# Ellie - A General, Fine-Grained, First Class Object Based Language

Birger Andersen

Department of Computer Science

University of Copenhagen

Universitetsparken 1

2100 Copenhagen, Denmark

E-mail: birger@diku.dk, Fax: +45 31 39 02 21, Phone: +45 31 39 64 66

Research Paper [1]

October 1991

1. To appear in *Journal of Object-Oriented Programming*.

# Abstract

This paper presents the general-purpose programming language Ellie. Ellie supports fine-grained objects and parallelism. As something particular, classes, types, blocks, and methods are all abstracted by first class objects.

The main goal of Ellie has been to obtain a very flexible, machine independent, parallel programming language. Ellie tries to meet these goals by extensive usage of few selected programming language concepts combined with compile-time analysis in order to adapt programs for efficient execution on actual hardware. Ellie runs on a parallel transputer network. [2] This paper concentrates on the generality of the language that makes it extraordinary flexible and extensible.

# Keywords

Extensible languages, First class objects, General-purpose languages, Language concepts, Fine-grained programming, Object-oriented languages

# Acknowledgments

I would like to express my thanks to Eric Jul and Niels Christian Juul, both of the Department of Computer Science in Copenhagen. Their comments on the Ellie language design and on this paper have been very helpful. I also thank Bjarne Steensgaard for his helpful comments and Bjarne Hansen for the discussions we have had on the language design and for implementation of most of the Ellie kernel.

2. The transputer is a trademark of the Inmos Group of Companies.

# Table of Contents

Section 1
Introduction . . . . . 5

Section 2
The Ellie Object Abstraction . . . . . 9

Section 3
Genericity and Polymorphism . . . . . 13

Section 4
Delegation and Multiple Inheritance . . . . . 17

Section 5
Fine-Grained Programming . . . . . 19

Section 6
Extension Examples . . . . . 21

Section 7
Status and Summary . . . . . 23

References . . . . . 25

Index . . . . . 29

# List of Figures

Figure 2.1: Point Object . . . . . 10
Figure 3.1: Generic Stack Object . . . . . 13
Figure 3.2: Polymorphic Square Function . . . . . 14
Figure 4.1: Specialization and Usage of Point Object . . . . . 18
Figure 5.1: Fibonacci Function . . . . . 19
Figure 5.2: Fine-Grained Fibonacci Objects . . . . . 20
Figure 6.1: Point Object Redefinition . . . . . 21
Figure 6.2: For-Loop Control Object . . . . . 22

# Section 1
# Introduction

Ellie is a fine-grained object-oriented programming language. It has been designed to support high level fine-grained parallel programming in a flexible and efficient way on distributed memory, MIMD parallel computers.

A goal of the language has been to achieve extraordinary language flexibility in order to be generally usable. This has been obtained by allowing definitions of new type and control structure abstractions and by having a non-strict conformity type system. Ellie is intended to be an all-purpose programming language that can meet the efficiency of languages like Fortran and C and the flexibility and expressiveness of languages like Lisp and Smalltalk.

This paper concentrates on the generality of the language that makes it flexible and extensible for specialized usage. The fine-grained programming [3] ideas are only briefly concerned in this paper. More details on Ellie and the research project is available in [Andersen 91a, 91b, 91c; Hansen 91].

One might complain that we have already seen enough object-oriented languages and that we should now concentrate our efforts on how to use these languages and how to implement them more efficiently. Of cause we must realize these problems and deal with them. Still it is possible to improve existing languages by applying higher abstraction facilities and new stronger concepts, so the design of new languages is still justified. The motivation for designing Ellie, besides the need for a parallel language more machine independent than others, has been the following observations:

1) Integration of applications from different fields (bookkeeping, simulations, text editing, real-time systems, etc.) is very difficult because traditionally different programming languages are used in each area (e.g., Cobol, Fortran, Pascal, C, Simula, Smalltalk, etc.). Awkward conversion of data structures and no reuse of code between the integrated applications are results hereof.

2) Some users/applications need high efficiency, others high abstraction. Therefore switching between, for example, Fortran and Smalltalk takes place, and we are back to the problem from 1) above.

Solutions to these problems applied to Ellie is from my point of view the following:

---

3. Fine-grained programming means *fine-grained parallelism* and *fine-grained objects*. In Ellie this means that the logically finest operations, e.g., multiplications, can be executed as parallel processes, and that objects can have as less quantity of state information as to represent characters, etc.

re 1) Classes, types, control structures, blocks, and methods must all be chosen to be object abstractions in order to make the language particularly extensible. This idea is considered in details in this paper.

re 2) Numeric calculations must be compiled to match the efficiency of Fortran and C when high abstraction facilities are not needed. This demand is considered in [Andersen 91c].

