periodica polytechnica

Electrical Engineering
51/3-4 (2007) 85–97
doi: 10.3311/pp.ee.2007-3-4.03
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

Objects and polymorphism in system programming languages: a new approach

Ádám Balogh / *Zoltán* Csörnyei

Received 2007-10-03

Abstract

A low-level data structure always has a predefined representation which does not fit into an object of traditional object-oriented languages, where explicit type tag denotes its dynamic type. This is the main reason why the advanced features of object-oriented programming cannot be fully used at the lowest level. On the other hand, the hierarchy of low-level data structures is very similar to class-trees, but instead of an explicit tag-field the value of the object determines its dynamic type. Another peculiar requirement in system programming is that some classes have to be polymorphic by-value with their ancestor: objects must fit into the space of a superclass instance. In our paper we show language constructs which enable the system programmer to handle all data structures as objects, and exploit the advantages of object-oriented programming even at the lowest level. Our solution is based on Predicate Dispatching, but adopted to the special needs of system programming. The techniques we show also allow for some classes to be polymorphic by-value with their super. We also describe how to implement these features without losing modularity.

Keywords

System Programming · Low-level Programming · Object-oriented Programmig · Inheritance · Polymorphism · Predicate Classes · Predicate Dispatching.

Ádám Balogh
Department of Algorithms and their Applications, ELTE, Hungary
e-mail: bas@elte.hu,

Zoltán Csörnyei
Department of Programming Languages and Compilers, ELTE, Hungary
e-mail: csz@inf.elte.hu

1 Motivation

The object-oriented programming model has enjoyed widespread use in application programming for many years. Encapsulation, polymorphism and inheritance give the programmer more powerful and safe tools for program development than the old imperative model. There are several programming languages and developer tools supporting it. A modern program is expected to be written fully object-oriented. More advanced paradigms such as aspect-oriented programming, subject-oriented programming and adaptive programming are also subject of research for more than a decade. There are also experiments about object-oriented operating systems, system software that allows every resource of the computer system to be handled as objects.

Although being almost forty-year old, the object-oriented programming paradigm could not conquer the lowest level of software development: the core of system programs, which work with externally defined data structures are still written in imperative languages, such as C or sometimes even assembly. The main reason for this is that classic object-oriented languages use a special representation for objects, because they have to store the dynamic type of every object to be able to dispatch virtual methods. This is usually done by an extra field at the beginning or the end of the object's representation, which points to the virtual method table of the class the object belongs to. (For example, *C++* stores this pointer at the end of the object if the class has at least one virtual method.) Low-level data structures, which are often defined by the hardware manufacturer or the programmer of a lower software layer (e.g. microkernel, operating system, middleware etc.) do not allow such memory overhead. On contrary, they determine the exact representation of the data which must be followed. There is usually no space left for extra information such as pointer to virtual method table.

However, the set of system data structures often compose a hierarchy similarly to a class tree: there are generic structures, and more specific ones. They have no explicit dynamic type, but the values in the fields determine which specific structure type the variable belongs to. There are often fields the whose type depends on the values in other fields. Thus there is no need for

explicit fields denoting the dynamic type of the variable, because values of fields inside the data structure determine it.

A typical example for this hierarchy of system data structures are the POSIX socket address descriptors (Fig. 2). The `bind()` and `connect()` functions take the socket address as `struct sockaddr`, which one is defined the following way:

```
struct sockaddr {
   sa_family_t sa_family;
   char sa_data[14];
};
```

However, no existing socket types have this address structure, but a socket type specific one. For example, *TCP/IP* sockets use `struct sockaddr_in`, while *Unix* sockets use `struct sockaddr_un`:

```
struct sockaddr_in {
   sa_family_t sin_family;
   in_port_t sin_port;
   struct in_addr sin_addr;
   unsigned char sin_zero[
      sizeof(struct sockaddr)-
      sizeof(sa_family_t)-
      sizeof(in_port_t)-
      sizeof(struct in_addr)];
};


struct sockaddr_un {
   sa_family_t sun_family;
   char sun_path[108];
};
```

The fields `sin_family` in `struct sockaddr_in` and `sun_family` in `struct sockaddr_un` must always have the values `AF_INET` and `AF_UNIX` respectively. Functions such as `bind` and `connect` getting socket addresses as arguments always expect `struct sockaddr` as formal argument:

```
int bind(int socket,
         const struct sockaddr* address,
         socklen_t address_len);
int connect(int socket,
         const struct sockaddr* address,
         socklen_t address_len);
```

If we call these functions for *TCP/IP* or *Unix* sockets, we have to use `struct sockaddr_in` or `struct sockaddr_un` variables as actual parameters and cast them to `struct sockaddr`:

```
bind(sock, (struct sockaddr*) addr,
     sizeof(addr));
```

Since *POSIX* is language-independent, arguments of the functions do not have explicit type information. Thus the functions decide on the `sa_family` field of their parameter which data type to use for interpreting the remaining fields as an address. In an object-oriented language the `struct sockaddr` could be an abstract class, without any fields at all, and `struct sockaddr_in` and `struct sockaddr_un` its concrete descendants. There would be no need for the `sa_family` field, explicit dynamic type tag could be used instead to recognize the object. The above shown type casting would be unnecessary as well, because a variable with static type `struct sockaddr` could be used and instantiated with `struct sockaddr_un` or `struct sockaddr_in`.

The generic and specific socket address descriptor types are very similar to classes: the `sa_family` field is the tag that determines dynamic type of the object, the remaining fields, which are different for different socket families contain the data of the address itself. The application programmer creates a specific socket address, and handles it over to the `bind` or `connect` function. The function then dispatches on the value in the `sa_family` field to the appropriate code.

Although in *POSIX* socket descriptors the type tag field is the first word of the structure, this is not always the case. Data structures defined by the hardware's manufacturer are usually divided into bit-fields of various length. The fields which determine the name, type and length of others reside somewhere in the middle of such structure. Furthermore, there are often more of them: the value in one field determines the type of some others, among which there are also fields determining the type of the rest. So we have a class-tree, but instead of one single type-tag field, we have more. Also another typical requirement in case of these structures is that unlike *POSIX* socket descriptors, which are referenced by pointer only, they usually have to fit into a fixed slot. This is the reason why specific structure types all have the same length as the generic one.

Fig. 3 shows the format of a generic descriptor of the *i386* architecture and its specializations. Since descriptors are stored in a descriptor table, which is a vector, they all are equal in size. The S field (44th bit) determines whether the structure describes a memory segment or system object. If its value is one, the value in the X field on the 43rd bit decides if it describes a data- or a code segment. Similarly, *Type* field on the 40th through 43rd bits determines if it is a descriptor of a system segment or for a gate. We cannot use the simple concatenation of these two fields for type tag, because in the $S = 1$ case the X field is one bit long, while in the $S = 0$ case *Type* occupies four bits. We have to use Boolean expressions such as $S = 1 \wedge X = 0$, $S = 1 \wedge X = 1$, $S = 0 \wedge Type \in \{0..3, 9, 11\}$ and $S = 0 \wedge Type \in \{4..7, 12, 14, 15\}$. This leads to our main idea: system data structures can be handled as *Predicate Classes*, where instead of explicit tag fields, a predicate on the data fields determines the dynamic type of an object. In this paper we present language constructs which are based on this idea, however they are developed especially for use in low-level programming.

