all of it can be transformed into more elementary syntax. Instead of complicating the syntax, it makes Bla programs very concise and easier to read.

In Bla, especially for basic, not too complicated declarations, a syntax based on operator-like symbols rather than a heavily keyworded one is used. Also, for certain constructors (such as for lists) a special syntax has been used. Instead of making syntax fuzzier, they differentiate the way code looks, which means a look at a piece of code will give you directly an impression of which constructs are contributing to the whole. Though all this syntactic richness makes the language larger, it should be considered "A Good Thing".

[7] Here, pattern matching is slightly less obvious, as values to be selected upon may come from other scopes, not just arguments. Bla does not have global variables, though.

2.5 Type System

> **Polymorphism**: The occurrence together in the same locality of two or more discontinuous forms of a species in such proportions that the rarest of them cannot be maintained merely by recurrent mutation. Polymorphism necessarily occurs during the transient course of an evolutionary change. Polymorphisms may also be maintained in stable balance by various special kinds of natural selection. — Richard Dawkins, *The Extended Phenotype*, 1982.

Bla's type system is *two-level* [Hat93] (classes and objects), *contravariant*, allowing for *multiple inheritance / subtyping (by name)* and *genericity*, *single-dispatch* polymorphism (quite similar to Sather [SO94]). It uses the *open-world* assumption.

It has a type hierarchy which starts with the type "any", and incorporates both a number of built-in types and a class hierarchy. The choice was made not to have built-ins like integers and list as classes. As all of them have special syntax, and most of them need special semantics, it was considered artificial to force them into being classes when they're not.

A choice for the "variance" of a language is hard, since none of co- / contra- / no- / any-variance is a universally good choice (Shang [Sha94] [Sha95] gives good examples). Contravariance was chosen simply because it's the most logical choice: it simply says that *subclasses* should conform to their *superclasses* no matter what happens, and thus is compile-time type-safe. Some have argued that *covariance* is more like how we would like to model the real world, but it is not type-safe and makes classes into "almost subclasses" which is not the nicest of concepts in a type hierarchy[8].

2.6 Pattern Matching vs. Data-Hiding

Various people (for example John Sargeant [Sar94]) have argued that *pattern matching* and *data-hiding* are two incompatible things, as the former makes use of the representations of types whereas the latter tries to hide it. We'll see how the two cooperate nicely in Bla, without sacrifices, further below.

There are two problems in the data-hiding models of most of today's Object Oriented languages:

[8] When crafting algorithms we are hardly modelling the real world all the time. Those few cases where relationships between classes actually model a situation where you'd want to use covariance, Bla's sophisticated version of parametric polymorphism allows you to 'emulate' it in a type-safe fashion.

- *They're too strict*. It's hard to write a set of tightly cooperating classes without endangering data-hiding. Various kludges have been devised to solve this (such as "friend" methods in C++ [Str91]), but none really satisfactorily.

- *They're not strict enough*. Data-hiding looses its function in languages like Eiffel [Mey92] that do not provide for data-hiding towards subclasses, and where the programming style essentially blurs the distinction between *client* and subclass.

Both problems are related and mainly stem from the level of data-hiding that was chosen: dividing an application with data-hiding boundaries is orthogonal to language concepts like "class". Bla's modules may contain one or more functions / classes, and form an impenetratable *black box* for both clients and subclasses outside the module, thus solving both problems[9].

What this boils down to is that *reuse* is everybody's business, but data-hiding is only the implementor's business.

Pattern-matching fits into this scheme very nicely. As pattern-matching is often used for implementing algorithms, as opposed to using them, code may arbitrarily exploit its knowledge of data structures within a module. At the same time, for all data structures outside the module, it may only match on the interfaces provided. This doesn't damage data-hiding in any way (in fact, it has the same visibility as for other constructs), but at the same time the opportunities for pattern-matching are excellent.

2.7 Extendibility

Once you've fixed an interface for a class, you can change the implementation "behind the scenes" in any way possible[10]. However, every now and then you might want to add something to an interface without disturbing the rest. As far as adding methods to an object is concerned, this is catered for quite well in OO languages. There is, however, one component that is quite often neglected in this respect (apart from languages such as Smalltalk [GR83]), and that is the arguments to a function. In Bla it was decided to have *tagged arguments*, which allow the caller to specify arguments in any order, and only the ones that have no defaults. This allows the implementor to extend the set of arguments at any time, provided he can think of a default value for them, so existing code does not need to be modified. Needless to say it also enhances the readability of a call enormously. For well-known functions that have few arguments however, tags would only be unnecessary noise, so a caller may decide instead to leave out tags and specify the arguments in order.

[9] This scheme has been copied from E [Oor93] where it was used in practice extensively and found to be a solid basis for creating software components. One other language with something similar to this is Ada 9x [WC85].

[10] At least, that's the OO promise.

2.8 Exception Handling

Exception Handling was considered an essential feature. This has always been a point of great controversy in the programming language community, but I will not go into it too deeply here.

The essential issue is: if deep in a *call graph* the program encounters an unexpected situation (or error, more specifically), what should it do? Classic programming techniques basically give you two options:

- Exit the function with a special error code as the return value. Apart from the question as to whether the return value is available for that purpose or whether it is semantically "right" to use it in that fashion, the biggest problem with this is that such an error must percolate up to `main()`, i.e. it demands that all calls in between also have error-code return values, and worse, error handling code. This is highly error-prone.

- Exit the whole application there and then (such as with `exit()` in C), for serious errors. For small programs this might work, but for multi-window, multi-project complex applications one would hardly want the whole application to shut down if somewhere locally a resource allocation failed. Doing this right as far resource-deallocation goes is even more problematic.

Both of these are unacceptable for serious programming. Exception handling allows code to recover from errors at various (dynamic) levels in an application without having error return codes and error handling code but in the handlers themselves. Exceptions neatly separate algorithm code and the large amounts of error-checking if-thens real-world applications are infested with.

A more fundamental issue is that without exceptions, one cannot write *truly modular code*. If one writes a software library that communicates with the application only via a well-specified interface, deep down in the library there will be code that knows about errors but doesn't know how to act on them, in the application there will be code that can decide what to do with errors but doesn't know about them. Making this work without exceptions means giving up carefully wrapped interfaces for abstract datatypes and breaking with modern software engineering techniques as advocated by OOP.

Most of the protests against exceptions stem from looking at them from the wrong perspective: they are (falsely) associated with gotos[11]. As opposed to the unconstrained goto, exceptions are a well-defined mechanism. If ever you wanted to associate them, "bottom" as available in many semantic-definition languages would be a more worthy comparison.

[11] The word alone sends shivers down the spine of many programming language researchers :-)

Chapter 3

Comparison to related languages

In the last chapter we saw general reasons why Bla was designed as it is, in this chapter we will look at important cornerstones of Bla in a broader context and compare with related languages.

3.1 First Class Environments in programming language design

While it may seem obvious and natural to use functions as the sole abstraction mechanism, almost all languages in the history of programming languages have chosen to equip data structure & code abstraction with unrelated and unorthogonal syntax and semantics. Especially in newer Object Oriented languages where the programmer deals with many name-spaces (method, parent-method, class, super-class) the fact that these are conceptually very much alike becomes painfully obvious. Few languages however, have something related to the concept of First Class Environments.

3.1.1 Simula

Simula [BDMN] was the first language to introduce the "class" in a language. Back then, when Fortran, Algol & Lisp were people's reference material, the designers of Simula used to refer to the similarity with Algol "functions" to be able to explain what the concept of a "class" was all about[1]. Besides, in Simula, classes and functions differ only in the fact that one starts with `CLASS` and the other with `PROCEDURE`: classes too have arguments, and have a body with code, performing initialisation. In one of the original books on Simula, the explanation of classes starts by telling how functions have environment objects created for the lifetime of the function, then relating this to the concept of "class" and "object". Simula was off to a good start, later languages inspired on Simula have often steered away from this uniformity, however.

[1] Personal communication with Peter Henderson.

3.1.2 Symmetric Lisp

This is the only language I have found that comes from the same design philosophy, with the same objective as Bla. Symmetric Lisp, a language designed by David Gelernter[2] and others, described in the paper "Environments as First Class Objects" [GJL87b] and in [GJL87a]. They have created a Lisp based language that uses a construct called an "Alpha Form" as its sole abstraction mechanism. Alpha Forms are name spaces and may have associated code, and when evaluated create an object and execute the code in that context. An alpha with three elements, adding the first two when evaluated, and resulting in the data structure as a whole after evaluation:

```
(ALPHA
  (NAME x 1)
  (NAME y 2)
  (+ x y))
```

Alphas may consist of any number of elements (which can be either expressions or `NAME`s which introduce identifiers) and whose value is that of the expression provided as the initialisation.

The original Lisp `LAMBDA` in Symmetric Lisp is now a bit of syntactic sugaring, picking the last element out of the alpha as the return value after evaluation. `PLAMBDA`, on the other hand, returns the whole alpha, as a data structure[3].

The exact semantics is quite a bit different from Bla though, as in Symmetric Lisp environments may be extended at runtime (this is used as a basis for concurrent computation), but we will not go into that aspect of Symmetric Lisp here.

The difference with Bla lies in the approach to environments: whereas in Bla the basis is the classical function from which the already existing environment is utilised in a novel way, the Alpha Form is an environment as a new semantic construct, not necessarily connected to a function. As such their usage of "1st class environments" should be read somewhat differently from Bla (they start out with environments), also their basic semantic construct is more low-level than in Bla,

[2] Better known for his coordination language called "Linda" [CGL86].

[3] The paper mentioned goes on to show how every abstraction mechanism / data structure under the sun can be replaced by Alphas.

i.e. Alpha Forms are more like building blocks, the machine code of function and data structure abstractions (we will see this also in "Self", later on).

From a lambda in Symmetric Lisp an Alpha Form is constructed upon application, For example:

```
((LAMBDA (x y) (+ x y)) 2 3)
```

results in:

```
(ALAST
  (ALPHA
    (ARGNAME x 2)
    (ARGNAME y 3)
    (+ x y)))
```

where `ALAST` picks the last element of an Alpha, and `ARGNAME` is equivalent to `NAME` except that resolving names within the expression starts with the enclosing Alpha, not the current one. The catch is that the relation between the lambda itself and the alpha is kept implicit (the semantics is based on the application).

As another example the function `X` which we saw in Bla (and C++) in chapter 2.2:

```
(NAME X
  (LAMBDA ()
    (ALPHA
      (NAME a 1)
      (NAME b (LAMBDA () (+ a 1))))))
```

Notice the fact that the outer lambda wraps an extra dummy Alpha around the whole. The reason for structuring code like this has to do with the above mentioned lambda semantics and the fact that Symmetric Lisp doesn't have something equivalent to `self` in Bla[4].

