Vrije Universiteit Brussel
Faculteit Wetenschappen

A survey on formal models for OO

Mens Tom

Tech report vub-tinf-tr-94-03

Department of Computer Science
Faculty of Science (FAC WET)
Brussels Free University (VUB)
Pleinlaan 2, 1050 Brussel
BELGIUM

Fax: (+32) 2-629-3495
Tel: (+32) 2-629-3308
FTP: progftp.vub.ac.be
WWW: http://progwww.vub.ac.be/

A survey on formal models for OO

Tom Mens

Department of Mathematics (DWIS)
Vrije Universiteit Brussel (VUB)
Pleinlaan 2, B-1050 Brussel, Belgium
Tel: +32.2.6293474
Fax: +32.2.6293495
E-mail: tommens@vub.ac.be
WWW: http://progwww.vub.ac.be/prog/persons/tommens/tommens.html

KEYWORDS. *Formal model, calculus, object-orientation, concurrency*

ABSTRACT. *In this text, an overview is presented of different kinds of formal models for OO that have been proposed over the years. We discuss both concurrent and sequential models. Within the realm of sequential models we make a distinction depending on whether the formalisms are based on lambda-calculus or not.*

A survey on formal models for OO

ABSTRACT. *In this text, an overview is presented of different kinds of formal models for OO that have been proposed over the years. We discuss both concurrent and sequential models. Within the realm of sequential models we make a distinction depending on whether the formalisms are based on lambda-calculus or not.*

1. Introduction

The aim of this paper is to provide a historic overview of the most important formal models for OO, starting with the very first, and ending with models that are still in development. This might help us in finding a way in the jungle of currently existing formalisms, and on deciding which model is the best choice for modelling a particular language.

There is a simple reason for the abundance of formal models for the object-oriented programming paradigm. Since OO lacks a simple model theoretical foundation for definition and discussion (unlike functional languages that can rely on the λ-calculus as basic formalism), there have been many attempts to fill this gap. This has lead to a multitude of different models for OO, concurrent as well as sequential. The aim of this paper is to describe the current state of the art in all of these formalisms.

Most researchers in the area of object orientation acknowledge the need for formal models of OO:

> "The development of concurrent object-based programming languages has suffered from the lack of any generally accepted formal foundation for defining their semantics."
>
> (Nierstrasz, 1992)
>
> "Unlike functional programming, grounded in λ-calculus, or logic programming, grounded in logic, object oriented programming lacks a simple model that we can use as a basis for definition and discussion."
>
> (Grogono & Gargul, 1994)
>
> "The arguments over what is fundamental in object-oriented programming have existed for as long as the field. Which features - classes, prototypes, inheritance, delegation, message passing, encapsulation, abstraction - are at the heart of object-oriented programming, and how these things relate to one another, are not issues that will soon be resolved."
>
> (Stein et. al., 1993)

A theoretical foundation for OO might help us in deciding which aspects are essential to OO and which are not (e.g. is the notion of class essential, or can it be modelled using more primitive concepts?). It could also be used to give a formal semantics to concepts that are more or less understood, but have no

generally accepted definition (e.g. What is an object? Should it incorporate state? Should it contain private methods?). A common framework is also useful for relating different approaches towards OO (e.g. class-based vs. object-based, delegation vs. inheritance, encapsulated inheritance vs. non-encapsulated inheritance). By expressing various object-oriented programming languages in the same formalism, one could compare their shortcomings and virtues, and eventually develop new languages with a more simple and orthogonal design, but with the same expressivity as currently available languages. Finally, formal methods could be used to establish properties of OOPLs to find more efficient compilation techniques.

The lay-out of the text is as follows. First of all, a distinction is made between concurrent and sequential models for OO. The non-concurrent ones can be divided into calculi that are extensions of λ-calculus and models that are not based on λ-calculus. In each of the cases, we give a brief discussion of the relations between the various approaches.

2. Concurrent OO-models

First we will investigate the realm of concurrent OO-formalisms. After having described the most salient features of each of the possible approaches found in literature, we will argue which of those models are most interesting from an object-oriented point of view.

Observe that developers of concurrent OO-calculi have totally different objectives than developers of sequential ones. While concurrent calculi focus on aspects like concurrency, distribution, persistence, object-identity and active objects, non-concurrent calculi emphasise aspects like encapsulation, inheritance, incremental modification and classes.