Machine independence has been realized by introducing fine-grained programming. At compile-time processes may be combined to larger less fine-grained processes and objects to larger less fine-grained objects. This is called *compile-time grain-size adaption*. When having a parallel machine that can execute fine-grained programs efficiently no adaption is done. Otherwise, more or less adaption has to be done. No other high-level object-oriented language, known by this author, supports fine-grained programming as a common idea.

Ellie is semantically and syntactically a rather simple language but it relies on some sophisticated ideas that all together constitute a very general and powerful language.

- Classes, types, blocks, methods, and regular objects are all abstracted by the fundamental and single abstraction mechanism called an *Ellie object*. Ellie objects are computing entities of varying sizes, e.g., fine-grained. The nature of objects is similar to that in Smalltalk [Goldberg 89], i.e., all data items are objects on which methods may be applied in a uniform way. All abstractions are *first class*, e.g., they may be passed as parameters (see [Cardelli 85; Danforth 88; Donahue 85]).

- Functional and operational methods are existing at the same time like in Emerald [Black 87; Hutchinson 87; Raj 91]. Functions are without side-effects and operations may have side-effects. Hence, an Ellie object is said to have associated either a *functional process* (may be the empty process) or an *operational process*. The reasons for this distinction are improved understanding of parallel programming and the ability to define *immutable* objects where only functions can access the state, i.e., the state will be constant. The strength of immutable objects is that they can be copied in any number on a distributed memory computer.

- Interfaces that relax encapsulation can be defined in order to make locally defined attributes of objects accessible from outside by *remote addressing*. Attributes are simple objects being able to transfer and alter their values, etc. This idea is similar to Self [Ungar 87]. *Label* and *constant* attributes are constantly bound to Ellie objects (a label object can not transfer its value), whereas *variable* attributes dynamically can be bound to Ellie objects.

- Block structured programming is supported by Ellie like, e.g., Beta [Kristensen 87; Madsen 87], Simula [Birtwistle 79], and Emerald, but in Ellie blocks are object abstractions (closures) that locally may define attributes and objects. Pascal/Algol scope rules are associated with attributes, the exception being that attributes are known inside the whole block in which they are defined. This promote recursive definitions.
- Conformity type checking is exploited to type check programs statically for efficiency reasons. The type system is analogous to that implemented in Emerald [Black 87] and Guide [Krakowiak 90]. Type checks are also available to the programmer at run-time in order to widen views of objects. Run-time type errors are only possible when widening views.
- Genericity and polymorphism [4] are available. Efficient code can often be guaranteed, since the compiler is able to compile optimally when the implementations of parameter objects are known at compile-time. Otherwise, the flexibility of conformity is utilized instead, and the compiler generates less efficient tag-checking code.
- Delegation and multiple inheritance is supported by an Ellie mechanism called *interface inclusion*. Delegation routes may dynamically be changed. Because requests may be delegated to a number of objects and the mechanism takes care of name collisions, multiple inheritance abstraction is also available. The compiler decides whether to use delegation or inheritance at the code level.
- Fine-grained parallelism and synchronization is expressed explicitly by means of *bounded* and *unbounded* (remote) method calls. In the unbounded case, the result is returned as a *future result object* which is an abstract object with an undefined state. This approach is also known from ABCL/1 [Yonezawa 86]. An Ellie mechanism called *dynamic interface*, which is about as powerful as Path Expressions [Campbell 74], is used for synchronization at all levels of parallelism. A dynamic interface may delay processes trying access.

The remaining part of this paper presents the Ellie object abstraction concept, generic/polymorphic objects, and the interface inclusion mechanism, in more details. A fine-grained programming example is also presented followed by extension demonstrations.

4. Genericity, polymorphism, and other language concepts are considered in details in [Watt 90].

# Section 2
# The Ellie Object Abstraction

The *Ellie object* is the single abstraction mechanism for the notion of classes, types, blocks, methods, and regular objects in Ellie. Anyhow, the Ellie object [5] is semantically and syntactically a surprising simple entity that can be used in three ways at the same time:

1. As a model or pattern for the creation and initialization of new objects to be based on. This is called the *prototype* attitude in Ellie terms, although the state is not implicitly delivered to new objects from their prototypes nor shared (might be very inefficient in a distributed environment).

2. As an abstract type specification for the definition of named attributes to be based on. This is called the *qualifier* attitude in Ellie terms, because named attributes by their definitions are restricted to refer to the qualifier object or objects *conforming* (see below) to the qualifier only. More than one object may implement the same abstract type.