3.1.3 Beta

Beta [KMMN87] [Man93] is a modern OO language designed by some of the people involved in Simula. In Beta, classes and functions are represented by the single abstraction mechanism, the *pattern*. Orthogonality in Beta comes at a high cost though: patterns themselves say nothing about the exact usage of the environment, and creation of pattern objects and execution of related code do not necessarily go together. What this boils down to is that you can have function environments / objects available in your program as entities, with the corresponding code / initialisation to be executed separately, if ever (in Bla, function application and environment creation are inseparable). Beta also makes a big deal out of static (in-lined in the parent object – no equivalent in Bla) vs dynamic object *instantiation*: environments for functions can be instantiated statically, except when the code uses for example recursive calls, then the user of the pattern has to instantiate it dynamically. Beta uses a whole array of operators to differentiate between all these different modes of declaration, instantiation and execution[5]. As an example, a Beta pattern `f` that takes two values and outputs their sum:

```
f:
(# x,y,o: @Integer
   enter(x,y)
   do x+y -> o
   exit o
#)
```

The `(#` and `#)` enclose the pattern. First the identifiers used in the pattern are declared. `enter` shows the variables that are used as inputs to the code, `do` contains one or more statements, and finally `exit` shows the output variables. This pattern can be instantiated, i.e. an object can be created that holds the three variables: this would make an environment object for the code, which can then be executed within it. Using this pattern as a function can be done in two ways, statically:

```
(1,2) -> f -> result
```

or dynamically:

```
(1,2) -> &f -> result
```

The `->` operator is a combination of application and assignment. What it does here is that the expressions `1` and `2` are evaluated, and assigned to the input variables `x` and `y` of the instantiated pattern `f`. After the code in `f` has been executed, the second `->` reads the output value from the `o` variable of `f` and puts it in `result`.

The difference in instantiation between the two is that in the former case the memory for the environment of the function is allocated as part of the surrounding object. The latter is needed if the implementation of our function were to use, say, recursive calls, since in the static case the same piece of memory would be used for each invocation.

Using this pattern as an object can also be done statically and dynamically:

```
my_f: @f
```

declares the new variable `my_f` which is a static `f` environment, i.e. memory for `f` is preallocated. To declare just a reference (a pointer):

[4] I suspect there is a good reason for this choice, but I couldn't quite figure out exactly what it was.

[5] I was flabbergasted by this design "move" by otherwise very knowledgeable people. The syntax is another area that needs special attention: a typical case of oversimplification.

```
my_f: ^f
```

The memory here needs to be allocated separately and assigned to `my_f`, as follows ([KMMN87] uses a neat little box in the syntax, which I have replaced here with []):

```
&f[] -> my_f[]
```

`&f` instantiates `f` dynamically as seen above. Normally mentioning `f` in such a statement would mean that the code should be executed, the [] box prohibits this and just passes the pointer on. Note that these two `my_f` objects both have their memory allocated now, but not their associated code run! We can accomplish that as follows:

```
(1,2) -> my_f -> result
```

From a software engineering perspective it seems it is the compiler and the pattern designer that need to be deciding about object in-lining and initialisation respectively, not the pattern user.

3.1.4 Self

Self is a prototyping delegation OO language with roots in Smalltalk (A good introduction is [Ung91], an in depth treatment can be found in [Gro92], additional worthwhile reading material is [CUL89], [CUCH91b] and [CUCH91a]. A recent paper with some "re-evaluations" is [SU]. Read about the general philosophy behind prototyping in [Lie86], [LSU87] or [DMC92]). Self is a *prototyping* OO language meaning that objects are not instantiated from classes but are *cloned* from existing objects. Sharing of behaviour is through *delegation*, e.g. one object treats another as parent, and when messages cannot be dealt with they are passed on to parents.

Self has a uniform object syntax that is used to define both objects and functions (methods), however Self does not have a truly 1st class environment concept as Bla has. It is at the lower level (the object "machine code"[6]) that similar representations are used, yet the semantics of message sends (function application, if you will) makes the environment into a specific sort of object not to be confused with "normal" objects. For example:

```
o = (|x. y.|)
```

() delimits an object (not a declaration!), || delimits the elements of an object (called *slots*) separated by dots, and anything that might follow is code. o is now bound to a ready made object with two elements, `x` and `y`, that can be used directly or cloned.

[6]Self's semantic simplicity causes the programmer to have to do many housekeeping jobs himself. For example, to implement shared behaviour he will construct special "traits" objects and install the right parent pointers to and from the traits object etc. The apparent simplicity of having only a few basic mechanisms makes you re-implement commonly used higher-level mechanisms over and over. This is, in a less severe form, comparable to Symmetric Lisp (see chapter 3.1.2).

```
f = (|:x. :y.| x+y)
```

Here : shows that the slot in question is a formal parameter. `x+y` is the body of the function. `f` is a method taking 2 arguments and returning their sum. Upon function application, the object `f` is bound to is cloned and serves as an activation record / environment. The receiver of the message is installed as parent of the environment, but the semantics of method lookup suddenly have a special case for environments as opposed to "normal" objects: search starts at the environment object, but the default *receiver* is the parent, not the object itself. This makes perfect sense, but shows that the language treats function environments and objects differently, even though they are implemented using the same underlying mechanisms. One can use `self` as a return value and make sure the receiver is the envisioned parent, yet one cannot create an object with the same behaviour as a "normal" object this way.

3.1.5 Remotely related languages

Worth mentioning here are various term / tree rewriting / equational logic languages (read [O'D85] for a good overview). Though semantically very different, these languages also make absolutely no distinction between functions and objects: the only language element is a *term* from which treelike[7] structures can be built. A set of rules then defines how certain tree patterns are to be transformed. Execution is complete when the tree structure has reached *normal form*. Terms having the role of data structure can be seen as transforming to themselves, which is related to Bla functions returning `self`[8].

Other papers somewhat related to First Class Environments are [Dun85] and [Dea], which will not be discussed here.

3.2 Type Inference and Type Systems

Type inference has been a popular choice for functional languages lately, while it has been almost absent in OO and imperative languages. We'll look at some type systems and their relation to Bla (see chapter 6.2 for a more technical view of Bla Type Inference).

3.2.1 Haskell

ML [MTH90] is the prototypical language with a Hindley-Milner type system (sharing many properties with Bla), the state of the art language however is Haskell [HPW92]. In some respects Haskell goes way beyond what Bla and similar

[7]Or even DAGs, sometimes.

[8]This is slightly far-fetched, but I like it.

functional languages have to offer: *Type Classes* for example. Basically, Type Classes form a structured alternative to a possibly dynamic form of *overloading*. They allow the programmer to group sets of operations in a *class*, and any type can become an instance of this class by implementing the operations for that type: the Haskell standard prelude does this successfully for datatypes on which equality is defined (the `Eq` class, which makes writing functions like `member` a lot more general) and various levels of numeric datatypes. Read [PJ93] for a good understanding of what type classes are all about.

The word "class" might suggest there is a correspondence to classes as found in Bla, but this is only very unstraightforwardly so. The essence of OO polymorphism is that apart from being able to define new datatypes that cooperate with existing code (like parametric polymorphism), those datatypes can have redefined operations that are used transparently and dynamically by existing code, which provides an extra dimension in code reuse. This is something Haskell does offer too, only from a different perspective: whereas OO views things from the datatype (having a set of functions working on it), Haskell views it from the functions (having a set of datatypes on which they work). Classes in Haskell are *state*-less, i.e. they are separated from datatypes, and datatypes cannot have their own inheritance hierarchy, which is common in languages that split type and class hierarchies. Many other OO related issues become twisted this way, and thus calling Haskell an OO language as suggested [Ber92] seems a bit far-fetched. As an example, typing `map op [A,B]` where `A` and `B` are expressions constructing values whose types are instances of a class defining `op` will result in a type clash rather than a *dynamic dispatch*.

3.2.2 Eiffel and Sather

As far as type systems for OO languages go, Bla is closest to languages like Eiffel and Sather, though these do not offer type inference. All three are statically type safe and offer both inclusion and parametric polymorphism. Differences lie in small details of *type parameter* semantics (see also chapter 6.2.2), and the fact that Eiffel is covariant whereas Bla and Sather are contravariant. Also, Sather has separate type and class hierarchies.

3.2.3 C++

The type system of C++ [Str91] is quite a bit different from Bla, which stems partly from the fact that C++ works by default with value semantics, and thus has different sized types: built-in types like `char` and `double` are different from `int`, and `class rectangle` or `char s[100]` have no reference semantics.

This becomes most obvious with the C++ equivalent to type parameters, *templates*, which are more like macros. In Bla type parameters are only necessary to keep the type system happy (they allow container-like code to be type safe), in C++ they duplicate the code for each type with a different value size or different overloaded operations[9]. In C++, it is not possible to narrow a type parameter to a specific type and its subclasses, though code will sometimes implicitly assume this will be the case: type errors will result only when the template is actually used (like macros). Furthermore, templates need to be specified explicitly everywhere, which makes their usage especially awkward: in Bla they can almost always be inferred, both in definition and usage.

The inheritance mechanisms of C++ and Bla are much alike, though C++ lacks subtyping as a mechanism. Instead of contravariance C++ has *no-variance*.

[9] I have argued that *overloading* is a bad thing altogether. The fact that it's only syntactic sugaring brings a whole range of problems to a language, and this is one of them. The advocated programming style for C++ focuses on the use of overloading where *virtual methods* should have been used instead, and gives code explosions for things like sorting functions that make use of overloaded equality operators.

Chapter 4

Language report

This chapter presents a (slightly informal) overview of all the language features.

4.1 Source Syntax

A Bla source is an ascii text file with a filename ending in `.bla`. White-space includes line-feeds, i.e. it's a free-format language. Programmers may use the well known "Layout Rule" (see Haskell [HPW92] for example) for `where` and `do` blocks. Comments start with `/*` and end with `*/` and may be nested, single line comments are introduced by a `--` and continue until the first line-feed.

Bla syntax will be introduced informally in the form of typical examples. For a more precise definition please refer to appendix B where you'll find the Bla grammar.

4.2 Functions

4.2.1 Introduction

Functions constitute the sole abstraction mechanism in Bla; they join the expressive power of both traditional functions (methods / lambdas) and objects (as found in Object Oriented languages).

4.2.2 Components of a Function Declaration

A typical function declaration looks like this:

```
functionname(args,...) = exp
```

Two typical examples:

```
append([],y)    = y
append([h|t],y) = [h|append(t,y)]

stack[T]() = self where
  d = []
  isempty() = d=[]
  push(x:T) do d:=[x|d]
  pop():T = d | []    -> raise stack_empty
              | [h|t] -> h do d:=t
```

Function declarations make available both a function value ready for use in function applications, and a type. The type is for the environment objects that are created when the function is invoked. These environments are accessible through the variable `self`, as demonstrated by the `stack` example.

The top level of a file consists solely of function declarations, a `where` may introduce functions and variables local to an expression. A precise explanation of each element of these declarations will follow.

4.2.3 Function Name and Modifiers

Usually a declaration starts with the name of the function (or variable) to be defined. The name may take extra modifiers:

Data-hiding modifiers: these control visibility of names from outside the module in which they are defined (see chapter 5 on data-hiding):

- `public`: visible from everywhere (default for function declarations).
- `private`: visible only within this module (default for local variables).

Additionally, for declarations local to functions, there are these "storage" modifiers:

- `object`: the item is not shared: each object gets a local copy (default for local variables).
- `class`: the item is shared among all objects of a class (default for functions). Functions defined like this will behave polymorphically towards subclass equivalents.
- `const`: the item is shared, only now the item isn't modifyable, and in the case of functions: not redefinable.

4.2.4 Arguments (Defaults / Pattern Matching)

Arguments to the function are grouped by compulsory parentheses (), and separated by commas. Arguments take the form of patterns, followed by a default:

```
kartoffel(a:int,b=[])
```

here `a:int` and `b` are simple variable patterns, both the type and default are optional. Types may be inferred, and calls may use tagged arguments to benefit from defaults and order-independence (see below).