The language constructs presented here are language-independent. However, this paper is a part of a research project where the final goal is a programming language for developing highly reliable system programs in an efficient way. We use the syntax of our language under development also in the examples of this paper. The main reason, beside consistency, is that it has a *Pascal* or *Ada*-like syntax which is more structured than a *C*-

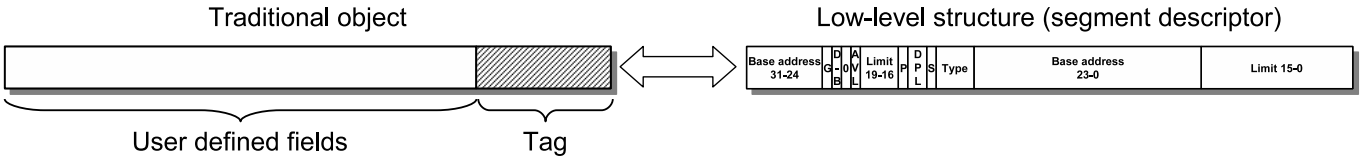

Fig. 1. Traditional object format and a segment descriptor in the i386 architecture

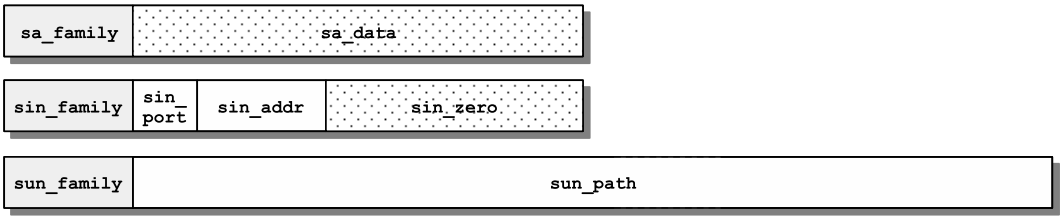

Fig. 2. POSIX socket address descriptors: generic, TCP/IP and Unix

like one. In this paper we do not give an exact definition of the language, but of course, we present basic syntax issues to make our sample source codes understandable.

The remaining of this paper is organized as follows. In Section 2 we review the basic types which will be used to build classes. Since executable instructions are not shown in the examples, we do not describe them in this paper. Section 3 presents *System Predicate Classes* as new language constructs, and shows their definition, inheritance and use. In Section 4 we discuss implementation issues. Section 5 describes problems that must be solved and our plans about them in the near future. In Section 6, we review past papers related to ours. Finally, Section 7 concludes.

2 Basic types

System data structures consist of fields of various types and sizes. These types are usually simple discrete types such as signed or unsigned integers, enumerated types, characters and Booleans. The size of these discrete types varies from a single bit to more words. Structures are composed by these types in such a way that the size of the whole structure occupies one or more words. *C* introduces bit fields to enable the programmer to define such structures. However, bit fields are no regular types, they can only be define inside structures and can have only types signed and unsigned int.

Our solution is more general: we allow declaration of standard variables having any discrete type with almost arbitrary length. These variables always occupy a whole number of bytes, when they stand alone, but in structures they are arranged similarly to bit fields in *C*. The advantage of this is type safety: assigning the value of a variable to a field of the same size will always be successful without overflow.

2.1 Generic discrete types

An n bit long unsigned integer is declared as `Unsigned(`n`)`, where n is an arbitrary integer. (Of course, the variable has to fit into the memory.) Signed integers are declared similarly, thus with the syntax `Signed(`n`)`, but here we do not allow the length to be less than two. The main reason for this is that a field that can have values -1 and 0 is very improbably, thus such a declaration can be considered as programming error.

Enumerated types are defined as follows: $\{i_1, i_2, \ldots, i_m\}(n)$, where $i_k (1 \leq k \leq m)$ are the values of the type. For example a 2 bit long field holding the three RGB colours is defined as `{Red, Green, Blue}(2)`. All the values have to fit into the space the variable occupies, thus $n \geq \log_2 m$ must hold. Characters are a separate type with declaration syntax `Character(`n`)`, but they are considered as an enumeration type holding all the *ASCII* characters. Thus the length of a character field has to be at least 7 bits long. Also Booleans are an own type of the language, however `Boolean(`n`)` corresponds to `{False, True}(`n`)`, where n can be any positive number similarly to the case of integers.

2.2 Predefined discrete types

Every architecture has a built-in integer type which is called machine word. The length of this word corresponds to the size of the integer registers in the processor, which is nowadays 32 or 64. Working with this type (i.e. 32 bit or 64 bit long integers) usually results in optimal performance, thus it is reasonable to declare discrete variables with this word length every time when it is possible. However, different architectures may have different machine word length, thus porting programs from one architecture to another one would mean that almost every discrete variable declaration has to be rewritten.

To make development of efficient and portable software easy, our language has predefined types with the length of a machine word. `Word` corresponds to `Unsigned(`m`)`, `Integer` to `Signed(`m`)` while `Bool` to `Boolean(`m`)`, where m is the length of the machine word in bits. Enumerated types of machine word length are defined by `Enum(`$i_1, i_2, \ldots, i_n$`)` as $\{i_1, i_2, \ldots, i_n\}(m)$ with m the same as before.

Beside machine words, programmers often need to work with

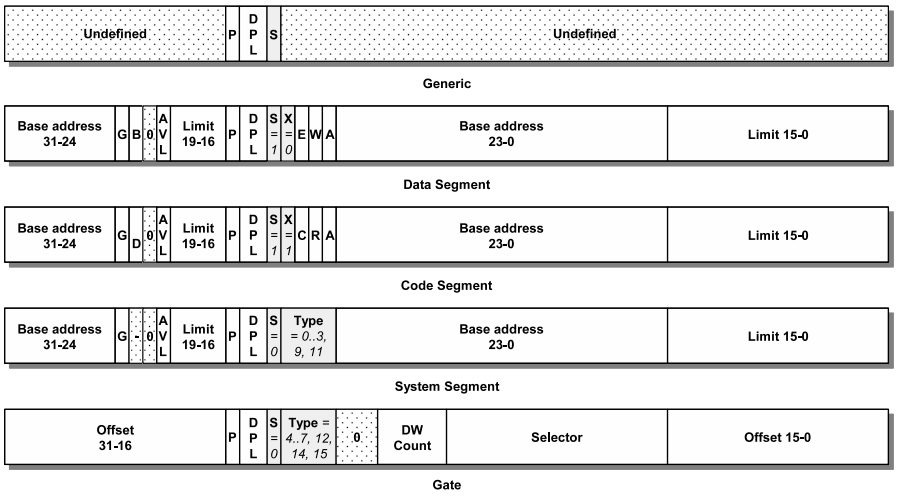

Fig. 3. Generic and specific descriptor formats on the i386 architecture

bytes. Bytes are 8-bit long on every modern architecture, but we introduced the type `Byte` as `Unsigned(8)` to make them easier to recognize. Similarly, `Bit` is equivalent to `Unsigned(1)`. For the same reason we also introduced `Char` as `Character(8)`. For *Unicode* characters the appropriate types have to be defined in external libraries.