2.1. Actor Systems

One of the earliest attempts to provide semantic foundations for concurrent object-oriented languages was definitively the *actor model* (Clinger, 1981). Actor systems can be regarded as a model for object-based programming, because they support a notion of objects (actors), messages and message sending.

The model is based on *asynchronous* message passing, and was developed by Gul Agha (1986) and Carl Hewitt (1977) at the MIT (Agha & Hewitt, 1985). Asynchronicity means that when an actor sends a message to another actor, it does not have to wait for an answer, but can immediately proceed to process another message. For this reason, actor systems are a computational model suited for describing the execution of programs on massively parallel computer systems.

2.2. Process Calculi

In *process calculi*, objects are viewed as "patterns" of agents that obey the higher-level protocols established by the programming language. Of course, one would like the primitives of the process

calculus to be as natural as possible for modelling the concepts of the programming language. In contrast with actor systems, process calculi are based on "synchronous" message passing. Process calculi are very important, since most of the currently existing models for concurrent OO make use of its underlying principles. Below we will give an overview of the most important process calculi, and their significance to concurrent OO.

2.2.1. CSP

One of the first and best known works on process calculi is Hoare's model of *Communicating Sequential Processes* (1978, 1985). There have been attempts to use this model as a foundation for object-oriented programming. For example, Hailpern & Nguyen (1987) have proposed a model for object-based inheritance in which the objects behave like communicating processes, but with a different message passing mechanism. This model leads to a formal semantics for both objects and inheritance.

2.2.2. CCS

Robin Milner (1980, 1989) proposed a calculus for concurrent systems which is now publicly known as *CCS*. Initially there were some difficulties to see how full mobility among processes could be handled algebraically. A first solution to this was proposed by Engberg & Nielsen (1986). They added mobility to CCS while preserving its algebraic properties. An alternative way to achieve mobility is by following the higher-order approach where processes can be transmitted as messages (Thomsen, 1990). Yet another approach is proposed by Boudol (1989) who tried to integrate CCS into a lambda-calculus framework.

2.2.3. Process algebras

Process algebras (Baeten & Weijland, 1990) are an attempt to unify the virtues of CSP and CCS into a common framework. Matthew Hennessy (1988) proposes a semantic theory of process algebras, and a logical proof system for reasoning about them. It is shown that this system is sound and complete. The proof system relies heavily on abstract algebra.

2.2.3. π-calculus

The *π-calculus* is presented in (Milner et. al., 1992) and Milner's Turing award lecture (1993a) as an extension of CCS. It is a natural consequence of the work by Engberg and Nielsen (1986), and is sufficiently expressive to describe concurrent systems in which the topology of communication may evolve dynamically.

The π-calculus gains simplicity by removing all distinction between variables and constants. It focuses solely upon the notion of naming, a concept indispensable for concurrent communication: communication links are identified by names, and computation is represented as the communication of

names across links. The calculus introduces a notion of migration which facilitates the creation and visibility of names analogously to the substitution and conversion rules of the λ-calculus.

Because naming is essential to object-oriented programming as well, the π-calculus can be used to express the formal semantics of simple object-oriented languages (Walker, 1990), since agents are in fact objects with an independent identity. However, this work does not capture essential features of OO such as inheritance and subtyping. A typed higher-order programming language based on the π-calculus extended with values, typing, higher-order programming, results and objects is presented by Pierce, Remy & Turner (1993).

Milner (1993b) also proposed an extension of the original π-calculus in which the atomic units of communication are finite tuples of names instead of single names. A higher-order variant of the π-calculus in which not only labels but also processes may be communicated is presented by Sangiorgi in his Ph.D.-thesis (1992). Its theoretical foundations are explored by translating it into the π-calculus. Pierce & Sangiorgi (1993) extend Milner's calculus by distinguishing between the ability to read from a channel, the ability to write to a channel, and the ability to both read and write. This refinement gives rise to a natural subtype relation similar to those studied in typed λ-calculi.

2.3. Action Structures

Action structures were introduced by Milner (1992) as an algebraic framework for studying various notions of concurrent interactive behaviour, in the hope of yielding some taxonomy for these notions, and some uniformity in their representation. Its full importance with respect to OO aspects is however not yet quite understood.

Milner (1993c) shows how a subset of the π-calculus (without guarding and replication) can be treated in this uniform theory. When we work in the framework of *action calculi* (a subclass of action structures), it is even possible to describe the full π-calculus (Milner, 1993d). A graphical form of this action calculus is described in (Milner, 1994) under the name of so-called π-nets.