3. As a regular object receiving messages in which case an interface defines the messages in terms of named attributes that the object can act on. This is called the *object* attitude.

The rules for conformity will not be elaborated in details in this paper. Briefly, an object *A* conforms to an object *B*, only if *A* at least defines the same attributes as *B* in its interface and the objects referred to by the identical attributes and the formal input parameters (see below) also conform in the same direction. The formal output parameters must conform in the opposite direction. Operations never conform to functions. See the exact definition in [Andersen 91a].

Instantiations of classes, invocations of methods, and executions of blocks are all expressed by the prototype attitude in which case an actual input parameter [6] is passed for *initializing* the new object. This is denoted the *activation* of a prototype yielding a new object. An activation can also be done *remotely* by sending a message consisting of a named attribute and an actual input parameter to an object. The new object will exist only temporary in case of a block or method abstraction. After being *initialized*, i.e., the execution of the associated process has finished, the new object will return an actual output parameter that may be itself in which case it can be used as prototype, qualifier, or regular object.

---

5. The generalized Ellie object will from now on just be called an *object*.

6. Parameters are passed by object references but the compiler may decide other strategies preserving this semantics.

An object is initially defined by means of a *constructor*. A constructor defines an *uninitialized* object without known state that exists only virtually; it can only be used as a prototype or qualifier.

Consider a constructor that defines a point object:

```
[ X, Y :- real ]
```

The value of such a constructor is a reference to the object being defined.

The attributes `X` and `Y` are defined to be local variables [7] (the `:-` operator) qualified by the object referred to by the attribute `real` defined somewhere else. `X` and `Y` are said to be *qualified* by the *qualifier expression* constituted by `real`. The qualifier expression is not evaluated at compile-time, only the resulting abstract type is deduced.

In order to hold the reference to the point object somewhere, the constructor must be a part of an expression. Below, the constant attribute `point` is defined (the `:` operator) and qualified by the point object constructor:

```
point : [ X, Y :- real ]
```

In a definition, the attribute is not only qualified by the qualifier expression, the attribute is also implicitly initialized to the actual value of the expression, i.e., a reference to the point object. Therefore, `X` and `Y` will both initially refer to the same object as `real` refers to. [8]

```
point : [ <: init_val : point_value :> self;          $ (1)
   export X, Y, move_to, rel_move;                    $ (2)

   point_value : [ X, Y : real ];                     $ (3)
   X :- init_val.X; Y :- init_val.Y;                  $ (4)

   move_to : [ <: p : point :> none;                  $ (5)
      X <- p.X; Y <- p.Y                              $ (6)
   ];

   rel_move : [ <: p : point :> none;
      X <- X + p.X; Y <- Y + p.Y                      $ (7)
   ]
];
```

Figure 2.1: Point Object

A simple expression like `point [ X : 1.5; Y : 2.5 ] []`, using a more likely definition of a point object in Figure 2.1, now shows us some of the real strength of the Ellie object since the expression will activate an object with the object bound to the attribute `point` as prototype. The activated constructor in the expression will be the actual input parameter to the object being created

7. Attributes in Ellie are placeholders for references to objects but the compiler may replace references with inserted objects only if the semantics can be preserved.

8. In general, the actual value of a qualifier expression may be an object different from the qualifier that just conforms to the qualifier.

(activations from right to left). Actually, the activations are caused only by the existence of actual input parameters. [9] The formal input parameter `init_val`, being a constant attribute (1), will be bound to the actual input parameter that must conform to the qualifier. The formal output parameter `self` (1), which is an implicitly defined constant attribute referring to the object itself, will be returned as an actual output parameter, being the value of the expression.

The point object locally defines the attributes `init_val`, `point_value`, `X`, `Y`, `move_to`, and `rel_move`. These attributes are encapsulated but by defining an interface (2), selected attributes, e.g., `move_to`, may be exported to users that have references to point objects.

If `P` and `P2` both refer to point objects then the expression `P.move_to P2` will cause the `P` point to be moved to the location of the `P2` point, i.e., the prototype bound to `move_to` (a method abstraction) is remotely activated by sending the message `move_to` with the `P2` reference to the `P` point. The value `none` (5) of the expression refers to an uninitialized predefined object that conforms to any object. A message like `P.init_val` is invalid because the attribute `init_val` is encapsulated by the point object, i.e., not defined to be accessible.

An assignment (6, 7) rebinds a variable attribute to a new object being the value of the expression to the right of the <− operator. This operator is an attribute defined and exported by a special object representing the variable attribute. In the last part of this paper we will study how an attribute object may be implemented.