Regarding the predefined types, one may ask the question we have introduced new names for `Integer`, `Word`, `Bool`, `Char` and `Enum` instead of allowing omission of length after `Signed`, `Unsigned`, `Boolean`, `Character` and `{...}`? Our answer is pretty simple: we would like to prevent programming errors resulted from forgotten lengths. Furthermore, we think that the code is more easy to read if separate names are used for generic and specific types.

2.3 Floating point types

Floating point types correspond to the *ANSI/IEEE 754* standard. `Single` is 32 bit long, and is implemented on every architecture. `Double` is 64 bit long, and it is usually available on every modern architecture. The 80 bit long `Extended` and 128 bit long `Quadruple` are optional, however one of them is defined on almost every modern processors. All data types available on the processor must also be available in the language implementation of the corresponding architecture. All these types are represented and handled exactly as defined in the standard. We also introduced the type `Float`, which is equal to the suggested floating point type on the given architecture. Note that this may differ from both the machine word length and the length of the floating point registers. For example, on the 32 bit *i386* architecture the registers are 80 bit long, but the suggested floating point type is the 64 bit long `Double`. Floating point types must always be byte aligned, even as fields in data structures. Since floating point types are very rarely used in system programming, we do not write more about them.

2.4 Type constructions

The syntax of array definitions is the following: `array[`$r_1, r_2, \ldots, r_n$`] of` t, where $r_i (1 \leq i \leq n)$ are ranges of format $k..l$, with k and l being values of the same discrete type and the integer representation of k must be less than or equal to l, and t an arbitrary type. Multidimensional arrays are stored in the memory first grouped by the last range, and then by every range to the first one.

Strings are arrays of characters, but to make their declaration easier we introduced the `String[`r`]` syntactical sugar for `array[`r`] of Char`. Note that both for general arrays and strings ranges not beginning with zero may have a negative impact on the performance of the program, because more computation is needed to determine the position of an element.

Object-oriented languages usually do not have record type, because class means a good replacement of it. This is especially true in our language: since we use no tag fields in our objects, a class without methods is exactly the same as a record in an imperative language. Furthermore, using predicate classes also eliminates the need for unions and even for variant records.

In system programming pointers are used very often. `Pointer to` t or simply `^`t is a type which is able to store the address of a variable of type t. On architectures having special addresses the `Near` and `Far` keywords may precede the `Pointer` keyword or the `^` symbol. Default is `Near`.

3 System Predicate Classes

System predicate classes resemble the well-known classes of classic object-oriented languages in many issues: they have data tags, methods, and new classes can be created from an existing one with the help of inheritance. However, objects of these classes are represented in the memory exactly as *C*'s structures: only the fields defined by the programmer are stored in the same order as it appears in the source code. There is no pointer before the beginning or after the end of the fields to a pointer to virtual method table. This makes it possible to define system structures

as classes. Definition of a root class named C is the following:

```
a class C is
    δ1;
    δ2;
    ...
    δn;
end C;
```

where a is the default access level of the class $\delta_i (1 \leq i \leq n)$ are data field declarations of the form $f : a\ t := d$ with f as field name, a as access level.t as type and d as default value (optional, if omitted then the := must also be omitted), or method declarations of the form a procedure $m(\alpha)$ or a `function` $m\alpha$): t with a as access level, m as method name α as argument list and t as return value type. (If α is the empty list, the parentheses are also omitted.) Two other special method formats are also possible which are explained further below in this paper.

To implement encapsulation similarly to conventional OOP languages, each data and method belongs to one of the three access levels, exactly as in *C++*, with the same keywords: `public`, `private` and `protected`. The access level is optional, if it is omitted, then the field or method belongs to the default access level of the class. If it is also omitted, then fields default to private, while methods to public. Fields in system predicate classes may have a special access level, defined by the keyword `reserved`. This access level is the lowest one, because these fields are not even accessible inside the class. This feature has double purpose: first, hardware manufacturers and low-level software designers often define fields as reserved. These fields must never be accessed in the software. The second purpose is more interesting, because it is connected to a unique feature of *system predicate classes*: fields defined as `reserved` can be overlapped in the inherited classes by new fields of arbitrary type. Even one long `reserved` field can be overlapped by more smaller ones, with total size of the original one. More about this feature see 3.1. Default access level of the class cannot be `reserved`.

The following example defines a simple class for a generic descriptor on the i386 architectures:

```
class Descriptor is
   Undefined1: reserved Unsigned(16);
   Present: Boolean(1);
   DescPrivLevel: Unsigned(2);
   DescType: {System, User}(1);
   Undefined2: reserved Unsigned(44);
   virtual procedure LoadDescriptor is empty;
end Descriptor;
```

Most of the fields are marked as `reserved`, because their structures and types depend on the value in the `DescType` field. The `LoadDescriptor` method has an empty body in this generic class, since segments and gates are different to load. The `empty` keyword is just a short form of empty `begin end` block. The `virtual` keyword is explained in 3.1. In a high-level OOP language this class would be defined as abstract together with the `LoadDescriptor` method, but in system programming it has no use. First of all, *system predicate classes* are *predicate classes*, thus denying instantiation does not save object from being instance of an abstract superclass, because object type changes during the execution by assigning new values to its fields. Secondly, in system programming abstract methods cannot be defined because of the lack of exception handling: every method has to be callable and do something (at least return). Thus there is no keyword for defining abstract classes in the language. Instead, *predicate classes* have other tools for preventing objects to belong to such a class as our generic descriptor. These tools are shown in the next sections.

3.1 Inheritance

Inheritance in object oriented languages is a tool for defining specialized versions of a class. In the conventional OOP languages subclasses inherit the fields and methods of their superclass, they may have additional fields, additional methods and also may override some of the inherited methods. Subclasses are polymorphic with their super, thus a variable with type of the superclass can also store an object of the subclass. Since superclasses often contain additional fields, thus they are bigger in size. This is only possible if this variable is a pointer to the object. The static type of an object variable is the type of the variable itself: the class that appears in its declaration. On the other hand, the object the variable stores may have another type which is a descendant of its static type: this is the dynamic type of the object variable. The dynamic type can change during the execution only whenever a new object is assigned to the variable, since the type of the same object cannot be modified after its creation. Methods can be divided into two groups: static or nonvirtual methods belong to the variable that stores the object: thus their call is already fixed during compile time, since the type of the variable is defined in the source code. The other group of methods is called dynamic or virtual methods because they belong to the object itself: if a virtual method is overridden in a descendant class, and a variable contains an object which belongs to this class then the overridden method is called irrespectively of the static type of the variable. Since dynamic type is only known in run-time these calls cannot be fixed in compile-time. Instead, these methods are called through the virtual method table. Virtual method table is a vector belonging to the class and contains the addresses of its virtual methods. The object has a pointer to this table at its beginning or end, and if a virtual method is called, it jumps to the appropriate address in its actual VMT, which always corresponds to the dynamic type of the object.