2.4. OC: An Object Calculus

Oscar Nierstrasz (1992, 1993) has also proposed a calculus designed to provide a formal semantics of concurrent object-based programming languages. This calculus is called OC (Object Calculus), and is based on process calculi too. OC integrates the concept of agents present in process calculi with that of functions present in lambda-calculi. Hence OC can be seen as a unification of the π- and λ-calculus. The object calculus tries to capture the following three fundamental aspects of concurrent object-oriented languages: encapsulation, active objects and composition.

2.5. Discussion

In this section I will give a discussion on all of the models mentioned above. A schematic overview of the available concurrent OO-models and their relations is depicted in figure 1.

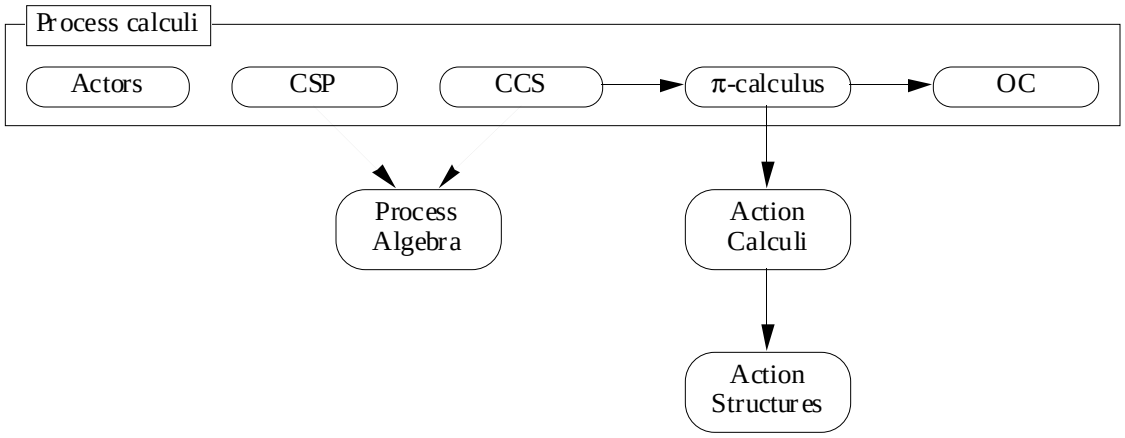

Figure 1: An overview of concurrent OO-models

It is clear that process calculi are very suited for describing OO-languages, since almost all of the models described in this section or based on some or other process calculus model. Even the asynchronous actor model can be formally represented in the form of a process calculus, as shown by Honda & Tokoro (1991).

Although the CSP and CCS model are interesting from a historical point of view, the π-calculus (which is a direct extension of CCS) offers more perspectives from an object-oriented point of view, as shown by Walker (1990) who used the π-calculus to express the formal semantics of simple object-oriented programming languages (without inheritance or subtyping). Some extensions of the π-calculus that include subtyping have beem proposed.

The π-calculus itself forms the basis of two further extensions. First of all we have Nierstrasz' Object Calculus OC that was also introduced to provide formal semantics of concurrent object-based programming languages, and is shown to capture OO aspects like encapsulation, active objects and composition. Secondly, we can distinguish action structures, which are important from a theoretical point of view, since they can be considered as the most general framework that comprises most of the others.

3. Sequential OO-models

A lot of authors share the opinion that it is more useful first to take a look at non-concurrent models for OO, because this domain already contains enough problems. Once such a formalism can be built in a satisfactory way, one can try to investigate whether concurrency aspects can be integrated into this model.

In the world of sequential foundations for OO, we can distinguish the models depending on whether they are extensions of λ-calculus or not.

3.1. Extensions of λ-calculus

Because originally lambda-calculus was the only important formal computational model, there have been several attempts to extend lambda-calculus to incorporate object-oriented features as well. An elaborate explanation of various extensions of the λ-calculus (starting from untyped λ-calculus, and ending with system $F^{\mathcal{Q}}_{\leq}$ of polymorphic typed λ-calculus with kinds, subtyping, existential quantification, records and recursive types) has been given by Mens (1994a). An overview of the most important object-oriented extensions of lambda-calculus is presented below, followed by a concluding section about their interrelations.