Attributes may be denoted by alphabetical names or operator names. In case of activations of operator names, i.e., activation of the prototypes bound to them, the periods to the left of the operator names may be omitted. Examples are activation of <− and + (7). When periods are omitted, precedence levels are associated with operator names to avoid excessive numbers of parenthesis in complicated expressions. Operator names may be overloaded and new ones may be defined, hereby reusing the associated precedence levels.

9. The actual parameter `[]` refers to the simplest possible object.

# Section 3
# Genericity and Polymorphism

The generic and polymorphic language concepts are nowadays a must for an object-oriented programming language. Genericity, usually by means of generic classes, known as *class genericity*, allows us to define general classes that can be specialized by providing class parameters. Polymorphism, usually by means of polymorphic functions (or methods), allows actual parameters of varying types. This is known as *parameter polymorphism* in this section. Class genericity and parameter polymorphism in Ellie will be considered using examples.

In Ellie, classes and methods are abstracted by objects. This means that genericity and polymorphism is achieved without further extensions of the Ellie object concept. In both cases actual parameters conforming to qualifier expressions of the formal parameters can be provided in activations yielding various kinds of specializations. Note, that not only class and method abstractions may be specialized in this way, also abstract types and blocks may be specialized increasing code reuse to a very high level.

*Genericity*

In order to study specializations of class abstractions by genericity, we will now consider the generic stack in Figure 3.1 abstracted by an object referred to by the constant attribute `stack`. The `#` operator in this example is the indexing operator, i.e., an attribute exported by vector objects.

```
stack : [ <: par : generic_par :> self;
   export push, pop, reset;

   generic_par : [ length : integer; element : any ];
   table : vector [ length : par.length;
      of : [ elem :- par.element ] [] ] [];
   top :- -1;    $ -1 is an initialized integer object

   push : [ <: new : par.element :> none;
      top <- top + 1; table # top <- new
   ];

   pop : [ <: any :> old :- par.element;
      old <- table # top; top <- top - 1
   ];

   reset : [ <: any :> none;
      top <- -1
   ]
];
```

Figure 3.1: Generic Stack Object

A specialized stack with 100 placeholders that can push and pop objects conforming to, e.g., the point object, may be activated by evaluating the expression `stack [ length : 100; element : point ] []`. By the definition of the stack in Figure 3.1 we can see that the qualifier expressions (and initialization values) of the attributes `table`, `push`, and `pop` will be specialized by the actual input parameter, hereby giving us a specialized stack. A point stack generator can be implemented by writing the following definition:

```
point_stack : [ <: any
   :> stack [ length : 100; element : point ] []];
```

The formal input parameter defined by `<: any` [10] may be omitted. [11] An activation of `point_stack` will return a new instance of the specialized stack.

*Polymorphism*

Parameter polymorphism is also directly supported by Ellie because an actual parameter supplied in an activation only needs to conform to the qualifier expression of the formal input parameter. Also, the value returned by the activated object needs only to conform to the qualifier expression of the formal output parameter. As an example, consider the definition of a square function in Figure 3.2.

```
square : [ <: actual : qualifying                    $ (1)
   :> result :- actual;                              $ (2)

   qualifying : [                                    $ (3)
      export *;
      * : [ <: any :> none ]                         $ (4)
   ];

   result <- actual * actual                         $ (5)
];
```

Figure 3.2: Polymorphic Square Function

In Figure 3.2, the formal input parameter `actual` is qualified by the qualifier bound to the attribute `qualifying` (1, 3), whereas the formal output parameter `result` is qualified by the input parameter bound to `actual` (2). The result hereof is that when activating `square`, the qualifier of the formal output parameter will be equal to the compile-time deduced qualifier from the actual input parameter. The qualifier of `qualifying` (3) is defined abstractly only in the sense that the locally defined multiplication has no process defined. The process is implemented by multiplication functions of actual parameters provided in activations of `square`. Actual parameters must implement a multiplication function conforming to the

10. The attribute `any` refers to an uninitialized object that all other objects conform to.

11. When only a qualifier expression is present the actual parameter must conform to the qualifier but the value of the expression is ignored.

abstract one (4). This is necessary because the prototype bound to the attribute * is activated in the action part (5).

*First Class Objects*

Since the Ellie object is an abstraction for the notion of classes, types, blocks, methods, and regular objects, actual parameters may be any of these abstractions. This means that class, type, block, method, and regular object abstractions all are *first class objects*. This is a realization of types as values [Donahue 85]. Hence, Ellie offers superior second order programming facilities that only the programmer's understanding will obstruct. Future research will be necessary in order to realize the advantages and disadvantages of this kind of unconstrained second order programming, i.e., programming with types and functions, etc., as parameters.