Inheritance of *predicate classes* is similar to that of conventional ones, but an object of a *predicate class* does not contain explicit dynamic type tag such as VMT. Instead, every subclass of the same superclass have a unique property which distinguishes object of that subclass from the others and also from its parent. A property is described by a predicate, i.e. a Boolean expression on the data in the object's fields. *Predicate classes* may also have virtual and nonvirtual methods. Nonvirtual methods

are called in the same way as in case of normal classes, but the calling of virtual methods is essentially different because of the lack of VMT. Whenever a virtual method is called, a dispatcher routine is invoked which evaluates the predicates of all the subclasses of the static type class of the variable. If a predicate of a subclass is true, then the dynamic type of the object is that subclass. Of course, if the subclass also has subclasses, more predicates have to be evaluated until determining the dynamic type of the class. Since dynamic type depends on the values in the fields of the class, it can change more dynamically during execution, not only by assigning new object to the variable.

Inheritance of *system predicate classes* is a little bit different. As we already mentioned in the introduction, system programmers often need to store the object itself in a variable of a type which is ancestor of the object's type, and not only a reference to it. This feature we call *polymorphism by value* and is a unique feature of *system predicate classes*. These classes must have the same size as their descendants, no new fields are added to the end of such a class, but existing fields are overlapped by them. Only fields defined as `reserved` can be overlapped.

3.1.1 Predicates

Our language constructions do not allow objects to belong to more than one subclasses of the same class. To prevent this, we have to ensure that predicates of the subclasses are disjoint. Checking this for arbitrary first-order Boolean expressions would be too difficult, so we restrict the terms that can appear in the predicate expressions. In system programming typical terms are hardly ever more complex than checking values of discrete fields being in a given constant set. For this reason we only allow comparisons of fields and constant expressions of the following formats: $f = C$, $f \neq C$, $f < C$, $f > C$, $f \leq C$ and $f \geq C$, where f is a discrete field of the superclass and C is a constant expression.

To check disjoincy of predicates, we first convert them to clause sets containing expressions of the format $f \in \{C_1, C_2, \ldots, C_n\}$. To achieve this, we collect constants for a field in the same clause into sets, thus every expression in the clauses checks whether the value of a given field is member of a specific set. Every clause can contain at most one expression for the same field.

A modified resolution algorithm can then be used to check that every pair of predicates are disjoint. Having them converted to the format described above, we take the union of the two clause sets, and search for resolvent clauses. The only difference from the conventional resolution algorithm is that we regard two clauses as resolvents, and only if they both contain expressions for the same field with disjoint sets. At he end of the algorithm, we either get the empty clause, what means the two predicates are indeed disjoint, or we get stuck, that means the opposite.

Although the problem is still $\mathcal{NP}$-hard, in the practice we only have very simple predicates: for a given level of the class-tree the separation to disjoint subclasses is usually done based on one or two fields, with almost only equalities. To further improve performance of checking for disjoincy, we introduced a special operator to be used only in predicates: the `isnot` c expression is equivalent with the $\neg P$ one, where P is the predicate of class c. Of course, class c must be a subclass of the same superclass as the subclass with this expression in its predicate. This operator also helps the programmer to divide the possible value-set of a superclass into disjoint subclasses, which together cover the whole value-set. This way the superclass is abstract, because every of its objects belongs to exactly one of its subclasses, the superclass itself cannot have any instances. Thus calling methods considered to be abstract are also prevented.

Fig. 5 shows the class hierarchy of descriptors on the *i386* architecture. Classes for *Descriptor* and *Memory Segment Descriptor* are abstract since the `S` bit cannot take other value than zero or one, and so does the X bit. In these simple cases we not even need the help of the `isnot` operator. However, class for *System Descriptor* is not abstract, because values 8, 10 and 13 of the `Type` field do not satisfy the predicates of any of its descendants. Since these values are invalid, the virtual methods of *System Descriptor* must contain code that signals runtime error rather than being empty.

3.1.2 Simple inheritance

Because of the special features of *system predicate classes*, we have to distinguish four kinds of inheritance. The first one is the simplest, so we call it *simple inheritance*. The field structure of subclass and superclass are exactly the same, only behaviour of the subclass is different. Since field structure is the same, size is also the same, thus the two classes are polymorphic by value. Different behaviour means that child class can have new methods or override existing ones. There is no special syntax for overriding: if the new method has the same name and signature as an existing one, it overrides that one. Virtual methods can only override virtual ones, while nonvirtual methods only nonvirtual ones. In other cases compilation error occurs.

Syntax of a class definition which *simply inherits* from another one is the following:

```
a class C' inherits C when P is
   μ1;
   μ2;
   ...
   μn;
end C';
```

where C' is the name of the subclass, C the name of the superclass, P the predicate and $\mu_i (1 \leq i \leq n)$ the method declarations or definitions. The default access level (a) of the class must be the same as for the superclass, and so must the access level of the methods. This requirement prevents programming errors that are difficult to find. In the following example we further divide the class of *i386 data segments* into two subclasses. The division is based on the *Ex-*

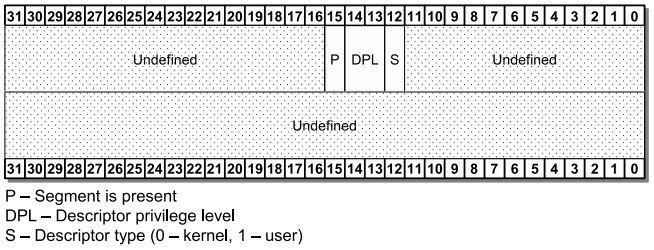

Fig. 4. Generic descriptor on the i386 architecture – more detailed format

pansion Direction bit. Let us suppose that this bit is defined with the name `Ex_Dir` and type `{Up, Down}(1)` in the `Data_Segment_Descriptor` class. The *Expansion Direction* determines whether valid offsets are in the range `0...Limit` (heap) or in the range `Limit...`$FFFFFFFFH$ (stack). (For the sake of simplicity we do not take the *Granularity* and *Bit* bits into account in the example.) The `Valid_Offset` checks if a given offset is valid for the segment, and is defined with empty body in the (abstract) superclass. Definition of the two subclasses is the following:

```
class Heap_Descriptor inherits
Data_Segment_Descripor when Ex_Dir=Up is
   virtual function Valid_Offset
(Offset: Word): Boolean is
   begin
      return Offset<Limit;
   end Valid_Offset;
end Heap_Descriptor;

class Stack_Descriptor inherits
 Data_Segment_Descripor when Ex_Dir=Down is
   virtual function Valid_Offset
(Offset: Word): Boolean is
   begin
      return Offset>Limit;
   end Valid_Offset;
end Stack_Descriptor;
```

3.1.3 Overlapping

The second kind of inheritance, *overlapping* is also for classes considered to be polymorphic by value with their parent. However, in this case not only the behaviour of the subclass may differ from its parent, but also its fields. New and overriding methods can be defined the same way as in case of *simple inheritance*. The main difference is that the whole field-structure has to defined anew.