Other patterns are the equivalents of constructors of, for example, lists, vectors, strings, ints, etc.

```
schnitzel(sauerkraut([1|t]),´xyz´)
```

The function `schnitzel` takes objects of type `sauerkraut` as the first argument, and a string as the second. `sauerkraut` has a single feature (matching is done on the interface of the object) that is a list, of which the head must be equal to the integer 1 and the tail is bound to `t`.

Because these matches might fail, function declarations are allowed to be repeated with different patterns, as shown in the `append()` example, and as explained further below.

Additional patterns are the "don't care" pattern `_` which does the same as a variable, but doesn't introduce a new name, and the pattern sequencer '/' (equivalent to `@` in many languages), which allows you to match two patterns onto the same value:

```
apfelstrudel([h|t]/[(a,b)|_])
```

Matches two patterns on the first argument, the first takes out the head `h` and tail `t`, the second matches the head on a tuple and ignores the tail.

4.2.5 The Function Right-Hand Side

An argument declaration may be followed by an "=" and an expression, which will form the return value, or, if the function is not to return a value, just a do block. For example:

```
len([])    = 0
len([_|t]) = len(t)+1

main() do
  puts(´Hello, World!\n´,stdout())
```

4.2.6 Genericity

Generic functions and objects can be constructed by attaching a generic type parameter to the function declaration, which must be a type. Syntax:

```
f[T](...) = ...
```

The identifier `T` can now be used as a type anywhere in `f`. More than one generic type parameter is allowed, and may be restricted to subclasses of a certain type:

```
[X,Y < burp,Z]
```

Here, `Y` needs to be a subclass of `burp`, which includes of course `burp` itself, and `X / Z` may be anything (subclasses are explained below).

Note that not all functions will need explicit parametric type declarations: for example the function `append()` above is automatically type-inferred to work on list of anything, and applications of the function do not need explicit types either. The stack example can also be written without any type declarations, but is explicitly parametrised for clarity.

4.2.7 Inheritance and Subtyping

In Bla both (multiple) inheritance and subtyping can be used, using the keywords `extends` and `implements` respectively. Both specify that a subclass conforms to the superclass with respect to the interface, with the difference that inheritance additionally provides the subclass with the implementation of the superclass's interface, which can be overruled.

Examples:

```
fast_stack[T]() implements stack[T]
  -- implement a stack using a
  -- different data structure but
  -- conform to stack´s interface


nicer_stack[T]() extends stack[T] where
  -- reuse the whole stack, while adding
  -- a method that swaps the two top
  -- elements, and one that drops an element:
  swap() do d <=> [a,b|t]
            d:=[b,a|t]
  drop() do d | []    -> raise stack_empty
              | [_|t] -> d:=t
```

From the point of view of a function, subclasses are those functions whose environments are an extension of each other, so it is generally most useful when you actually do something with the environment (object).

4.2.8 Multiple Declarations

As seen in quite a few examples above, a function may be defined using multiple declarations. Note however, that this is just syntactic sugaring, and that the compiler internally transforms this to more conventional constructs. Other object components of multiple declarations, such as return-value expressions, may need to conform to each other with respect to types etc. Also, pattern matching has an effect on the interface of a function / object, as, by definition, it splits *scopes*.

4.2.9 Lambda's

A lambda function differs only from a normal object declaration in that it has the keyword "lambda" instead of a name. Lambda's are useful to create all sorts of function values at run time.

```
map(lambda(x) = x*x,[1,2,3,4,5])
```

Lambda's in Bla are closures (i.e. they can be used in higher-order functions):

```
acc(n) = lambda(a) do n:=n+a

a = acc(5)

a(10) -- 15
a(10) -- 25
```

Here, acc is a function that generates accumulator functions at run time, much resembling an object (one might want to bracket () lambda's in some cases, but it's not needed in the example). Another classical example:

```
dot(f,g) = lambda(x) = f(g(x))
```

4.2.10 Object Interface: Data-Hiding

Data-hiding in Bla goes with the philosophy that data-hiding actually means chopping a large application into pieces (placing fences between parts), and that communication between these parts only occurs via specified interfaces. Within one part however, no data is hidden. Traditionally these 'borders' lay between classes but in Bla they lay between modules, where each module may contain one class (which equals the traditional case) or more, if necessary. A point where Bla is more restrictive is that it maintains the same data-hiding for both client and subclasses of a class. If the implementor wishes, his class can become a true black box.

Each function declaration results in an interface automatically. For example, the interface of our stack is:

```
stack[T]():stack
  pop():T
  push(x:T)
  isempty():bool
```

For clients and subclasses outside the module in which this stack is defined, this is all the information they'll get. For example the fact that the stack uses lists for implementation is completely hidden from them. An implementor can easily choose to change what is visible in the interface by appropriately applying `public` and `private` as explained above.

4.3 Declarations and Types

4.3.1 Scopes

Bla has arbitrarily nested *lexical scoping*, familiar from languages like Pascal etc. The top-level scope is a global one, which only allows function definitions. Each function definition is a (function) scope on it's own, and consists of the scope of the primary expression of that declaration, as well as argument declarations. Expressions may contain any level of scopes (using `where` and `do`).

Function and variable definitions need not be in a `where` part, they may occur anywhere within an expression. Declarations in expressions are preceded by `` ` ``, to distinguish them from the use of those entities.

4.3.2 Built-In Types

Bla has quite a few built-in types, most of them with special syntax for constructors, types and pattern matching. All entities in Bla have *reference semantics*, with the exception of `int` / `bool` / `real` which have *value semantics*.

- `int` is the built-in integer type (of *at least* 32 bits in size). Constructors are integer values such as `1` and `-128`, and character constants `"A"` and `"\n"`. Longer character constants (up to 4) may be used, such as `"FORM"`.

- `bool` is the built-in boolean value, and can be either `true` or `false`. Exchange with integers (in the form of `0` or `not 0`) is allowed. A pointer may also be seen as a boolean (`nil` or `not nil`) i.e. `false` = `nil` = `[]` = `<>` = `0` = `"\0"` holds.

- `real` is the built-in float type with same size as int. Constructors always contain at least a `.`, i.e. `3.14159`, `.1` and `4.`. Automatic coercion to and from ints is provided.

- `any` is the supertype of *all* types. The only operation defined on it is assignment, and equivalents to that such as parameter passing.

- `vector` is a linear collection of any other type. It differs from an array in the fact that it has a current length, smaller or equal to the maximum length that was allocated for it, i.e. it's a bit more like a string or a linear list. A constructor for a vector of reals of length 3 could look like: `<1.0,3.6,.7>`, and its type would be `<real>`. Vectors have their own selection operator which may access elements from 0 to current_length-1. If `v` is the example vector above, `v[1]` gets us the value `3.6`.

- `string` is a built-in string type. It has no fixed length and may be extended as needed.

- `list` is a regular *cons cell*. The type `[bool]` for example tells us that every head is a `bool` and every tail again a `[bool]` (or `nil`, the special value of any pointer type). A constructor has the form `[head|tail]`, or for example the more convenient `[1,2,3]` which equals `[1|[2|[3|nil]]]` (combinations like `[1,2|3]` are of course also legal). Lists in Bla are *non-mutable data structures*.

4.3.3 Tuples

Tuples are on the fly *heterogeneous data structures* made out of 2 or more values. For example `([1,2],true)` is a tuple of type `([int],bool)`. Tuples can be handy for returning multiple return values, performing pattern matches on any number of values at once and various other tasks. They are non-mutable as well.

4.3.4 Objects

We can create objects by invoking functions and accessing their environments. Remember that a function declaration introduces two types, one as a class, and one as a function value. For example, our stack introduces a value `stack` of type `()->stack` which can be used in expressions, and a type `stack`, which can be used to type a variable (leaving parametrisation for the type inferencer, for now):

```
s:stack
```

Note that this has reference semantics, i.e. no memory for the stack has been allocated at this point. In the case of for example `append` there will never be any value which can be assigned to a variable `a:append`.

`stack` as a function value can be used as follows to construct new stacks:

```
stack()
```

This is an expression returning an environment object of type `stack`.

4.3.5 Parametrised Types

How to produce a parametrised type was already introduced in our stack example above. A type for a specific stack is simply the stack type with the parameter filled in:

```
kartoffel:stack[int]
```

makes `kartoffel` a pointer to a stack of `int`s (uninitialised). To get an initialised stack:

```
kartoffel = stack()[int]
```

The type parameter may of course be left out.

4.4 Expressions

4.4.1 "where" and "do" suffixes

Any expression may be suffixed with a `where`, which allows for definitions local to the expression. The syntax is:

```
exp where { def ; ... }
```

The "{" and ";" are subject to the layout rule. An example is:

```
sqr(a) + sqr(b) where
  sqr(x) = x*x
  b:int = 1
```

In this case, the expression and the where part form a scope level together, i.e. `sqr()` is not available outside that expression. Definitions in a `where` may depend on earlier definitions only. If the `where` contains expression as initialisations of variable declarations, these are evaluated in order too. `where` has a very low operator priority.

As Bla is not a "pure" language, there is a need to evaluate *side-effecting* expressions as well. These can be put in a `do` part (with the same "{;}" syntax as `where`), suffixed to the expression[1].

```
sqr(5)
  where
    a = 1
    sqr(x) = x*x
  do
    s.pop()
```

The evaluation is as follows: any expressions in the `where` part are evaluated. Then, all expressions in the `do` part are evaluated. Finally, the main expression is evaluated and this is the result of the whole.

A `where` / `do` may be used separately from an expression, to create a new scope (as used in function definitions that return no value).

[1] Further "except" and "close" suffixes are not shown, as exception handling wasn't implemented at all in the current version of Bla.

4.4.2 Function Calls and Instance Variables

A function is called, by specifying its name followed by arguments in compulsory brackets:

```
fun(1,2)
```

You may supply the arguments in the same order as they are specified in the interface. You may leave the last few arguments out if they have defaults. An alternative way of calling functions (preferred if the number of arguments is greater than three) is a tagged call: for each argument you specify the name of the argument (as in the interface) followed by ":" and the actual expression. You need specify only those arguments that don't have defaults, and arguments may be specified in any order:

```
blob(x: 10,
     y: 20,
     size: 5,
     colour: 1)
```

Normally functions are called within the current scope. You may specify some other scope (an object) by using the . operator:

```
grass.digest()
```

You may apply . many times, and the object specification may be any expression:

```
(a=1 -> cow | cat).selectfood().digest()
```

`selectfood` is called within the scope of `cow` or `cat` depending on the conditional, the result of which is the object on which `digest` is invoked.

The same . operator is also applicable to variables, i.e.:

```
cow.defaultfood.quantity
```

will select the variable `quantity` within the object `defaultfood` which is obtainable from the interface of the `cow`.

4.4.3 Assignment

Assignment is an operator that works on two expressions, the left hand side being the destination for the value on the right hand side. The left hand side thus needs to be what is often referred to as an *lvalue*, i.e. a variable (augmented with lots of .s if necessary):

```
a:=1
cow.defaultfood:=grass
```

4.4.4 Control Structures

Bla has one conditional (an "if" / "case") and one iterative (a "loop") control structure:

The most general form of the conditional structure is:

```
case_exp | condition -> exp | ... | default
```

There may be arbitrarily many `condition -> exp` blocks. Evaluation is as follows: the `case_exp` is evaluated, then matched to all `condition` patterns one by one. If one matches, the corresponding `exp` is evaluated, otherwise `default` is evaluated.