# Section 4
# Delegation and Multiple Inheritance

The *object interface* defines which locally defined attributes of an object that are to be accessible from outside. These unencapsulated attributes will be inaccessible during the initialization of an object.

The special Ellie *interface inclusion* mechanism is a part of the interface definition. An inclusion specifies which remote attributes, i.e., attributes in the interface of another object, to make accessible for users of the object itself. An inclusion does not specify whether to make the names accessible by delegation or inheritance. The compiler decides which code to generate. Consider, as an example, the following interface definition of an object *X*:

```
export local_1, (local_2), (local_3).(remote_1, remote_2);
```

This interface makes a locally defined attribute `local_1` accessible by users of *X*. It also makes accessible all attributes which are accessible from the object bound to the locally defined attribute `local_2` and the two attributes `remote_1` and `remote_2` accessible from the object bound to the locally defined attribute `local_3`. The attributes `local_2` and `local_3` will be inaccessible by users of *X*, i.e., they continues to be encapsulated.

The delegation routes may dynamically be changed. This might, in case of the object *X*, be done by rebinding the variable attribute `local_2` to another object that conforms to the qualifier. If we think in terms of inheritance, this will cause the inherited attributes (and the objects bound to them) to be substituted with another set of identical attributes (and new objects bound to these) inherited from another object. This may be called *object-based inheritance* [Hailpern 87].

The attribute `self` is implicitly defined locally by any object to refer to the object itself. This diverges from most class/inheritance based languages, e.g., Smalltalk. Accordingly, actions must be taken explicitly in Ellie in order to pass self references via delegation routes or upwards to super objects.

If an attribute is included in an interface more than once, only the first version will be accessible. This is a simple solution to the well-known name collision problem. By explicitly mentioning each remote attribute, the correct version can always be selected by the programmer. Run-time selection can be implemented by writing methods to make the decisions explicitly.

We will now investigate an example that shows the differences and equalities between delegation and inheritance in Ellie. Figure 4.1 implements a specialization (a subpoint) of the point object of Figure 2.1 and a circle object. This shows that a subpoint object uses a point object to represent the point and adds an attribute `distance` referring to a method abstraction that calculates the distance between two points. Hence, a subpoint object delegates accesses to the attributes `X`, `Y`,

```
subpoint : [ <: init_val : point_value :> self;
   export (p).(X, Y, move_to, rel_move), distance;

   point_value : [ X, Y : real ];
   p : point init_val;

   distance : [ <: p2 : point
      :> sqroot  (square p.X - p2.X) +
                 (square p.Y - p2.Y)
   ]
];

circle : [ <: init_val : circle_value :> self;
   export (C).(move_to, rel_move), C.(->), R, area;

   circle_value : [ C : point; R : real ];
   C :- init_val.C;
   R :- init_val.R;

   area : [ <: any :> PI * square R ]
];
```

Figure 4.1: Specialization and Usage of Point Object

`move_to`, and `rel_move` to a point object in agreement with the include part of the interface definition. A user of a subpoint object may consider this as inheritance or a stand-alone version of a point object, whereas a compiler may generate code that actually implements inheritance by including the properties of the point object in the subpoint object because the attribute `p` is a constant, i.e., the super object will always be the same. Furthermore, the definition of the subpoint object shows that the value of `p` will never be transferred.

The circle object given by Figure 4.1 delegates some accesses to a point object referred to by the attribute `C` representing the center of a circle. In this case, a compiler can not generate code that implements inheritance since `C` is a variable. Furthermore, the value of `C` can be transferred to users of circle objects (the `C.(->)` definition in the interface), so a point object can not be included as a physical part of a circle object. A user can not abstract inheritance either since the value of `C` (a point object) can be transferred to the user.

Finally, note that if the attribute `p` of subpoint objects was defined as a variable then object inclusion by a compiler would be impossible but a user would continue abstracting inheritance. By encapsulated alternation of `p`, dynamic inheritance could be abstracted.

# Section 5
# Fine-Grained Programming

All items in Ellie are Ellie objects that may have associated processes. For each object, the grain-size of the object and its process will be somewhere in the interval between fine-grained and coarse-grained. Examples of other languages with varying grain-sizes are Cantor [Athas 87] and MELD [Kaiser 89]. Cantor is intended for machine programming and therefore, the level of abstraction is much lower than in Ellie. MELD's level of abstraction comes somewhat closer to Ellie but it has a complicated syntax for expressing fine-grained parallelism. Also the Actor languages [Agha 86, 87] may be implemented with fine-grained parallelism because computing entities are applicative. The level of abstraction is lower than in Ellie.