The field structure of the subclass cannot be arbitrary. Its total size must be the same as that of the superclass. Also fields not defined as `reserved` in the superclass must have same position, type, length and access level in the subclass, only their name can differ. (In the predicate of the class the field names of the superclass must be used.) However, every `reserved` field can be overlapped with arbitrary new fields with their total size equal to the length of the `reserved` field. It is also allowed that new fields overlap two or more neighbouring `reserved` fields together, regarded as one long `reserved` field. The formal syntax for overlapping is the following:

```
a class C' overlaps C when P is
   δ1;
   δ2;
   ...
   δn;
end C';
```

where a, C', C and P are the same as in case of *simple inheritance*, while $\delta_i (1 \leq i \leq n)$ are data field declarations and method declarations or definitions.

An example for overlapping is the definition of different descriptors. We have already seen the definition of the generic descriptor format earlier. Now we show how to define memory segment descriptors. A descriptor describes a memory segment in the case the bit S is set to 1. In the example we omit methods that would be there in real software, such as segment loading, unloading etc.

```
class Memory_Segment_Descriptor overlaps
 Descriptor when Desc_Type=User is
   Base1: private Unsigned(8);
   Granularity: {Byte, Page}(1);
   AddrSize: {S16, S32}(1);
   Zero1: reserved Bit:=0;
   Avail: Bit;
   Limit1: private Unsigned(4);
   Present: Boolean(1);
   DescPrivLevel: Unsigned(2);
   DescType: {System, User}(1);
   SegType: {Data, Code}(1);
   Undefined: reserved Unsigned(2);
   Accessed: Boolean(1);
   Base2: private Unsigned(24);
   Limit2: private Unsigned(16);

   procedure SetBase (Addr: in Unsigned(32)) is
   begin
      Base1:=Addr/1000000H;
      Base2:=Addr and FFFFFFH;
   end SetBase;

   function Base: Unsigned(32) is
   begin
      return Base1*1000000H or Base2;
   end GetBase;

   procedure SetLimit (Lim: in Unsigned(20)) is
   begin
      Limit1:= Lim/10000H;
      Limit2:= Lim and FFFFH;
   end SetBase;

   function Limit: Unsigned(20) is
   begin
      return Limit1*10000H or Limit2;
   end GetBase;
end SegDesc;
```

As shown in the figure, the segment descriptor has a base address and a limit broken into two parts. To make it comfortable to the programmer, we applied a trick here: the two parts of

both are defined as private, and appropriate public methods are defined for getting and setting the value of the `Base` and the `Limit` virtual fields. A syntactic sugar of the language makes it possible to use the `:=` operator instead of calling the appropriate `Set` method. With a pure record in an imperative language the above trick could not be applied. There we would have to compose and decompose the two parts of the fields manually always when accessing them. Of course, we could use a function for that, but it would be outside of the record.

Since the fields of a class can be overlapped exactly the same field structure one may think that *simple inheritance* is superfluous. However using *overlapping* only if the field structure in the root class has to be changed during the development, then all of its descendants have to be changed to remain consistent. One single forgotten class may cause compilation error, or even worse – if the change is around the `reserved` fields – incorrect behaviour of the program. Further difficulties could arise if the program is developed by a team where different classes of the same class-tree are maintained by different programmers. Using *simple inheritance* eliminates these problems and also improves the readability of the source code.

3.1.4 Extension

Extension is typical form of inheritance in the classic object-oriented languages. The field structure of subclass begins with the fields of the superclass and is extended by new fields at the end. Because of the new fields objects of the subclass have bigger size than objects of the superclass, thus the two classes are polymorphic only by reference. Definition of new and overriding methods is the same as before.

The syntax for an extended class definition is the following:

```
a class C' extends C when P is
   δ1;
   δ2;
   ...
   δn;
end C';
```

where a, C', C and P are the same as in case of *simple inheritance*, while $\delta_i (1 \leq i \leq n)$ are new data field declarations at the end of the class and new or overridden method declarations or definitions.

Appending of new fields to the end of the original ones is not mandatory. However, even if no fields are appended, the extended class is not the same as a simply inherited one. The subclass and the superclass are polymorphic only by value in this case as well.

Special care must be taken when working with extended objects. If a pointer defined to point to a type of the superclass and currently points to an object of the subclass is dereferenced and the value is assigned to a variable with static type of the superclass, the extended fields are lost. The same happens when using the dereferenced object as actual parameter to a method where type of the formal is the superclass. In this case a method call on the truncated object could lead to illegal memory references, because the dispatcher function invokes the method of the extended object which does not know that the object is truncated. This kind of error is very difficult to find. To help the programmer to prevent them, a warning is given to the programmer whenever he tries to store a dereferenced pointer to an object in a variable or use it as actual parameter. For a well written program no warnings are received, because dereferencing pointers and using their values as actuals or storing them in variables is usually a bad programming technique.

3.1.5 Redefinition

Redefinition is a mixture of *overlapping* and *extension*: New class may overlap `reserved` fields with new fields and also new fields can be appended to end of the class. Moreover, if the last field of the superclass is `reserved`, this field may be overlapped new fields of greater size. The only restriction is that the size of the new class must be greater than or equal to the size of the original one. Of course, the subclass is polymorphic with the superclass only by value also if its size is not bigger. The fields not defined as `reserved` can only be renamed, similarly to the case of *overlapping*, and the rules for new and overlapping methods are the same as before.

Although *redefinition* is also capable to substitute *extension* it is not unnecessary for the same reason as *simple inheritance*. The risk of object truncation is the same as in case of *extension*.

The syntax for a redefined class definition is the following:

```
a class C' redefines C when P is
   δ1;
   δ2;
   ...
   δn;
end C';
```

where a, C', C, P and $\delta_i (1 \leq i \leq n)$ are the same as in case of *overlapping*.

A good example for *redefinition* of classes are the *POSIX* socket address descriptors, as shown earlier in Fig. 2. Although the size of `Sockadd_IN` is the same as the size of `Sockaddr`, we also use *redefinition* instead of *overlapping* because of consistency with `Sockaddr_UN` which has greater size. We also do not need `Sockadd_IN` to be polymorphic by value with its parent.

```
class Sockaddr is
   SA_Family: SA_Family_T;
   SA_Data: reserved String[14];
   virtual procedure Bind
(S: ^Socket) is empty;
   virtual procedure Connect
 (S: ^Socket) is empty;
end Sockaddr;

class Sockaddr_IN redefines Sockaddr
 with SA_Family=AF_INET is
   SIN_Family: SA_Family_T;
   SIN_Port: IN_Port_T;
   SIN_Addr: IN_Addr;
```

```
   SIN_Zero: Invalid;
   virtual procedure Bind (S: ^Socket);
   virtual procedure Connect (S: ^Socket);
end Sockaddr_IN;

class Sockaddr_UN redefines Sockaddr
 with SA_Family=AF_UNIX is
   SUN_Family: SA_Family_T;
   SUN_Path: String[108];
   virtual procedure Bind (S: ^Socket);
   virtual procedure Connect (S: ^Socket);
end Sockaddr_UN;
```