3.1.1. λ-calculus with records

Luca Cardelli (1988) extended λ-calculus with records including subtyping in order to give a formal semantics to multiple inheritance. Cardelli & Mitchell (1991) developed a calculus of record operations, allowing records to be extended or truncated by functions. This provided a formal treatment of certain object-oriented capabilities in a statically typed functional framework. This work was extended with subtyping by Cardelli (1994).

A similar approach was followed by Michael Wand (1991) who also proposes a lambda calculus with records, including a record concatenation operator. He shows that the type inference problem is decidable in this calculus. Moreover it is demonstrated how a simple model of object-oriented programming, including hidden instance variables and multiple inheritance, may be coded in this calculus. In contrast to (Cardelli, 1988), his model can deal with records of indefinite width.

3.1.2. λ&-calculus

Castagna, Ghelli & Longo (1994) introduced λ&-calculus as a kernel functional language to study some features of object-orientedness such as subtyping, class-based inheritance, multiple inheritance and message passing. This calculus is a simple extension of typed λ-calculus. Just like in λ-calculus, messages are treated as functions, and message sending is basically function application. The practical use of λ&-calculus is illustrated by Castagna (1993b) who developed a meta-language to prove properties of a typed object-oriented language.

A distinctive feature of l&-calculus is that functions can be overloaded by adding different "pieces of code". The piece of code to execute when the function is applied, depends on the type of the argument. In this way, we obtain a kind of multi-methods approach, similar to the CLOS-language. More specifically, a λ&-function has many branches, each corresponding to a method in an object-oriented language.

Function application chooses a branch depending on the (dynamic) class of the argument objects, yielding a notion of dynamic binding. The class of a generic function is the set of classes of the branches that constitute it. To ensure that there is always a "best fitting" branch for a particular invocation, there are some (intuitively acceptable) restrictions on the methods that can be combined in a generic function.

3.1.3. Calculus of constructions

Yet another extension of the λ-calculus includes the possibility to build type constructors (i.e. types dependent on other types), or even dependent types (types dependent on terms). In such systems, the usual distinction between types and values, and between type checking and computation, becomes blurred. The most complex system including all of these features is Coquand and Huet's calculus of constructions (1988), a higher-order typed functional calculus. Luo (1989) extended this calculus to the Extended Calculus of Constructions (ECC) by adding to the theory an infinite, fully cumulative type hierarchy and also so-called strong sums.

3.1.4. System F

An extension often used together with records (although records are not really essential), is to allow λ-expressions to be parametrised by their type. This is called the second order λ-calculus, or the polymorphic Girard-Reynolds λ-calculus, also referred to as system F (Reynolds, 1990). It allows abstraction on types (terms dependent on types). In this way we can deal with polymorphic functions.

3.1.5. System $F_{\leq}$: bounded quantification

System F has been developed further into systems $F_{\leq}$ with bounded polymorphism (also called bounded quantification) by Cardelli et. al. (1985, 1994). The type parameter of a second order expression is not allowed to range over the universe of all types, but only over a restricted subset of types. This is a natural extension of type system F, allowing us to reason about polymorphic functions in the context of subtyping.

Initially there were some problems with bounded (universal) quantification, as pointed out by Pierce (1991a), who showed that bounded quantification lacked the important syntactic property of decidability. Katiyar et. al. (1992) give a partial solution to that problem by showing that the subtyping problem is decidable over a fairly restricted subset. An even better solution is presented by Castagna & Pierce (1994) by means of the so-called "decidable" bounded quantification. The model proposed there has a natural semantic interpretation, enjoys a number of important properties that fail in $F_{\leq}$, and includes all of the programming examples for which $F_{\leq}$ has been used in practice.

Castagna (1993a) has extended $F_{\leq}$ to $F^{\&}_{\leq}$, by adding functions that dispatch on different terms according to the type they receive as argument. In other words, the explicit parametric polymorphism of $F_{\leq}$ is enriched

by an explicit multi-methods like "ad hoc" polymorphism. In this respect, system $F^{\&}_{\leq}$ can be seen as the unification of $F_{\leq}$ and $\lambda\&$-calculus.

Finally, in (Katiyar et. al., 1994) a system based on bounded universal and existential quantification is used to provide a framework for a prototyping language, RAPIDE, which supports the OOP and ADT styles of programming. Bounded existential quantification may be understood as both hiding a type and at the same time stating requirements on of what form the hidden type should be.