The `case_exp |` part may be left out: in this case each `condition` is a boolean expression instead of a pattern.

The `| default` part may also be left out: the expression will then not return a value (which only makes sense with side-effect code).

Examples:

```
a=1 -> puts(´a is 1 !\n´)
```

```
a:=(c -> 1 | 2)
```

```
fun(x) = a(x) -> 1
       | b(x) -> 2
       | 3
```

```
c | "\n" + " " + "\t"              -> ...
  | "a" .. "z" + "0" .. "9" + "_" -> ...
  | "-" + "+"                     -> ...
  |  ...
```

```
(x,y) | ([],     ´empty´) -> 0
      | ([1|_],  ´one´)   -> 1
      | ([_,b|_],´two´)   -> fun(b,c)
```

A loop in Bla consist of 3 optional prefixes to a `where` / `do` block, of which at least one needs to be present:

- `loop` followed by zero or more variable declarations separated by commas, which will be available only in the loop.
- `while` followed by an expression which tells us when the loop is over.
- `next` followed by an expression that advances whatever we're looping over to the next item.

Examples: looping a from 1 to 10

```
loop a = 1 while a<=10 next a++ do
  putc(a,stdout())
```

Iterating all elements of list `x`:

```
while x next x:=t do
  x <=> [h|t]
  h.bark()
```

Additionally, `exit` will terminate any loop. An optional identifier to `exit` signifies which loop to terminate (rather than just the enclosing one).

```
loop x do
  ...
  a<>1 -> exit x
  ...
```

Chapter 5

Example

In this chapter we will look at a real-world example written in Bla, a ProLog interpreter. We will use this example to highlight some Bla features in the context of a real program. All the verbatim parts in this chapter (i.e. typeset in `Courier`) together form the source code, and can be compiled and run without further modifications.

This program was translated from a similar one written in "E" [Oor93], which in turn was translated from LISP (there's a minimal version in [Cam84]), and greatly transformed / enhanced along the way. However, programming styles of these two previous languages may still shine through.

```
foldr(f,z,[])    = z
foldr(f,z,[h|t]) = f(h,foldr(f,z,t))
```

Included directly because the module system wasn't implemented at the time of writing (`foldr` is part of the standard set of list processing functions). The Type inferencer will deduce `((T,U)->U,U,[T])->U [T,U]` as the type for `foldr`, which is a particularly easy example to follow.

```
stdout():int extern
stdin():int extern
getc(file:int):int extern
strcmp(a:string,b:string):bool extern
puts(s:string,file:int):int extern
put(s) = puts(s,stdout())
```

Standard I/O functions have also been added in a very ad-hoc manner. `extern` allows you to manually reference a function in another module (in this case a built-in function), but in a normal implementation this would never be necessary: the module system takes care of that automatically.

```
prolog() = self where
```

The start of the ProLog engine. `prolog` is a function that will return a ready for use engine-object (by returning its own function environment, `self`). The functions below (note the indentation) are local to this environment.

```
  v()
  cell(h:v,t:v):v  implements v
  atom(s:string):v implements v
  var(s:string):v  implements v
```

The types of values of ProLog expressions used in the interpreter. `v` is the *base class*, `cell`, `atom` and `var` are declared as subtypes of `v` (using `implements`). Note that `= self` is the default right-hand side for functions, which facilitates easy creation of "term"-like constructor functions, as shown here.

For example, the prolog code `X` is represented as `var(´X´)` and `[x|y]` as `cell(atom(´x´),atom(´y´))` . A predicate or term like `blerk(a)` is represented as `cell(atom(´blerk´),cell(atom(´a´),nil))` i.e. the same way as `[blerk,a]`, to reduce complexity (see `pred()` below).

```
database:[[v]] = []
setdatabase(b) do database:=b
```

prolog has only one local variable, database. It is by default private, and so an access function is provided. Types are given explicitly for databases and the above types, for clarity[1].

```
pred(h,l) = cell(atom(h),foldr(cell,nil,l))
```

Predicates are built out of cells for simplicity in *unification*. This function simply allows the client of the engine to provide arguments as Bla lists, onto which we then fold the cell constructor.

```
molec(l:int,e:v)

lvl(m:molec) = m.l
xpr(m:molec) = m.e
```

lvl is an example of where the Type inferencer would fail if the :molec type spec were not available: upon type-checking the . operator it insist that the type of m is narrowed down sufficiently to contain at least l (see chapter 6.2.1 on this subject).

A molec is a tuple of (level,prolog_exp), where level is used to discriminate variables at different levels in the proof tree. Two of these make up a binding in a global bindings list (the env-list). This is arguably not the most efficient way of handling variables, but it certainly makes it simple (many features of this engine are designed for simplicity).

```
bind(x:molec,y:molec,e:bind)
```

This makes up the list of bindings.

```
bond(_,nil) = nil
bond(_,bind(_,_,nil)) = nil
bond(molec(xl,xe)/x,bind(molec(yl,ye),b,e)) =
  xl=yl and equal(xe,ye) -> b | bond(x,e)
```

Searches a list of bindings for a certain variable, respecting levels. Note the pattern matching on types: molec(xl,xe) is not a function call, but a pattern. It matches an object of type molec with two fields, which are bound to xl and xe.

```
equal(nil,nil)                 = true
equal(cell(xh,xt),cell(yh,yt)) = equal(xh,yh) and equal(xt,yt)
equal(var(x),var(y))           = strcmp(x,y)
equal(atom(x),atom(y))         = strcmp(x,y)
equal(_,_)                     = false
```

Standard *structural equivalence* stuff for any ProLog expression. Note that the and is *short-circuit* so equal will finish as soon as an inequality is found.

```
lookup(nil,env)               = nil
lookup(molec(_,nil)/p,_)      = p
lookup(molec(_,var(_))/p,env) = lookup(bond(p,env),env) or p
lookup(p,env)                 = p
```

lookup retrieves values through levels of variable bindings. In the unification process a value bound to a variable may actually be various levels deep as variables may be bound to variables. lookup returns the original variable if it is unbound.

[1] As rule of thumb, I personally put explicit types whenever the reader would have to search a lot through the code to find the inferred type.

```
gety(a) = (c="\n" -> a | gety(c)) where c = getc(stdin())

toplevel(env) = (gety("n")="y" -> 0 | -1) do
  showenv(env,env)
  put(´\nMore? (y/n) ´)
```

This function will be called when the `seek` function below has finished proving everything, allowing the user to request more solutions or "call it a day" (see below for what these integer values do).

```
prove(goals) do put(seek([goals],[0],bind(nil,nil,nil),1) -> ´yes.\n´
                                                           | ´no.\n´)
```

Starts a query.

```
seek(nil,_,env,_) = toplevel(env)
seek([nil|goalrest],[_|ntail],env,n) = seek(goalrest,ntail,env,n)
seek([[goalh|goalt]|goalrest],[nhead|_]/nlist,env,n) =
  seekdb(database,molec(nhead,goalh),[goalt|goalrest],env,n,nlist)

seekdb(nil,_,_,_,_,_) = nil
seekdb([[head|tail]|db],goalmolec,rest,env,n,nlist) =
  (`env2:=unify(goalmolec,molec(n,head),env)) and
    (`tmp:=seek([tail|rest],[n|nlist],env2,n+1))
      -> tmp<>n and tmp | seekdb(db,goalmolec,rest,env,n,nlist)
```

`seek` tries to prove a `[[goal]]`. The other three arguments are the current list of variable bindings (the environment) and level control information.

`seekdb` tries to unify the current goal with the head of each clause in the database (the `unify()` call), if it succeeds it will also try to prove the predicates of that clause (i.e. the ones after the `:-` in ProLog syntax), using the `seek()` call. On failure it will try an alternative clause from the database (the `seekdb()` recursive call).

The return value is either 0 for failure so it will *backtrack* to try something else, or a level number to go back to (used by the *cut* predicate). What `toplevel` does is simply "unwind" the interpreter by returning `-1` (the last line of `seekdb` shows how levels are handled). Together with `unify()` below, this is the interpreter core.

Backtracking is implemented using a well-known technique for mapping backtracking onto function-oriented architectures: creating a stack of goals (the first argument of `seek()`) that still need to be proved, going down deeper in the dynamic call-graph when goals succeed, staying at the same level or going back when backtracking (failing). As a consequence, the program will have arrived somewhere at the bottom of the call graph (with an empty goal-stack) when it reaches a sollution. This way of structuring ProLog interpreters is about as simple as you can get since there's no need to manually keep track of choice-points and variables te be unbound upon backtracking, but it's not very efficient.

```
unify(x,y,env) = unifyl(lookup(x,env),lookup(y,env),env)

unifyl(molec(xl,xe)/x,molec(yl,ye)/y,env) =
  (xl=yl and equal(xe,ye) and env) or
    ((xe,ye) | (nil,nil)         -> env
             | (nil,var(_))      -> bind(y,x,env)
             | (nil,_)           -> nil
             | (var(_),nil)      -> bind(x,y,env)
             | (_,nil)           -> nil
             | (var(_),_)        -> bind(x,y,env)
             | (_,var(_))        -> bind(y,x,env)
             | (atom(a),atom(b)) -> strcmp(a,b) and env
             | (cell(xeh,xet),cell(yeh,yet)) ->
                 (env:=unify(molec(xl,xeh),molec(yl,yeh),env)) and
                   unify(molec(xl,xet),molec(yl,yet),env)
             | _                 -> nil)
```

Unification of two level-tagged expressions. First performs a `lookup()` of both expressions to remove any variable bind indirections that may be present. `unifyl()` tests if both expressions are equal, if they are, we need not try any further (also important to prevent recursive variable bindings). If this was not the case, the second part of the `or` does a case-selection. Returns an environment, possibly with new bindings, or `nil` for failure. Note the number of cases that handle `nil` expressions, and the particular order they have to be in: a `nil` object would have been a better idea.

```
  showenv(bind(_,_,nil),_) do nil
  showenv(bind(molec(0,x)/h,t,e),env) do
    show(x)
    put(´ = ´)
    show(convmolec(h,env))
    put(´; ´)
    showenv(e,env)
  showenv(bind(_,_,e),env) do
    showenv(e,env)

  show(nil) do put(´nil´)
  show(atom(s)) do put(s)
  show(var(s)) do put(s)
  show(cell(h,t)) do put(´[´); show(h); put(´|´); show(t); put(´]´)
```

showenv shows all variable-bindings of the goal the user typed (which is level 0).

```
  convmolec(m,env) = (`lv:=lookup(m,env))=m -> atom(´?´)
                                             | convexp(lvl(lv),xpr(lv),env)

  convexp(l,nil,env)       = nil
  convexp(l,cell(h,t),env) = cell(convexp(l,h,env),convexp(l,t,env))
  convexp(l,var(_)/e,env)  = convmolec(molec(l,e),env)
  convexp(l,e,env)         = e
```

These two convert an arbitrary ProLog expression to one where all variables are replaced by whatever they're bound to, which is useful to show final solutions to queries.

```
main() do
```

Call the ProLog engine with a sample database and some goals. For a real ProLog interpreter there should be a parser for this instead.

```
  `p:=prolog()
```

Calls the function `prolog` to create one engine (we could create infinitely many). Variable `p` is declared on the fly (using `).

```
  p.setdatabase([
    [p.pred(´true´,[])],
    [p.pred(´eq´,[p.var(´X´),p.var(´X´)])],
    [p.pred(´append´,[nil,p.var(´X´),p.var(´X´)])],
    [p.pred(´append´,[p.cell(p.var(´X´),p.var(´Y´)),
       p.var(´Z´),p.cell(p.var(´X´),p.var(´U´))]),
       p.pred(´append´,[p.var(´Y´),p.var(´Z´),p.var(´U´)])],
    [p.pred(´father´,[p.atom(´med´),p.atom(´small´)])],
    [p.pred(´father´,[p.atom(´big´),p.atom(´med´)])],
    [p.pred(´grandfather´,[p.var(´X´),p.var(´Y´)]),
      p.pred(´father´,[p.var(´X´),p.var(´Z´)]),
      p.pred(´father´,[p.var(´Z´),p.var(´Y´)])],
    [p.pred(´call´,[p.var(´X´)]),p.var(´X´)]])
```

Creates a sample database, equivalent to:

```
-- true.
-- eq(X,X).
-- append([],X,X).
-- append([X|Y],Z,[X|U]):-append(Y,Z,U).
-- father(med,small).
-- father(big,med).
-- grandfather(X,Y):-father(X,Z),father(Z,Y).
-- call(X):-X.

p.prove([p.pred('grandfather',[p.var('X'),p.var('Y')])])
p.prove([p.pred('append',[p.var('X'),p.var('Y'),
  p.cell(p.atom('a'),p.cell(p.atom('b'),p.cell(p.atom('c'),nil)))])])
```

Calls the engine with the queries `grandfather(X,Y)` and `append(X,Y,[a,b,c])`, which are pretty classical examples. The latter is the smallest example showing some ProLog power, as it tries to find various combinations of two lists to satisfy the query.