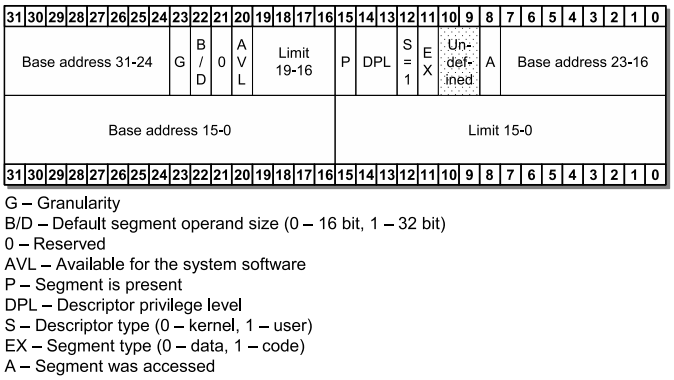

Fig. 6. Memory segment descriptor on the i386 architecture – more detailed format

The `Bind` and `Connect` procedures are now methods of the class `Sockaddr`. In practice, when implementing *POSIX* in our language this is perhaps not the case, these methods may belong to the `Socket` class, but eventually at some points they call a method of the class `Sockaddr` to bind or connect to the address. However, to simplify things we considered that `Bind` and `Connect` themselves are methods of `Sockaddr`. There is no need for the parameter `Size`, because the dynamic type of the object determines its size. Both methods are abstract in the class `Sockaddr`, i.e. they have empty bodies. However class `Sockaddr_IN` and `Sockaddr_IN` override the methods to bind or connect themselves to the socket given in the argument. Whenever the method is called on an object with static type `Sockaddr` a dispatcher method is executed which examines its `SA_Family` member and jumps to the appropriate routine. The polymorphism by reference thus also eliminates the need for explicit type cast.

Although there are a few other socket address families, they cannot cover the whole integer range the `SA_Family` field can take as value. To prevent calling empty methods it is suggested to create a failsafe subclass called for example `Sockaddr_Err` whose methods display error in some form whenever they are called. Another solution is to replace the empty bodies in the class `Sockaddr` itself to display the run-time error message.

Fig. 7 shows the relation between super- and the subclass in the four cases of inheritance regarding size and field structure. The main reasons that *system predicate classes* have four kinds of the inheritance instead of just one similarly to the conventional classes lies in the special properties of system data structures

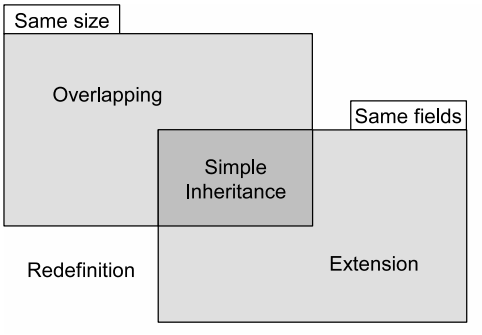

Fig. 7. Relation between the super- and subclass in cases of the four different kinds of inheritance regarding size and field structure

as classes. The reason for distinguishing inheritances that preserve original field structure or let it replace by new one is that these classes are not always extended at the end, but often there is space reserved for extension inside the structure. We also explained that always replacing them may cause several difficulties during the development. The reason for distinguishing inheritances that preserve original size or let it be greater is that some structures have to be polymorphic by value while for the rest it is sufficient if they are polymorphic by reference. Of course the same keyword could be used for the two cases, but for easier compiler implementation and better code readability we decided to separate them. Programming errors are also easier to find: if an overlapping class is accidentally defined to have greater size than its parent, the compiler displays an error message immediately for the definition. If we would use *redefinition*, the error would appear for the place where it is tried to be assigned by value to its shorter parent, perhaps in another module, maybe written by another programmer.

3.2 Objects

Objects are instances of classes, similarly to conventional OOP languages. They can be defined as local variables or allocated dynamically. Syntax of variable declaration is, similar to the *Pascal* language, the following:

```
var v1,1, v1,2, ..., v1,n: T1;
    v2,1, v2,2, ..., v2,n: T2;
    ...
    vm,1, vm,2, ..., vm,n: Tm;
```

where $v_{i,j} (1 \leq i \leq m, 1 \leq j \leq n)$ are the names of the variables and $T_i (1 \leq i \leq m)$ are basic types, type constructions (arrays, strings or pointers) and class names. Dynamic allocation happens by declaring pointer to the class type and then allocating memory for the class with the following operator: `new` C, where C is the name of the class. The result of the allocation is an object of type C, therefore it can be assigned to a variable with type C or an ancestor class of C. Dynamically allocated objects are freed by the `dispose` v instruction, where v is the name of the variable to be deallocated.

Like in conventional object-oriented languages, classes can contain two special kinds of methods, called *con-*

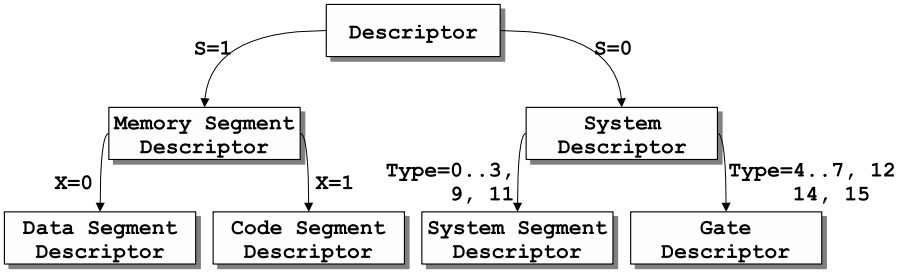

Fig. 5. Simple class hierarchy of the descriptors on the i386 architecture

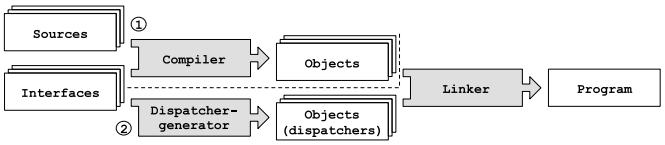

Fig. 8. Phases of compilation (1) and linking (2)

structors and *destructors*. The syntax of a *constructor* is `constructor` $C(\alpha)$, while destructors are defined by `destructor` C, where C is identical with the name of the class, and for the arguments list α the same rules apply as in case of normal methods. The constructor of an object is called whenever the object is created, but always after assigning default values to the fields: immediately after variable declaration or dynamic allocation. If the constructor has formal arguments, actual arguments must be given at the place of the creation after the name of the class. One single object may have more constructors with different argument signatures. However, there can be only one destructor for each class with no arguments at all. Destructors are called at the end of the object's lifetime: for local objects at the end of the block, for dynamically allocated ones at their deallocation.

4 Implementation issues

The implementation of a language based on the presented language constructs is in progress. Several experiments have already been conducted to prove that our idea is working in the practice. In this section we show how to implement our construct in such a way that we obtain a modular language that is also object-compatible with existing languages.

Complex software systems are usually developed by teams rather than individual programmers. This is especially true for system software which is too complex for a single programmer. Therefore our language must support team work, i.e. it has to be modular: programmers should be able to compile their code separately and then link the compiled objects together. In OOP languages this is usually solved in such a way that class interfaces (field and method declarations without their definitions) are separated from their body (method definitions). If all programmers have all the interfaces, each of them is able to compile the body of the own classes. They also may create own classes inherited from the classes of the commonly distributed interfaces or from common libraries.