An object with a *bounded process* returns its output parameter when the process has terminated. An object with an *unbounded process* returns its actual output parameter as a *future result object* in an undefined state when activated. The state of a future result object will be defined when the process has terminated. A future result object delays attempts to access local attributes until the state has been defined [12] but the reference to it may be passed by messages without delays.

As a fine-grained programming example, consider the function abstraction in Figure 5.1 which when activated makes its process compute the *n*th Fibonacci number in parallel by recursively activating itself, hereby generating a number of unbounded processes without side-effects.

```
Fibonacci : [ future function <: N : integer
   :> fib :- integer;

   option [
      if : N > 1;
      then : [
         fib <- Fibonacci (N - 1) + Fibonacci (N - 2)
      ];
      else : [
         fib <- N
      ]
   ]
];
```

Figure 5.1: Fibonacci Function

When activated, e.g., by evaluating the expression `Fibonacci 5`, the object defined in Figure 5.1 will spawn a binary tree of 15 objects with unbounded

12. The delay will be of the process trying to access an attribute by sending a message.

processes. The attribute `option` refers to a control structure object that activates its actual input parameter and hereby binds the attribute `if` to the value of `N > 1`. Depending on the value of `if`, either the attribute `then` or the attribute `else` will be activated. The operators >, −, and + are bound to fine-grained method abstractions with fine-grained processes that are also activated.

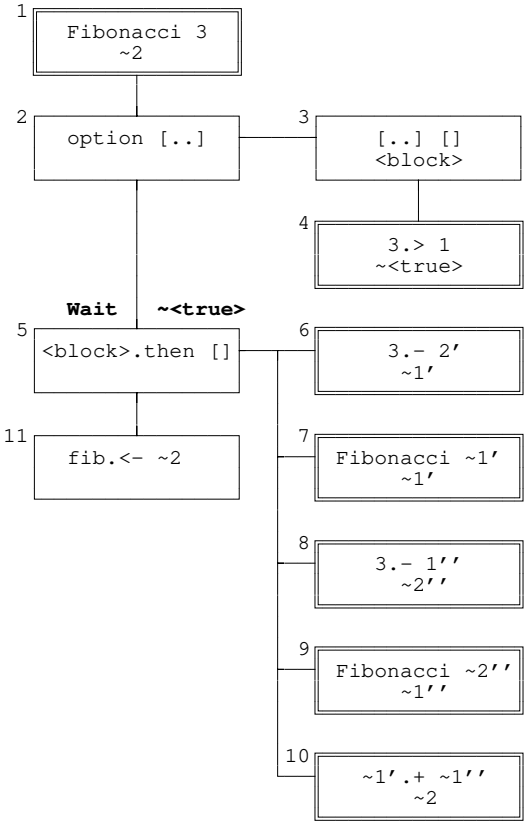


Figure 5.2: Fine-Grained Fibonacci Objects

In Figure 5.2 we examine the level of the object structure generated before the recursive activations when the prototype is activated by the expression `Fibonacci 3`. Each box contains in the top the expression activating the object represented by the box. In the bottom of the boxes the returned value, if applicable, is shown. The tilde (~) means a future result object, whereas the mark (') is used to distinguish different objects with the same state. The double lined boxes represent objects with unbounded processes, whereas the single lined boxes represent objects with bounded processes. No. 5 will not be activated until the future result object ~<true> has been defined to the concrete object <true>. Objects generated by processes of object no. 4, 6, 7, 8, 9, and 10 have been left out for convenience.

An important idea of Ellie is compile-time combination of fine-grained processes to larger processes and fine-grained objects to larger objects when the target computer for a program compilation does not efficiently support the fine-grained programming supported by Ellie.

# Section 6
# Extension Examples

The unlimited extension abilities available in Ellie are mainly based on the superior second order programming facilities and/or the idea of attribute objects.