Chapter 6

Implementation & Semantics

6.1 The implementation

A sufficiently complete implementation of the language described in this document has been made. The structure of the implementation is as follows:

- A (portable) compiler written in C++, that translates Bla source code to a Bla virtual machine called "Emmer" (read all about that in chapter 6.3), as portable binary file format.
- Various back-ends that read and / or transform Emmer binaries. Currently available are only
  - `bdsm` disassembles an Emmer file to readable ASCII
  - `bint` executes an Emmer file. The current interpreter is targeted towards safety and correctness rather than speed: it checks everything dynamically, which helped a lot while debugging the compiler's code generator.

  Planned are C and native code generators (and other tools).

The lexical analyser and LL-parser are very straightforward. The choice to not use parser generator tools came from the fact that Bla's grammar is easy to parse in top-down fashion, and precise parser control was needed for error-recovery[1]. The parser generates an *AST* with all symbol and scope information incorporated in it. The AST is shaped as an inheritance-graph, and all subsequent passes use virtual methods to do comfortable tree walks.

The second pass does type-checking and inference, it also takes care of required program transformations such as making one function body out of multiple pattern-match cases. The type inferencer is by far the largest (in terms of code-size, KLOC, or whatever) and the most complex component of the compiler. Shared among the inference code for specific AST parts are sets of functions to do unification of types, cloning, binding and scanning (to infer type variables). The actual algorithm differs quite a bit from specifications of type inference, since the use of mutable data structures helps greatly in implementing variables (very similar to implementations of ProLog interpreters).

The last pass generates the Emmer code, which is rather straightforward. The only slightly tricky part here is keeping track of the stack in connection with the control flow of complex pattern matches. All parts of the code generator deal with keeping the stack at the right level, since each AST-expression can generate code that does or doesn't leave a value on the stack. At the end of this pass the binary Emmer file is written.

A sample interaction with the Bla system:

```
# bla prolog
```

compiles the source `prolog.bla`, currently the most interesting piece of example code, a full ProLog interpreter (`#` is the prompt). The result is the file `prolog.il`, which can be executed by typing:

```
# bint prolog
```

Read the source code or chapter 5 to see how it works. To have a look at the code generated by the Bla compiler, type:

```
# bdsm prolog | more
```

or something similar. Both `bla` and `bint` have some command line options to print debugging output, and to trace execution, dump stacks / opcodes etc.

6.2 Bla Type Inference

The whole of type inference in Bla is based on the very well known Hindley-Milner type inference system (I found [Car87] very enlightening on this subject, another good introduction is chapter 7 of [FH88]. A classic is [Mil78], a related paper well worth a read is [CW85]).

As far as the functional subset of Bla goes, Bla's type inferencer is supposed to work exactly as the type inferencers for languages like ML [MTH90] and to some extent

[1] The large speed-increase was only a secondary consideration.

Haskell [HPW92]. Very standard code such as the well-known list processing functions in `list.bla` are all inferred without manual type annotations, as expected. Various tests showed that the inferencer could handle similar code to Hindley-Milner, and rejects / fails on cases which are known to be problematic in Hindley-Milner: recursive types such as in `x <=> [y|y]` will result in an error, premature type variable bindings such as in `id(x) = x do id(3)` gives `id` the type `(int)->int`, etc.

6.2.1 Limitations on OO type inference

The differences kick in where Bla is different from a purely functional language. It is important to note that, in designing the Bla type system, it was never an aim to make every possible piece of code inferrable, i.e. in Bla type annotations are not entirely superfluous. Doing *full* type inference for an OO language with side effects is, depending on the exact semantics of the language in question, either very complex or impossible. Palsberg and Schwartzbach are among the few people who have been researching this, [PS91] is a good start. Some of the things they accomplish in this paper Bla was clearly not meant to do.

A clear restriction in Bla's inference algorithm was not to infer classes from methods, e.g. `x.f()`, where `x` is not inferrable by any other means. The obvious action to take would be to search the inheritance graph for a unique most general class with an `f` feature. Especially in the context of multiple inheritance this was deemed very unnatural. In this particular case the declaration of `x` will need an obligatory type specification. Note that this doesn't happen too often, as one other usage of `x` that associates it with a type that contains an `f` will be enough to give the type inferencer a clue what to do.

The objective of type inference in Bla was merely to remove 95%[2] of the unnecessary clutter of types, and to do so in a statically type-safe way. Looking at example code one can hardly find any type specification, and of the ones that are there, half are unnecessary. A simple stack ADT can be declared and used without using a single type specification.

6.2.2 Bounded unification

Classical unification of types as used for example in Hindley-Milner results in a generalisation of both types involved. In a language like Bla there are constructs where this is inadequate: assignment is a perfect example, but other constructs require more than this as well. This introduces a second kind of unification in the Bla type inferencer, which I call *bounded unification*[3], which also cooperates very nicely with subclasses. Basically it says that one of the two types is the upper bound for a type in unification. If a more general type than the bound would be needed to unify the two, this results in an error (rather than being accepted). In an assignment the left-hand side is the upper bound. In a function application, the type of the formal parameter would be the upper bound against the type of the actual parameter, whereas for the return type this would be exactly the reverse. Other constructs such as branches of an `if-then` use traditional unification.

To show a rather involved example of how deeply this cuts into the semantics of a type system where type parameters are concerned, consider this contrived but realistic example[4]. We assume a class `person` with a subclass `employee`, in addition to our traditional parametrised `stack[T]`:

```
do_person[T < person](s:stack[T]) do
  person_fun(s.pop())
```

This is allowed, thanks to `person` being the upper bound when compared to the type parameter `T` in the application of `person_fun`. What is perfectly catered for is that we can call `do_person` with a stack of employees, and `person_fun` won't mind dealing with an employee.

```
push_a_person[T < person](s:stack[T]) do
  s.push(person())
```

This function looks deceptively similar in structure to the previous function, except that here the roles of bound towards the type parameter are reversed! Of course, the type `person` can never be less general than any specific instance of type parameter `T`, so this code needs to be rejected otherwise we could end up with an employee stack with an odd person in it! Luckily Bla's system of bounds detects this naturally, even in a tricky situation like this[5].

6.2.3 Type parameter binding stacks

Another special feature of the Bla type system is what could be called "type parameter binding stacks": keeping track of temporary bindings through various levels of environments. Nested function definitions may *each* bring along their own set of type parameters. When using such a deeply nested function out of context using selectors (`.`) on expressions resulting in environments (as in `a.b().c()`, for example), type variable bindings of lower level environments need to be kept around, since they might be essential to type-check the result of an inner function. This is obviously something not catered for in Hindley-Milner: not only do functional

[2] 100% for purely functional code.

[3] I hope this doesn't clash with some preconceived notions people may have ;)

[4] At this level, type systems of other, comparable languages simply fail, i.e. don't allow code like this or result in serious loopholes in the type system. Eiffel comes to mind. Type problems shown in this chapter go beyond normal co- / contra-variance problems, and to my knowledge have not been seriously studied.

[5] I say "tricky" because this clash of unification and type parameters doesn't occur that frequently, only if we take arguments that have types which are parametrised themselves, i.e. apart from the function itself.

languages generally lack something equivalent to the . operator, it is the first-class environments teamed with Bla's expressions in its full orthogonality that make it particularly hairy[6].

6.3 The Bla virtual machine: "Emmer"

Emmer is a stack-machine designed especially for Bla (examples of other similar stack machines are [Car86] and [jav95]). As such, core Bla features such as 1st class environments are part of the virtual machine as well, though it contains more low-level information to tell the back-end how to generate good quality code.

6.3.1 File Format Layout

The general layout was inspired by the IFF file format [Inc92], and is organised into largely autonomous, possibly hierarchical *chunks*. Each chunk is identified by a name, the length & version of the chunk. Implementations should ignore chunks they don't know about, and reject chunks they do know about but have too high a version number: this facilitates safe file format changes and extensions.

The `INFO` chunk stores information about the source from which the code was compiled. There may be more than one of these since Emmer linkers and global optimisers may coalesce several Emmer modules into one. The `FUND` / `FUNR` (FUNction Definition / Reference) chunks define functions, where `FUNR` is a subset of `FUND` since it defines a function whose implementation resides in another module. Function chunks are not nested i.e. the compiler lifts all local functions and lambdas to file level.

As for all code & data structures in the Bla implementation, but especially in the Emmer file format, care has been taken to get rid of any form of hard-coded limits. Virtually everything is encoded with arbitrary precision integers, can be used on machines with any integer size and endianness[7], without recompiling.

Apart from the usual information about arguments, types, and implementation hints (such as "the environment can be stack-allocated"), the most important part of the `FUND` chunk is the actual code. The instructions are encoded as at least one byte each, followed by the appropriate number of arguments encoded as arbitrary precision integers (or strings etc.). Here is a not too detailed overview of all opcodes with examples. Stack changes are denoted as usual in forth-type languages: `(bottom..top - bottom..top)`, for example `(a,b-c)` means that the instruction expects to pop `b` from the top of the stack and `a` below that, and pushes `c` back when done.

[6] And I mean *hairy*. I do not claim to have harnessed all potential semantic interactions in this area.

[7] 32 bits or more of course, I hasten to add.

6.3.2 Data Movement Instructions

```
ld n            (-x)
st n            (x-)
parld n         (-x)
parst n         (x-)
indld n         (e-x)
indst n         (x,e-)
argld n         (-x)
argst n         (x-)
```

These are responsible for moving values between environments and the stack. `ld n` for example takes the value at position `n` of the current environment (e.g. `self`) and pushes it onto the stack. The `par` versions do the same thing for `parent`[8], `ind` uses any environment pushed on the stack previously, and `arg` accesses arguments (which reside on the stack at first). For example, compiling `a:=e.b` where `e` is part of the current function, and `a` of the enclosing one:

```
ld 1
indld 0
parst 2
```

The position integers are chosen arbitrarily.