In our language the sources of the difficulties are the dispatcher routines. In conventional OOP languages if new class is created, a new VMT is also created and the objects of the new class point to this new VMT. However in our languages classes have no VMTs but dispatching is done by dispatcher routines. These routines evaluate the predicates of all the subclasses of the class to find the correct method to be invoked. If a new subclass is created, its predicate has also to be evaluated. This means that dispatcher routines cannot be generated before link time, when all the subclasses of any class are already known.

To overcome this problem, we not only need a proprietary compiler for the implementation but also a proprietary linker. However this linker has also to handle objects compiled from other languages. This can be done easily if our linker calls the general linker of the operating system (e.g. `ln` on *Linux*) after generating dispatcher routines and compiling them into an object file. The format of this object file corresponds to the standard object file format of the OS that can be handled by the linker, and so the object files generated by our compiler. These three phases of compilation and linking are shown on Fig. 8.

To implement dispatcher routines in an efficient way, a little trick can be applied. Dispatcher routine is called as a subroutine, saving the return address in the stack or link register, depending on the architecture it runs on. After evaluating the predicates, the dispatcher routine invokes the appropriate method. The most straightforward way to do this is another subroutine call. However, this means that after returning from the method, control is transferred back to the dispatcher which returns again to the original caller. This double return can be eliminated, if the dispatcher routine jumps to the method body instead of calling it. In this case, the method returns directly to the caller. However implementing this is more difficult than the straightforward method, especially when compiling through *C*, what does not support jumping but calling only.

5 Open Problems and Future Plans

The final goal of our project is a language which can be used in a safe and efficient way to develop operating systems. Introducing *system predicate classes* that enable to handle system data structures is the first step to achieve this. Of course we would like to show this language working in the practice

thus our most important plan is to develop the compiler and the linker.

However, there are some open problems which are to be solved in the near future. The first one is the already mentioned object truncation problem. Our current suggestion in this paper is that the compiler has to give warning if a reference to an object is dereferenced and the value stored or used as actual parameter. We want to further study this problem maybe find better solutions as the result of our research work. Only solutions that omit run-time checks are acceptable since our language is used at the lowest level of software where no overhead is allowed.

Another possibility of programming error is that an object can be modified in such a way that it does not fulfil the predicate of the class which is the static type of its storing variable. In this case dispatching algorithm may find incorrect implementations of methods. A safe solution would be if dispatching routine of the root class would be invoked even if the static type of the variable is only an ancestor of it. However, this solution would increase the overhead of the program – in most of the cases completely unnecessarily. Another solution should be found for this problem that does not increase overhead.

We also think to further improve our constructs in the near future and to develop new language constructs that help system programmer to create safe programs in an efficient way. Our plans include multiple inheritance and generic classes, similarly to the generics of *Ada* or the templates of *C++*. Exception-handling at the lowest level is also an interesting topic that we would like to study. To show the features of the language in the practice, we also want to develop an interface to the *POSIX* library.

6 Related Work

One of the first constraint based object-oriented languages is *Cecil* by *Chambers* [8], [9]. In *Cecil* constrained descendants of a class are called *predicate classes*. This kind of inheritance extends the normal one, thus the object structure in *Cecil* consists of normal classes and predicate classes. *Cecil* also defines *implicit inheritance* between predicate classes, thus a class can be subclass of another one, if they are constraint-based (explicit) descendants of the same superclass and the constraint of the one is stronger than the constraint of the other one. Predicate-based dispatching of methods works dynamically, in case of ambiguities error messages are generated, thus *Cecil* includes a dynamic runtime environment. Furthermore, because of the normal inheritance, object representation is similar to other object-oriented languages, thus it must include a dynamic type tag. On the other hand, *Cecil* does not have encapsulation, thus methods are normal procedures and functions with tagged parameters.

The *e* programming language by *Hollander*, *Morley* and *Noy* [19] is used in the microchip industry for modelling and functional verification. The *e* language also combines the normal object-oriented mechanism with the constraint-based ones. Instead of creating new subclasses, *e* makes it possible to extend a class in-place. The paper describes the advantage of this language construction in functional verification. However, for general-purpose languages – including languages designed especially for low-level programming – subclass creation is more preferable. Furthermore, *e* is not as low-level language as ours since it does not allow the programmer the specification of the object's representation.

The *CCC* language by *Harada*, *Yamazaki* and *Potter* [18] enables to create objects with user defined structure in C. It is a small but highly efficient extension to C, its goal is to introduce classes and objects into system-programming. *CCC* allows defining polymorphic functions which are dispatched on the value of the arguments. The paper also describes an efficient dispatching algorithm. However, *CCC* does not define real classes, the keyword *class* is only used to arrange the polymorphic functions into a constraint-based hierarchy. Furthermore, *CCC* is just an extension to the C language, thus it has the most disadvantages of *C*, as for example not being type-safe.

The *SysObjC* language by the authors of this paper [4] is another *C* extension, which partially implements *system predicate classes* by enhancing standard C structures with methods and predicate based inheritance. It also allows all four kinds of inheritance described in this paper and thus *polymorphism by value*, but in a less safe manner. The language is intended to be an intermediate language between *C* and our future safe system programming language, and is hoped to ease migration for *C* system programmers.

Primitive concepts and defined concepts in *Yelland*'s experimental *Smalltalk* extension [29] correspond to normal classes and predicate classes. Defined concepts are limited to Boolean expressions examining whether an attribute is part of a fixed set. Although we also restrict our terms similarly, we allow more complex Boolean expressions to be built of them.

The *SELF* language by *Ungar*, *Smith*, *Hölzle et al.* [28], [27], [2] uses *dynamic inheritance* to change behaviour of an object during run-time. An object in *SELF* can dynamically replace its parents, so change the set of inherited methods. The *Garnet* system by *Myers et al.* [25] very similar, it also uses *dynamic inheritance* to change behaviour of objects. Although *dynamic inheritance* is more powerful than predicate dispatching, it is a less structured language construct: while any object may be assigned as a parent of any other one during run-time in the above languages, the DAG of predicate inheritance is always fixed during link time in our language. Yet more powerful than *dynamic inheritance* in changing the representation and implementation of an object is the `become:` operation of *Smalltalk-80*, which allows the identities of two objects to be swapped. However, it is yet more unstructured than *dynamic inheritance* and also difficult to implement efficiently.

7 Conclusions

We presented a new language construct, called *system predicate classes*. The main novelty of this construct is that it enables

object-oriented programming even at the lowest level, where no run-time environment exists, and different predefined data structures must be handled as objects. It applies encapsulation to objects as classic object-oriented languages, but with fixed. A very useful feature of the construct is that subclasses of a class can have different data structure, with renaming of fields and overlapping of fields which are undefined in the superclass. This allows to build hierarchical class trees of system data structures, where the value of some fields determine the type and name of other ones, without change in the structure's size. A well-composed class tree eliminates the need for almost all type-casts in the program. The only case where type cast is needed to cast a variable to a descendant of its static type, which is very common even in the high-level OOP languages. Also unions are obsolete when using *system predicate classes*, since inheritance replaces them in a more safe way.