Three different attribute prototypes are available: Label, Constant, and Variable attribute objects. Each time a named attribute is defined an attribute object to represent the attribute is implicitly defined. A variable exports the attributes <- and -> [13] referring to functions for altering and transforming the value. [14] A constant exports ->, whereas a label exports no locally defined attributes. Furthermore, all attribute objects delegate messages to their value objects. Let us again consider the simple point prototype:

```
point : [ X, Y :- real ]
```

If we now want to represent the state of a point object by the angle and the length of a vector, we can redefine the simple point prototype as shown in Figure 6.1.

```
point : [
   export X, Y;

   angle, length :- real;

   X :: [
      export <-, ->;
      <- :: [ <: newX : real :> newX;
         $ Calculation of new values for angle and length
      ];
      -> :: [ <: none :> currentX :- real;
         $ Calculation of current value for X
         $ from angle and length
      ]
   ];

   Y :: [
      $ Analogously defined as for X
   ]
];
```

Figure 6.1: Point Object Redefinition

The reader might complain that the state of an object always should be encapsulated by the object. Actually, this example demonstrates that variables are encapsulated (the :: operator defines labels). Otherwise, it would be impossible

13. The value of an attribute A is transferred by just mentioning the attribute because an expression A is implicitly transformed to an extended expression A.-> [].

14. Other attributes are also exported by attribute objects for conformity checking, etc.

to substitute exported variable attributes by label attributes bound to objects simulating variable attribute objects. An attribute object insure that the value, i.e., the state of the attribute object, is always encapsulated. Therefore, states will always be encapsulated in Ellie. Non-encapsulated references can not be exported at all and will exist only as temporary results of expression evaluations.

```
for : [ operation <: block :- for_block;
   for_block : [ times : integer; do : any ];
   i :- 0;

   block <- block []; $ Activation of the block
   loop [
      while : [ :> block.times >= i <- i + 1 ];
      do : block.do
   ]
];
```

Figure 6.2: For-Loop Control Object

Another interesting example to be considered is the definition of a new control abstraction in Figure 6.2 based on the reality that blocks (closures) are also abstracted by objects. This for-loop control abstraction requires that the block provided as actual input parameter in an activation of `for` defines and exports two constant attributes `times` and `do`. The constant `times` must refer to an integer object describing the number of times that `do` shall be activated. The constant `do`[15] must refer to a block abstraction. Let us consider the following example:

```
for [
   times : 5;
   do : [ write "hello world\n" ]
];
```

Evaluation of this expression causes the string `hello world` to be written five times on separate lines.

Two control abstractions, `loop` and `select`, implementing very general versions of loop and selective control structures are defined in a basic Ellie library that also defines class/type abstractions like `Boolean`, `char`, `integer`, `real`, `vector`, etc. New specialized libraries may then be based on these definitions in the basic library which is always included as a part of an Ellie program. A new library could, for example, define control and class abstractions that includes the logic programming paradigm or a list class abstraction that would make Ellie as powerful as Lisp, etc.

15. Actually, `do` needs not to be a constant. It may also be a label or variable. In the latter case, the body of the loop can be replaced by a new body while the loop is being executed.

# Section 7
# Status and Summary

This last section gives a project status and a summary.

*Status*

An Ellie compiler generating fine-grained virtual code (a byte code) has been implemented and is currently running on Personal Computers and various Unix workstations. The compiler is written in C (about 6000 lines). All semantic checks has been implemented, including conformity type checks. The object constructor fragmented fine-grained virtual code is only slightly optimized. More work has to be done in order to realize the generation of more efficient fine-grained code. Grain-size adaption also remains to be implemented.

A kernel including an interpreter has also been implemented. The kernel runs on the same computers as the compiler and is also written in C. Run-time conformity type checks, object comparisons, and parts of basic library are currently not implemented. Some kernel optimizations also remain to be done, including the implementation of a memory garbage collector. Later on, the interpreter will be replaced by machine code generated by an additional compiler module in order to observe efficiency in numerical calculations, etc.

The kernel has been moved to a parallel network consisting of 16 T800 transputers by adding a message router and a naive load balance mechanism to the kernel. A more powerful load balance mechanism shall later on decide when to move objects in order to optimize the object distribution dynamically. Although this parallel implementation already has shown interesting results in terms of speedup measurements, not much can be concluded at the moment. The speedup factor 11 has been observed for a `fibonacci 12` activation (Figure 5.1) but this is without compile-time adaption. Many interesting results can be expected later from this ongoing research project.

*Summary*

Ellie is for several reasons a promising language. The concept of first class objects abstracting classes, types, blocks, and methods is powerful and supports various kinds of genericity and polymorphism that the future hopefully can show the applications of. Ellie also seems to be the language known today that most easily can express fine-grained objects and parallelism at a high abstraction level. Future work must show if the ideas in Ellie also will succeed in practice when more complicated and meaningful Ellie applications are available within a few months.

# References


[Agha 86] Gul Agha. An overview of Actor langauges. *ACM SIGPLAN Notices*, 21(10):13-22, October 1986.

[Agha 87] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 49-74, The MIT Press, 1987.