6.3.3 Integer operations

```
val x           (-x)
valn            (-0)
```

These two push an integer value on the stack (`valn` is just a shortcut).

```
add             (x,y-z)
sub             (x,y-z)
mul             (x,y-z)
div             (x,y-z)

neg             (x-y)
not             (x-y)

and             (x,y-z)
or              (x,y-z)

eq              (x,y-t)
uneq            (x,y-t)
higher          (x,y-t)
lower           (x,y-t)
higheq          (x,y-t)
loweq           (x,y-t)
```

[8] The current implementation misses level information, like "clos" below.

These perform very predictable operations, note however that `and` and `or` are bitwise operations, and as such are not use for implementing the Bla `and` and `or` keywords, which are short-circuit logical operations. As an example, `-a+1<10` would be compiled as:

```
ld 0
neg
val 1
add
val 10
lower
```

and leaves a truth value on the stack (see chapter 6.3.6 what happens next).

6.3.4 Lists, Tuples and Objects

```
cons            (tail,head-cell)
tuple n         (arg1,arg2,...-tuple)
```

Construct cons-cells and tuples. Putting the tail first is more natural when constructing long lists.

```
hdtl            (cell-tail,head)
tupd n          (tuple-arg2,arg1)
```

Deconstruct cons-cells and tuples back onto the stack again. The order is determined by pattern matching. An example: `[1,2,3] <=> [0,x|_]` would be compiled as:

```
valn
val 3
cons
val 2
cons
val 1
cons
hdtl
bt 0
hdtl
st 0
drop
```

The first two instructions push 3 and a `nil` tail on the stack, `cons` then makes this into `[3]`. The next four instruction continue to construct the list out of 2, 1 and the previous list, resulting in `[1,2,3]`. `hdtl` splits the cons-cell at the top of the stack, resulting in 1 and `[2,3]` on the stack. `bt` tests the 1 for being 0, which fails here (Arriving at label 0 would mean `false`). If the condition was met, `hdtl` splits up the rest resulting in 2 and `[3]`. `st` stores 2 in variable 0 of the current `self`. `drop` throws away the last list bit, since it is not needed (caused by the pattern `_`).

For objects we have:

```
self            (-self)
parent          (-parent)
```

Note that these of course don't construct objects, they merely get you a reference to existing ones. Obviously, there are no instructions to create objects: this is implicit in function application.

```
ttype f         (o-b)
```

Tests an environment object to be from a certain function `f` (results in a boolean). This is used in the implementation of object pattern matching.

6.3.5 Closures and Application

```
clos f,n        (-c)
close f         (e-c)
```

Closures are tuples consisting of a function, and a parent environment to be inserted whenever the function is applied[9]. Both construct closures for function `f` the difference is where they get their parent environment from: the latter simply from the supplied environment on the stack (as in the expression `e.f`), the former from `self` (as in the expression `f`), where `n` is the number of levels down in the nested function hierarchy we need to go (remember that we might construct closures out of functions that are actually part of our parent etc.).

```
jsrcl           (?,c-?)
```

Function application. Takes a closure, and any number of arguments.

```
jsr f           (?-?)
jsrm n          (?-?)
jsrme n         (?,e-?)
```

Not used at this moment. used to optimise function application when the function is known, and to implement calls to functions part of an inheritance hierarchy.

```
sys n           (?-?)
```

Used to call various built-in functions, such as vector memory allocation etc.

[9] This allows for all those beautiful higher-order function programming techniques :)

6.3.6 Branching and Exceptions

```
lab l           (-)
bra l           (-)
bt l            (t-)
bf l            (t-)
```

`lab` sets label `l` at that point in the code, `bra` unconditionally branches to it, `bt` and `bf` (Branch True / False) will branch depending on truth value `t` on the stack. Worth noting about these instructions is that dataflow remains statically checkable, i.e. all control flow paths arrive at one label with the same stack height (all code generation routines in the Bla compiler maintain this invariant, and a back-end should really check for this). Having this information is valuable for a back-end to be able to easily generate reliable code and do straightforward optimisations: simple but very effective register allocation is peanuts this way.

as an example, `a>1 -> 2 | 3` could be compiled as:

```
ld 0
val 1
higher
bf 0
val 2
bra 1
lab 0
val 3
lab 1
```

Observe that at any point the shape of the stack frame is statically known. This is of course a trivial example, but check some complex pattern-matches using `bdsm` to see that this is not always trivial[10].

```
ret             (?-)
jtab n,l,l..    (x-?)
```

`ret` quits a function prematurely (used for implementing the Bla `return` keyword), `jtab` is meant for optimising large `if-then` / `switch` code constructs (Bla's `->` and `|` operators).

```
raise           (x-)
try l           (-)
endt            (-)
```

meant for implementing exception handling. `raise` raises an exception, `try` and `endt` denote an area of code in which exceptions are monitored and handled by the code at label `l`.

[10] Note that bdsm translates labels to "n:" on the next line instead of "lab n".

6.3.7 Stack Manipulation and Misc.

```
dup             (x-x,x)
drop            (x-)
swap            (a,b-b,a)
pick n          (-x)
rot             (a,b,c-b,c,a)
```

Of these stack manipulation operations only the first two are currently used. Note that in anything but a trivial back-end these operations would not result in code, rather changes in the compiler's tables of allocated registers. They also show changes in reference counts very explicitly, which can be useful.

There are some other instructions for string & vector manipulation of which some are only partially implemented. they won't be discussed further here.

```
strc str        (-s)
```

`strc` constructs values of type `string` using character data held in `str`.

```
idx             (i,v-x)
idxc            (i,s-x)
idxs            (x,i,v-)
idxsc           (x,i,s-)
```

These store and retrieve value from strings (the `c` variant) and vectors (`i` is the index).

6.3.8 Example Emmer Code

As an example the code generated for `append()` is shown here[11]. The four items between square brackets mean: 2 args, 1 local for h, stack allocatable, no parent.

```
append[2,1,S,N] {
   argld 0
   bt 0
   argld 1
   bra 1
0: argld 0
   hdtl
   st 0
   argld 1
   jsr append
   ld 0
   cons
1:
}
```

As a last slightly more involved example: `stack()`:

[11] This will differ from the bdsm output, as various optimisations might not have been applied.

```
stack[0,1,D,N] {
   valn
   st 0
   self
}

pop[0,1,S,P] {
   parld 0
   bt 0
   val stack_empty
   raise
0: parld 0
   hdtl
   st 0
   parst 0
   ld 0
1:
}

push[1,0,S,P] {
   parld 0
   argld 0
   cons
   parst 0
}

isempty[0,0,S,P] {
   parld 0
}
```

Chapter 7

Conclusions

We have done an exercise in language design where we took a construct that has always been perceived as an implicit element of the language semantics (the function environment) and changed it to a powerful feature just by making it first-class. We upgraded this basic semantic model with state of the art programming language constructs to result in a coherent, usable, real-world language. Furthermore we have put serious effort into providing a non-toy implementation of the language to prove the viability of the whole as a practical tool, stressing a fast, compact and solid type inferencer and abstract machine architecture.

First-class environments have indeed shown to enlarge the expressive power of a language enormously, through a tiny semantical change. There's hope that this sort of design will become more trendy in the future, with languages like Self and Beta becoming more popular and paving the way to more advanced designs. As both objects and functions become more orthogonal and flexible, designers will find themselves doing everything twice and burdening programmers with unnatural separations of concepts in languages. On the downside, where Bla's advantage over purely functional languages is obvious, the number of programming techniques using first-class environments that have been discovered (to give Bla an edge over languages that simply use twice as many language constructs) is not too large (so far!).

Type inference is a marvelous thing, and *every* language should have it, really. Type inference in Bla is very practical and has been shown[1] to be safe for the constructs involved, though the study of its interaction with imperative / OO constructs could go deeper, and be more solid than in Bla. This is not necessarily a shortcoming of Bla, rather something one could profit from more.

In the first designs of Bla it was decided to be politically correct and go for a statically type safe language, even though I have always favoured typeless and dynamically-typed languages. Bla's type system turned out well, but looking at the tiny details one will always find rough edges. I keep to the statement that the semantics of a type system should be fully dynamic, and that static type-checks (through clever type inference) should be part of the *implementation* rather than the *language*[2]. The difference in programming errors caught should be nearly non-existent, and the language design will be clean and not unnecessarily restrictive.

From the various techniques of rooting a language in one abstraction mechanism, as shown by Bla and Symmetric Lisp, Beta and Self in chapter 3.1, Bla is the clear winner. The road ahead for First Class Environments is therefore not so much tuning the concept itself, rather its interaction with other features: dynamic typing, alternative subtyping mechanisms (including delegation) and referential transparency come to mind[3].

[1] Well, "assumed" here and there ;)

[2] The actual approach taken is not that different: instead of trying to push the limits of what is allowed in a language while still being type-safe, and trying (in vain) to remove restrictions on code, one simply tries to put enough intelligence in the type inferencer. The type inferencer will warn when a construct is sure to cause problems at run time, rather than impose arbitrary restrictions.

[3] Each of these has been considered extensively, but weren't matured enough to be part of Bla at the time of writing.

Appendix A

Glossary

Here I try to shed some light on more technical terms used in this text, for readers with a less comprehensive background in programming languages. These are not definitions of the concepts, and I do not claim correctness of these descriptions *at all*. Some of these concepts are even controversial as to their exact meaning; if you want a precise background you will have to look beyond this text (suitable references are provided here and there).

AST: "Abstract Syntax Tree", a tree-like structure in the front-end of a compiler that represents the parsed input program, without the syntax bits that are irrelevant to the semantics. For example, using *term* syntax, `2*(a+3)` would be represented in memory as `times(num(2),plus(ident("a"),num(3)))`. AST's are central structures in a compiler, as they are often the level at which type-checking occurs, and sometimes also program transformations / optimisations.

backtracking: a technique to implement non-determinism in a program. The program will keep track of a tree of *choice points*, that can be used to select another path if one branch fails.

base class: a class that forms the root of a certain class-hierarchy. Also *abstract base class*: a base class whose only purpose is being the root, i.e. specifying *interface* and no *behaviour* (popular in languages without special *subtyping* mechanisms).

behaviour: how a class behaves when its *interface* is accessed. Two classes may have the same interface, but different behaviour. See *inheritance* and *subtyping*.

black box: an object that can only be accessed via a specified interface, and whose implementation is completely hidden from outside interference (including *subclasses*).

call graph: a graph where the nodes are all functions in a program and the edges denote calls from one function to the other. Call graphs find their use in optimisers, debuggers, and the study of *reuse*.

client: the code that uses a class (apart from subclasses).

clone: copy an object to create a new one, often used in *prototyping* languages. Alternative to *instantiation*.

closure: a tuple consisting of a function and a function environment (the one where the closure was created). Used to implement lambda's with free variables, or lazy evaluation (as functions with no arguments).

compile-time type-safe: the types of values in a program can be checked to such a precision at compile time that no runtime type-checks are necessary and at runtime no operations on incompatible arguments can occur.

cons cell: the simplest dynamic data structure consisting of two fields, originating in Lisp (one of the original papers well worth reading is [McC65], see also [McC78]). In its original form both fields of the cons cell could be used for anything, allowing one to build any data structure, most notably lists and trees. In recent incarnations, especially in strongly typed languages such as Bla and Haskell, the first field (the *head*, or *car* for historical reasons) contains the data, and the other field the pointer to the next cell (the *tail* or *cdr*), restricting its usage to lists.