We showed two small examples, where almost all features of the language were used. These examples were *POSIX* socket address descriptors and *i386* segment and gate descriptors, very typical ones in system programming. While the first one is an example for handling data structures of the operating system as objects, the second one shows that similar method can be applied to the hardware's data structures as well.

Modularity of the language enables larger systems by teams. Using standard object formats makes it possible to mix modules with other modules written in different languages. Efficient dispatcher routines ensure best possible performance of programs based on *system predicate classes*. Checking for disjoint predicates increases program safety.

References

1 *Advanced Micro Devices, Inc.*: *BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors*, May 2006, available at `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/32559.pdf`. Revision 3.00.

2 **Agesen O, Bak L, Chambers C, Chang B W, Hölzle U, Maloney J, Smith R B, Ungar D, Wolczko M**, *The SELF programmer's reference manual.*, 2000. Version 4.1.

3 **Balogh Á, Csörnyei Z**, *Multiple inheritance of system predicate classes.* (2006).

4 _____, *SysObjC: C extension for development of object-oriented operating systems*, PLOS 2006: Linguistic Support for Modern Operating Systems Workshop of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (San Jose, California, United States), October 2006. ACM Press.

5 **Caseau Y**, *An object-oriented language for advanced applications.*, TOOLS USA '91: Proceedings of the 5th International Conference on Technology of Object-Oriented Languages and Systems (Kaiserslautern, Germany), August 1991. Prentice-Hall.

6 **Caseau Y, Perron L**, *Attaching second-order types to methods in an object-oriented language*, ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science, vol. 707, Kaiserslautern, Germany, July 1993. Springer-Verlag.

7 **Caseau Y, Silverstein G**, *Some original features of the LAURE language*, OOPSLA '92: Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum), Vancouver, British Columbia, Canada, October 1992. ACM Press.

8 **Chambers C**, *Object-oriented multi-methods in Cecil*, ECOOP '92: Proceedings of the 6th European Conference on Object-Oriented Programming (O. Lehrmann Madsen, ed.), Lecture Notes in Computer Science, vol. 615, Springer-Verlag, Utrecht, The Netherlands, 1992.

9 **Chambers C**, *Predicate classes*, ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming (O.M. Nierstrasz, ed.), Lecture Notes in Computer Science, vol. 707, Springer-Verlag, Kaiserslautern, Germany, 1993.

10 _____, *The Diesel language specification and rationale.* Version 0.2., January 2006, available at `http://www.cs.washington.edu/research/projects/cecil/www/Release/doc-diesel-lang/diesel-spec.pdf`.

11 **Chambers C, Chen W**, *Efficient multiple and predicated dispatching*, OOPSLA '99: Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press, Denver, Colorado, United States, November 1999.

12 **Chambers C, The Cecil Group**, *The Cecil language specification and rationale.* Version 3.2., February 2004, available at `http://www.cs.washington.edu/research/projects/cecil/www/Release/doc-cecil-lang/cecil-spec.pdf`.

13 **Intel Corporation**, *Intel® Core™2 Extreme Processor X6800Δ and Intel® Core™2 Duo Desktop Processor E6000Δ Sequence.* Revision 002., September 2006, available at `http://download.intel.com/design/processor/datashts/31327802.pdf`.

14 **Ernst M, Kaplan C, Chambers C**, *Predicate dispatching: A unified theory of dispatch.*, ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming (Jul E, ed.), Lecture Notes in Computer Science, vol. 1445, Springer-Verlag, Brussels, Belgium, 1998.

15 **Goldberg A, Robson D**, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Massachusetts, United States, 1983.

16 *The Open Group.*, available at `http://www.unix.org/single_unix_specification/.`. The single UNIX specification, verison 3 (IEEE Std 1003.1 and ISO/IEC 9945).

17 **Hamer J**, *Un-mixing inheritance with classifiers.*, Multiple Inheritance and Multiple Subtyping: Position Papers of the ECOOP '92 Workshop W1, Utrecht, the Netherlands, June/July 1992.

18 **Yasunori Harada, Kenichi Yamazaki, Richard Potter**, *CCC:User-defined object structure in C*, ECOOP 2001: Proceedings of the 15th European Conference on Object-Oriented Programming (J. Lindskov Knudsen, ed.), Lecture Notes in Computer Science, vol. 2072, 2001, pp. 118–129.

19 **Yoav Hollander, Matthew Morley, Amos Noy**, *The e language: A freh separation of concerns.*, TOOLS Europe 2001: Proceedings of the 38th International Conference on Technology of Object-Oriented Languages and Systems (Wolfgang Pree, ed.), IEEE Computer Society, Zurich, Switzerland, March 2001.

20 **Igarashi A, Nagira H**, *Union types for object-oriented programming*, SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing, ACM Press, Dijon, France, April 2006.

21 **Wilf R. LaLonde, Dave A. Thomas, John R. Pugh**, *An exemplar based Smalltalk*, OOPSLA '86: Proceedings of the 1st Annual ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, SIGPLAN Notices, vol. 21, Portland, Oregon, United States, September 1986.

22 **McAllester D, Zabih R**, *Boolean classes*, OOPSLA '86: Proceedings of the 1st Annual ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, SIGPLAN Notices, vol. 21, ACM Press, Portland, Oregon, United States, September 1986.

23 **Millstein T**, *Practical predicate dispatch*, OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming,

Systems, Languages, and Applications, ACM Press, Vancouver, British Columbia, Canada, October 2004.

24 **Mugridge W B, Hamer J, Hosking J G**, *Multi-methods in a statically-typed programming language*, ECOOP '91: Proceedings of the 5th European Conference on Object-Oriented Programming (G. Goos, and J. Hartmanis, eds.), Lecture Notes in Computer Science, vol. 512, Springer-Verlag, Geneva, Switzerland, July 1991.

25 **Drad A. Myers, Dario A. Giuse, Brad Vander Zanden**, *Declarative programming in a prototype-insance system: Object-oriented programming without writing methods*, OOPSLA '92: Proceedings of the 7th Annual ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, SIGPLAN Notices, vol. 27, ACM Press, Vancouver, British Columbia, Canada, October 1992.

26 **Stein L A**, *A unified methodology of object-oriented programming*, Inheritance Hierarchies in Knowledge Representation and Programming Languages, Wiley & Sons, 1991.

27 **Ungar D, Chambers C, Chang B W, Hölzle U**, *Organizing programs without classes*, Lisp and Symbolic Computation, vol. 4, Kluwer Academic Publishers, June 1991.

28 **Ungar D, Smith R B**, *SELF: The power of simplicity*, OOPSLA '87: Proceeding of the 2nd Annual ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, SIGPLAN Notices, vol. 22, ACM Press, Orlando, Florida, United States, October 1987.

29 **Yelland P M**, *Expreimental classification facilities in Smalltalk*, OOPSLA '92: Proceedings of the 7th Annual ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, SIGPLAN Notices, vol. 27, ACM Press, Vancouver, British Columbia, Canada, October 1992.