[Andersen 91a] Birger Andersen. *Ellie Language Definition Report*. PhD thesis (partial), DIKU Report no. 91/3, Department of Computer Science, University of Copenhagen, Denmark, June 1991 (2nd ed.). Also *ACM SIGPLAN Notices*, 25(11):45-64, November 1990 (1st ed.).

[Andersen 91b] Birger Andersen. *Fine-Grained Parallelism in Ellie*. PhD thesis (partial), DIKU Report no. 91/4, Department of Computer Science, University of Copenhagen, Denmark, June 1991. Also to appear in *Journal of Object-Oriented Programming*.

[Andersen 91c] Birger Andersen. *Grain-Size Adaption in the Fine-Grained Object-Oriented Language Ellie*. PhD thesis (partial), DIKU Report no. 91/5, Department of Computer Science, University of Copenhagen, Denmark, 1991. In preparation.

[Athas 87] William C. Athas. *Fine Grain Concurrent Computations*. PhD thesis, Technical Report 5242, Department of Computer Science, California Institute of Technology, Pasadena, California, May 1987.

[Birtwistle 79] Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden, 1979.

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65-76, January 1987. Also Technical Report 86-02-04, Department of Computer Science, University of Washington.

[Campbell 74] R.H. Campbell and A.N. Habermann. The specification of process synchronization by path expressions in operating systems. In *Operating Systems Proceedings*, April 1974. Lecture Notes in Computer Science, (16):89-102, Springer-Verlag, 1974.

[Cardelli 85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471-522, December 1985.

[Danforth 88] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29-72, March 1988.

[Donahue 85] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426-445, July 1985.

[Goldberg 89] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley Publishing Company, Massachusetts, 1989.

[Hailpern 87] Brent Hailpern and Van Nguyen. A model for object-based inheritance. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 147-164, The MIT Press, 1987.

[Hansen 91] Bjarne Hansen. *Object Activation in a Transputer Implementation of the Ellie Language*. MSc thesis, DIKU Report no. 91/6, Department of Computer Science, University of Copenhagen, Denmark, March 1991.

[Hutchinson 87] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, Technical Report 87-01-01, Department of Computer Science, University of Washington, Seattle, Washington, January 1987.

[Kaiser 89] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush, and Shyhtsun Felix Wu. MELDing multiple granularities of parallelism. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 147-166, Cambridge University Press, July 1989.

[Krakowiak 90] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, and X. Rousset de Pina. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming*, 3(3):11-22, October 1990.

[Kristensen 87] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The Beta programming language. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7-48, The MIT Press, 1987.

[Madsen 87] Ole Lehrmann Madsen. Block-structure and object-oriented Languages. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 113-128, The MIT Press, 1987.

[Raj 91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software - Practice and Experience*, 21(1):91-118, January 1991.

[Ungar 87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1987. ACM SIGPLAN Notices, 22(12):227-242, December 1987.

[Watt 90] David A. Watt. *Programming Language Concepts and Paradigms*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Yonezawa 86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1986. ACM SIGPLAN Notices, 21(11):258-268, November 1986.

# Index

Abstract object 2, 7, 9, 10, 13, 15, 18, 25
Abstraction 5-7, 9, 11, 15, 17, 19, 22, 23, 26
Actors 25
Actual input parameter 9-11, 14, 20, 22
Actual parameter 11, 14
Attribute object 11, 21, 22
Beta 7, 26
Classes 2, 6, 9, 13, 15, 23
Coarse-grained 19
Conformity 5, 7, 9, 21, 23
Delegation 7, 17
Distributed memory 5, 6
Dynamic interface 7
Emerald 6, 7, 25-27
Encapsulation 6
Fibonacci function 19
Fine-grained 1, 2, 5-7, 19, 20, 23, 25
First class objects 2, 15, 23
For-loop control object 22
Functions 6, 9, 13-15, 21
Future result object 7, 19, 20
Grain-size adaption 6, 23, 25
Inclusion 7, 17, 18
Inheritance 7, 17, 18, 26
Input parameter 9-11, 14, 20, 22
Interface inclusion 7, 17
Keywords 2
Object activation 26
Object constructor 10, 23
Object interface 17
Operations 5, 6, 9
Operators 20
Output parameter 9, 11, 14, 19
Path Expressions 7, 25
Polymorphic square function 14
Polymorphism 7, 13, 14, 23, 26
Precedence levels 11
Prototypes 9, 11, 21
Qualifier 9-11, 13, 14, 17
Scope rules 7
Side-effects 6, 19
Simple objects 6
Synchronization 7, 25
Type checking 7
Type checks 7, 23