constructors: a special method of an object that has the task of initialising it. Available in many OO languages.

continuation: a mechanism found for the first time in Scheme [WC91] that makes the function call mechanism first class. Instead of calling function B from function A, and returning to A when B is finished, you can also give the remaining code of function A as argument to B, and have B execute it when done: the difference is, however, that B can choose whether to return to A, or maybe do something else, or store the continuation someplace etc. This caters for some interesting programming techniques.

contravariance: the "variance" of a language says what constraints formal parameter types of methods of subclasses need to obey with respect to their superclass equivalents. Keeping the type the same is always correct but slightly limiting: this is called *no-variance*, and used by for example C++ [Str91]. Observe that if subclasses need to be compatible with the superclass, these types may also be more general: this is called *contravariance* (used by for example Sather [SO94] and Bla). As the name suggests, this correct way is also unnatural, since specific classes may want specific types to deal with: a type system that allows this is said to be *covariant*. The classic example is an animal hierarchy, where the class `animal` has a method `eat` with argument type `food`. In the `pony` class we will want to narrow `food` to say `grass`, and not make it more general. The problem now is that `pony` is not a subclass

anymore, since it cannot do everything an `animal` object can do; it will choke on `my_little_pony.eat(lotsa_meat)`. Languages with covariance such as Eiffel [Mey92] will resort to global type-checks and / or dynamic type-checks to fix this. A last possibility is `any-variance` which is used mostly in dynamically typed or typeless languages such as Smalltalk [GR83] or E [Oor93]: they don't prohibit any constructs that might give type clashes at run time, so also any direction of "variance" is allowed.

covariance: see *contravariance*.

cut: a special predicate in logic languages such as ProLog [Cam84] to prune branches of the tree of choice-points manually. See also *backtracking*.

data-hiding: the technique of hiding the implementation of a piece of software from its *clients*, who only access it via its *interface*. The purpose of this is to reduce the dependencies between parts of the software, making modifications to code easier, and helping to prevent hard-to-find bugs.

delegation: a way of sharing behaviour between objects, an alternative to inheritance with classes. One (or more) objects can make another their delegate object or *parent*, and when they receive a message they can't deal with, it is passed on to the parent. Often used in languages that take the *prototyping* approach to OO.

dynamic dispatch: the implementation technique necessary to make OO method calls work. Since the precise type of an object is not always known, objects often carry tables of methods around, used to dynamically select the method appropriate for that object.

environment: a data structure containing all the values of names local to a function. Implementation terms are *activation record* or even *stack frame*.

Exception Handling: a control structure mechanism meant for handling "exceptional" situations in a program, layered on the dynamic version of the call graph. Exceptions allow for a decoupling of the algorithm structure and the treatment of "problems" which is essential for modular / large-scale programming and robustness of an application. A problem is signaled by *raising* or *throwing* an exception and other parts of a program may deal with the problem by defining an *exception handler*. The scope of exception handlers (which may be structured at various levels of an application) is determined dynamically by the call chain of functions.

first class value: generally, an entity is *first class* when it can be treated like other general values in a language, such as integer values, i.e. it can be passed as an argument to functions, stored in data structures, etc. Environments in Bla fit this description, and so do *continuations* in Scheme. Other languages too have environments and continuations (both hidden in the function call mechanism), but they are not accessible as values. A clear example of something that is not first class in Bla is a *type*: you cannot create a list of types and then later on use them to assign a type to a variable in a declaration.

free variables: variables used in a function body which are declared in an enclosing function. The reason why these are so special is that if the inner function is used as a function value and evaluated in another context, the references to these variables would be lost. That's why in most functional languages function values are packed together with the environment that contains these variables into a *closure*. Non-functional languages tend to simply forbid the usage of free variables in one way or another.

functional language: a referentially transparent (i.e. no side-effect) language whose central computational construct is function application. This is often considered too narrow a definition since key functional languages such as ML and Lisp in fact do allow side-effects, and therefore one uses the terms *pure* and *impure* to accentuate the absence and presence of side-effects respectively. A language is therefore functional when it adheres to the functional style of doing things. Features nowadays often associated with functional languages are: higher-order functions, closures, currying, lazyness, type inference, parametric polymorphism, (algebraic) non-mutable data structures, lack of side-effect operations, and pattern matching.

heterogeneous data structures: A data structure whose elements are not necessarily of the same type. For example, a "list of animals" that contains chickens and cows.

higher-order function: a function that can deal with another function as an argument or return value. A more narrow definition includes the fact the the function passed as a value must be a *closure*, which is necessary to make good use of higher-order functions (C has function values, but no closures).

Hindley-Milner: the type system / type inference mechanism underlying many modern functional languages. The basics of what type inference does are easy to understand: the type inferencer assigns types to every part of the *AST*, and whenever it misses type information (which is most often) it assigns a variable[1]. Next, the type inferencer *unifies* all type expressions according to the language constructs (for example, the left- and right-hand sides of an assignment), and binds most variables in the process: those variables that remain unbound at the end are polymorphic type variables. In practice it's slightly trickier though.

inclusion polymorphism: polymorphism based on subtyping hierarchies or, more generally, sets of types. An operation may work polymorphically on a set of types, though be implemented differently for each of them.

inheritance: see *subtyping*.

instantiation: creation of a new object (the *instance*) from a recipe (the class). The class says what objects of this type should look like, what properties it should have and how it should be initialised.

interface: the part of an object that is visible to the *client*; the protocol that the client uses to communicate with the

[1] Variables in this context are first class entities, i.e. data structures that can be bound just once, much like in ProLog.

object. The interface hides other parts of the object, namely the implementation which specifies its *behaviour*.

lexical scoping: scopes that are structured hierarchically based on their form as source code, used by most languages today. Also called *block structuring* (an excellent paper is [Mad86], also worthwhile are [Ten82] and [Han81]). An alternative is *dynamical scoping*, which organises nested scopes according to the dynamic function call chain of a program.

lvalue: an entity that can be both read (as a value) and written to (as a location), for example variables in Bla.

method: the OO version of a function. The difference with normal functions is that methods have a *receiver*, i.e. the object they belong to.

multiple inheritance: inheritance of more than one superclass; changes the inheritance hierarchy from a tree to a DAG. For example, to create a `clock radio` object we inherit from `clock` and `radio`. As well as benefits in conceptual modelling, there are some technical downsides.

name space: a set of identifiers that constitute the names of a certain level of declarations in a program. For example, in C++ global variables, the local variables of a function and the instance variables of a class all are different name spaces (in Bla only local variables can form a name space). The difference with a *scope* is that a scope denotes the borders of the code for which a certain name space is visible, but a scope may include more than one name-space, often hierarchically. And not every name space is necessarily part of a scope.

non-mutable data structures: data structures that are created and initialised in one go, and thereafter cannot be modified: they have to be read and reconstructed to make "changes". Lists in Bla and algebraic data structures in Haskell are good examples. The advantages and disadvantages are one and the same thing: they cannot be modified such that different parts of a program "profit" from the change transparently, as is the case with destructive updates. This "transparency" is at the same time the biggest problem: if various parts of the code use and modify a central data structure this can result in subtle and hairy bugs[2]. Non-mutable data structures protect you from this.

normal form: the point at which an expression cannot be reduced further. `3+4` would not be in normal form, but after reduction to `7` it would be. This of course depends on the set of reduction rules: in a system that doesn't know how to reduce `+`, `3+4` would in normal form.

no-variance: see *contravariance*.

Object Oriented: a system that allows you to construct a *subtyping* hierarchy, provides for *inclusion polymorphism* and *data-hiding* is pretty OO by my standards, but feel free to fill in your favourite definition here. See [Hat93].

open-world: or, the *open world assumption*: modules written in the language can type-checked and otherwise shown correctly separately. This is the case for most languages including Bla. The opposite is of course the *closed world assumption*: to compile an application the compiler needs to have overview of all code involved. This hampers separate compilation and is therefore unpopular, but it can aid with code optimisation and tough type-checking problems (that's what Eiffel uses it for).

orthogonal: independent. If two features are not orthogonal then they cannot be judged separately. Trying to design an orthogonal language means that if the language has a feature A with a certain semantics and a feature B with a certain semantics, then there's not a special set of rules about what is different when A and B are used together, i.e no exceptions. For example, *overloading* a method and making a method `virtual` in C++ are two unorthogonal features since when used together there are all sorts of exceptions.

overloading: syntactic sugar that allows the same identifier to be used for different entities by having the compiler decide which one it is from the compile-time known types in its context. Not to be confused with any form of *polymorphism*[3].

parametric polymorphism: the ability to define functions (and data structures) that handle values of any type, regardless of the form or shape of that value. `append()` is parametric polymorph: I can invent any new data structure, and I will always be able to use `append()` to work with lists of them.

parent: in OO terminology this often is the *superclass* or *delegate*, in Bla however it is the environment of the function enclosing the one that contains `self`[4].

pattern matching: used to select complex structures in memory and extract information from them. A pattern is an expression built from constructors of the datatypes one wishes to match: the shape of this data structure and the constants therein determine if a successful match on some value can be made, while variables denote the information to be extracted when the match succeeds.

polymorphism: the ability of one language construct to work with many types of values, and also future data structures. Well-known variants are *parametric polymorphism* and *inclusion polymorphism*. Cardelli [CW85] gives a solid overview.

predicate: a function with a boolean return value. In a more specific sense, also the basis of computation in logic languages.

prototyping: an alternative way of viewing OO, or programming in general. Whereas in class-based languages it is usual to build abstractions for everything you intend to use, in prototyping languages you simply construct the object straight away. If you want to do something with an elephant called "fred" you simply built a `fred` object with all the necessary properties, instead of making an elephant abstraction first. If

[2] Trust me, the author talks from experience.

[3] Except for some languages like Haskell which use this term also when this choice is made dynamically. It's a confusing world.

[4] This may sound confusing, but the fact that Bla misses an abstraction level makes this actually very logical. See also *self*.

you need a second elephant you simply clone `fred` (`fred` being the prototypical elephant), and add some stuff that may be different. If you need something else that shares many properties with `fred`, you can make `fred` into a *delegate* (or *parent*). Self is a prototyping language, read chapter 3.1.4 for some references.

receiver: see *method*.

reference semantics: if a certain type has reference semantics then values of that type are passed around by pointer. The opposite is *value semantics*, where the whole value is copied. For example, if I pass an object as argument to a function with reference semantics, both caller and receiver have access to the same object, but with value semantics a copy would have been made and changes made in the receiver would not affect the object in the caller.

referential transparency: if two expressions are syntactically the same, the values they compute will also be the same, regardless of the position in the source code. An expression like `f(2)*3` in a language like C may compute different values on different occasions, since `f()` may make use of global variables in computing its return value, i.e. the function can have a *side-effect*. If a language is referentially transparent, this cannot be the case. Referential transparency helps the reader in understanding the code.

reuse: the modern holy grail of software engeneering. The observation is that if writing bug-free code is so hard, then having to write it only once will help a lot. Modern reuse is targeted toward not having to *modify* old code, which object orientation claims to have the answer for, but which will prove to be slightly more involved.

scope: see *name space*.

self: in most OO languages this is the *receiver*, i.e. the object viewed from the method that works on it. In Bla however its the environment of a function, what other languages call *self* would in Bla be *parent*.

short-circuit: the method of evaluation in logical expressions (i.e. `and` and `or`) that guarantees that parts of the expression that are not necessary to compute the result will in fact not be computed.

side-effect: an extra action performed by an expression that does not contribute to the computation of the result, often modifying the *state* of a program. For example, the expression `a:=3+7` will result in the value `7`, but also modifies `a` in the process. Side-effects are considered a source of trouble by many because they increase the complexity of dependencies in a program.

single-dispatch: the way dispatch works in most OO languages: in a method call one "argument" (the *receiver*) is used to dynamically select which implementation of the method is executed. The alternative is called *multi methods* and takes *all* arguments into consideration: this has been less popular so far because conceptually people like methods that belong to one class (and not many), and also various issues in data-hiding and implementation become a lot more complicated.

slot: an element of an object, often another name for *instance variable*.

state: an element in a computation that can change value (while still being accessible in the same way). A global variable is a good example of state. The functional languages community has been busy for a while with this subject, read [HR93]. See *side-effect* and *referential transparency*.

structural equivalence: comparing two values for equality by comparing their shape and contents in memory. This is often used in functional and logic oriented languages, whereas in object-oriented languages two values (two objects) are the same if their pointers are the same: this is to preserve the idea of *object identity*.

subclass: a class whose interface is compatible with that of its superclass; objects of this class can be accessed transparently as having the superclass type.

subtyping (by name): creating new classes (or types) by making it a *subclass* of another class (the *superclass*). Subtyping is different from *inheritance* in that the language just checks that the new class you implement is interface-compatible with the superclass, whereas if you inherit something you also get the implementation of the interface as in the superclass, which can then be redefined.

superclass: the class to which a *subclass* tries to conform.

syntactic sugaring: syntax for constructs in a language that has been added simply to make the language friendlier to work with, since the construct could have been written using other (core) constructs instead.

tagged arguments: names of formal parameters that can be used as labels in a function call, making the meaning of such a call clearer, and less dependent on order.

templates: a macro-ish version of *parametric polymorphism* used in C++ [Str91].

term: a node in a tree-like data structure, consisting of the name of the node and a number of subtrees. Used in term-rewriting languages.

two-level: way to classify OO languages: two-level means the language splits the world into classes and objects, while *one-level* languages only have objects. There are languages with even more levels, for example Smalltalk has *meta-classes* (a class of classes). See [Hat93].

type class: the grouping of a set of *types* (i.e. not objects) into a class. Used in Haskell to specify sets of types on which certain operations are defined. See [HPW92] or [PJ93].

type inferencer: part of a compiler that can infer the types of all values / names in a program with little or no help from the programmer. See *Hindley-Milner*.

type parameter: a formal parameter to a function that needs to be passed a *type*. The handling of these types however, is strictly static, i.e. at compile-time.

unification: two-way *pattern-matching*: both expressions involved may bind variables. Requires variables to be first class. Used extensively in logic languages and also in type inferencers.

value semantics: see *reference semantics*.

virtual method: a method that is *dispatched dynamically*. This term is used in C++, where strangely enough methods are by default dispatched statically.

Appendix B

Bla Grammar

This should be quite complete[1].

```
source  = ( "module" seq( string ) ... ) seq( privacy fdecl )

fdecl   = ident fpart
fpart   = [ "[" ( ident "<" type ... "," ) "]" ] "(" adecls ")" [ ":" type ]
          [ ( "extends" | "implements" ) type ... ] ( "=" exp | block | "extern" )

privacy = [ "public" | "private" ]
decl    = privacy [ "const" | "class" | "object" ] ( fdecl | sdecl )

sdecl   = ident [ ":" type ] [ "=" exp ]
adecls  = [ pat [ "=" exp ] ... "," ]
pat     = all(pat) | "_"  | pat "/" pat | int [ ".." int ] ...
        | ident "(" [ pat ... "," ] ")" | ident [ ":" type ]

tupl(x) = "(" x "," x [ "," x ... ] ")"
all(x)  = "[" [ x [ "," x ... ] [ "|" x ] ] "]" | "<" [ x ... "," ] ">"
        | tupl(x) | int | real | string

type    = "int" | "bool" | "real" | "string" | "any" | "[" type "]"
        | tupl(type) | "<" type ">" | ident [ "[" ( type ... "," ) "]" ]

seq(x)  = "{" ( x ... ";" ) "}"        -- or Haskell Layout rule!
block   = [ "where" seq(decl) ]
          [ "do" seq(exp) [ "except" seq(seq) ] [ "close" seq(exp) ] ]

factor  = ( "self" | "parent" | "true" | "false" | "nil" |
            all(exp) | "(" exp ")" | block |
            [ "loop" ( ... sdecl ) ] [ "while" exp ] [ "next" exp ] block |
            "exit" [ ident ] |
            [ "++" | "--" ] ident |
            ( "-" | "not" ) factor )
          [ ( "." ident [ "++" | "--" ] |
              "(" [ ( [ ident ":" ] exp ) ... "," ] ")"
                [ "[" ( type ... "," ) "]" ] |
              "[" exp "]" |
              "<=>" pattern ) ]

-- these are in order of precedence, high ones first (usual associativity):
```

[1] Note I use my own grammar formalism with parametrisation and loops.

```
opexp   = factor | factor ( "*" | "/" ) factor | factor ( "+" | "-" ) factor
        | factor ( "=" | "<>" | ">" | "<" | ">=" | "<=" ) factor
        | factor "and" factor | factor "or" factor
exp     = opexp block
        | [ opexp "|" ] ( ( pat | exp ) "->" exp ... "|" ) [ "|" exp ]
        | factor ":=" exp
        | "return" exp | "raise" exp
        | "lambda" fpart | "`" ( fdecl | ident )

-- the lexical part:

ident   = "[A-Za-z][A-Za-z0-9_]*"
int     = "[0-9]+" | "\"[.]*\""
real    = "[0-9]+\.[0-9]+"
string  = "´[.]*´"

layout  = "[ \t\n]*" | "/*[.]*/" | "--[.]*"        -- need a preceding white-space
```

Bibliography

[BDMN] G. M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard. *Simula BEGIN*. Studentlitteratur et al.

[Ber92] E. Berger. FP + OOP = haskell. Technical Report TR-92-30, University of Texas at Austin, Austin, TX, 1992.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[Cam84] J. A. Campbell, editor. *Implementations of Prolog*. Series in Artificial Intelligence. Ellis Horwood, 1984.

[Car86] L. Cardelli. The amber machine. In G. Cousineau, P.-L. Curien, and B. Robinet, editors, *Combinators and Functional Programming Languages*, pages 48–70. Springer-Verlag, Berlin, DE, 1986. Lecture Notes in Computer Science 242.

[Car87] Luca Cardelli. Basic polymorhic typechecking. *Science of Computer Programming*, April 1987.

[CGL86] Nicholas Carriero, David Gelernter, and Jerry Leichter. Distributed data structures in Linda. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 236–242, St. Petersburg Beach, Florida, January 1986.

[CUCH91a] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation: An International Journal*, April 1991.

[CUCH91b] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation: An International Journal*, April 1991.

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF – a dynamically-typed object-oriented language based on prototypes. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, pages 49–70, October 1989.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Dea] Alan Dearle. Environments: A flexible binding mechanism to support system evolution. In *Proc. 22nd International Conference on Systems Sciences*.

[DMC92] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 201–217, October 1992.

[Dun85] M. R. Dunlavey. A progress report on D, a compiled language featuring continuations. *ACM SIGPLAN Notices*, 20(5):8–15, May 1985.

[FH88] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, Wokingham, Berkshire, 1988.

[GJL87a] D. Gelernter, S. Jagannathan, and T. London. Parallelism, persistence and meta-cleanliness in the symmetric Lisp interpreter. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, St. Paul, Minnesota, June 24–26, 1987*, Published as ACM SIGPLAN Notices, pages 274–282, 1987.

[GJL87b] David Gelernter, Suresh Jagannathan, and Thomas London. Environments as first class objects. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 98–110, Munich, Germany, January 1987.

[GR83] A. Goldberg and D. Robsen. *Smalltalk 80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gro92] The SELF Group. The SELF programmer's reference manual. Technical report, Sun Microsystems, Stanford University (ftp), 1992.

[Han81] David R. Hanson. Is block structure necessary? *Software— Practice and Experience*, 1981.

[Hat93] Bob Hathaway. comp.object faq. Technical report, Geodesic Systems, 1993.

[Hen80] Peter Henderson. *Functional Programming: Applications and Implementation*. Prentice-Hall, 1980.

[HPW92] Paul Hudak, Simon Peyton Jones, and Philip Wadler et al. Report on the programming language haskell, A non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.

[HR93] Paul Hudak and Dan Rabin. State in functional languages: An annotated bibliography. Technical report, 1993.

[Inc92] Commodore Amiga Inc. *AMIGA ROM Kernal Reference Manual: Libraries (3rd ed.)*. Addison Wesley, 1992.

[jav95] The java virtual machine. Technical report, Sun Microsystems Laboratories, 1995.

[KMMN87] B. B. Kristensen, O. L. Madsen, B. Möller-Pedersen, and K. Nygaard. The BETA programming language. In *Research Directions in Object-Oriented Programming*. The MIT Press, 1987.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, pages 214–223, November 1986.

[LSU87] Henry Lieberman, Lynn Stein, and David Ungar. Treaty of Orlando. *ACM SIGPLAN Notices*, 23(5):43–44, May 1987.

[Mad86] Ole Lehrmann Madsen. Block structure and object oriented languages. *ACM SIGPLAN Notices*, October 1986.

[Man93] Steve Mann. The Beta programming language: An OO language with Simula roots. *Dr. Dobb's Journal*, October 1993.

[McC65] J. McCarthy et al. LISP 1.5 programmer's manual. Technical report, The MIT Press, 1965.

[McC78] John McCarthy. History of LISP. *ACM SIGPLAN Notices*, 13(8), August 1978.

[Mey92] B. Meyer. *Eiffel, the Language*. Prentice Hall, 1992.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[O'D85] Michael J. O'Donnell. *Equational logic as a programming language*. MIT Press series in the foundations of computing, 1985.

[Oor93] Wouter van Oortmerssen. Amiga E v3.2a reference manual. Technical report, ftp from ftp.wustl.edu (/pub/aminet/dev/e/), 1993.

[PJ93] J. Peterson and M. Jones. Implementing type classes. In *227–36*, 1993.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, pages 146–161, November 1991.

[Sar94] J. Sargeant. Uniting functional and object-oriented programming. Technical report, Univ. of Manchester, 1994.

[Sha94] David L. Shang. Covariant specification. *ACM SIGPLAN Notices*, December 1994.

[Sha95] David L. Shang. Covariant deep subtyping reconsidered. *ACM SIGPLAN Notices*, May 1995.

[SO94] D. Stoutamire S. Omohundro. The sather 1.0 specification. Technical report, ICSI Berkeley, 1994.

[Str91] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, 1991.

[SU] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for Self. Technical report, Sun Microsystems Laboratories.

[Ten82] R. D. Tennent. Two examples of block structuring. *Software— Practice and Experience*, 1982.

[Ung91] Smith Ungar. Self: The power of simplicity. *Lisp and Symbolic Computation: An International Journal*, April 1991.

[WC85] T. Taft et al. W. Carlson. Annotated ada reference manual v6.0 (iso 8652:1987). Technical report, U.S. Government, 1985.

[WC91] J. Rees W. Clinger. Revised(4) report on the algorithmic language scheme. *ACM Lisp Pointers*, July 1991.