3.1.6. F-bounded quantification

A disadvantage of the previous model is that bounded polymorphism fails to provide adequate typings for object-oriented systems, because it cannot handle recursive types. A partial answer to that problem has been proposed by Canning et. al. (1989) under the name F-bounded quantification, a specialised form of second-order quantification. According to Katiyar (1992), F-bounded quantification is an improvement over bounded quantification with respect to recursive type definitions and subtyping. An alternative approach to solve the problems related to recursion is proposed by Mitchell et. al. (1993), by defining type systems that directly include recursive types.

F-bounded quantification contains a form of polymorphism appropriate to the task of defining functions that operate uniformly on objects of various classes. Moreover, the type variable introduced by a quantifier may appear free in its bound. The model provides a basis for typed polymorphic functions in object-oriented languages. For example, Bruce & Mitchell (1992) have developed a semantic model that encompasses all type phenomena currently used to model OO concepts by record calculi. More specifically, their paper synthesises and unifies several techniques for constructing semantic models for languages with subtyping, recursive types, and polymorphism (such as F-bounded polymorphism).

A last interesting point to notice about F-bounded quantification is that it has already been used by Bruce (1993) to construct a useful (you can actually program in it) object-oriented language (called TOOPL), complete with denotational semantics. Together with Robert van Gent (1993), this work has been extended into TOIL: a Type-safe Object-oriented Imperative Language (Bruce & van Gent, 1993). In TOIL, methods in a subtype can have any subtype of the corresponding method of the supertype. TOIL can be type-checked in a modular fashion, and inherited methods are type-safe.

3.1.7. System F^{ω}

A direct extension of system F is system F^{ω}. The difference with F is that in F^{ω} higher-order connectives are allowed, i.e. it is possible to build type-constructors (just like in the calculus of constructions) and quantification over such type-constructors is allowed. System F^{ω} was first proposed by Girard (1972).

3.1.8. System $F^{\omega}_{\leq}$

One of the most general object-oriented extensions of λ-calculus is system $F^{\omega}_{\leq}$ that describes a higher-order polymorphic lambda-calculus with subtyping. $F^{\omega}_{\leq}$ can either be considered as an extension of $F_{\leq}$ with higher-order connectives, or as an extension of F^{ω} with subtyping.

There are several models for OO based on $F^{\omega}_{\leq}$. A first one was presented by Mitchell (1990). Another one is proposed by Pierce & Turner (1994), where system $F^{\omega}_{\leq}$ with records, existential quantification and recursive types is used as a simple calculus to model encapsulation (via type abstraction), message passing, subtyping and class-based inheritance. Pierce (1993a) uses the same model for the construction of a delegation-based language with encapsulation of object states with their methods by existential types. Steffen & Pierce (1994) have developed a fundamental metatheory for $F^{\omega}_{\leq}$.

Hofmann & Pierce (1993) present an abstract framework unifying both encodings of objects based on recursive record types (Cardelli, 1994) and encodings based on existential types (Pierce & Turner, 1994). It can be used to give a type-theoretic characterisation of some basic mechanisms of object-oriented programming: objects, methods, message passing, subtyping and encapsulation (via procedural abstraction). Note that this model doesn't handle inheritance. However, it can be shown that, once the fundamental mechanisms of encapsulation and subtyping are accounted for and their interaction is properly handled, inheritance arises as a programming idiom completely within the resulting type theory, in a way similar to (Pierce & Turner, 1994).

3.1.9. Intersection types

Intersection types make it possible to state that a variable has a finite number of types at the same time. The information expressed by an intersection type is necessarily stronger than any member type of the intersection. The use of intersection types provides a direct basis for subtyping and polymorphism, by using the ordering of types induced by the various intersection types.

Benjamin Pierce (1991b, 1993b) tried to integrate the notion of intersection types in the system $F_{\leq}$ of bounded polymorphism. He goes even one step further, by also looking at the dual notion of intersection types, namely union types (1991b, 1991c). They allow a restricted form of abstract interpretation to be performed during type-checking.

Compagnoni & Pierce (1993) look at multiple inheritance via intersection types in system F^{ω}. Compagnoni (1994) shows that subtyping in this extension of F^{ω} is decidable. A direct consequence of this result is that subtyping in $F^{\omega}_{\leq}$ is decidable too.

Knowing that intersection types can be integrated in both system $F_{\leq}$ and F^{ω}, one could conclude a similar result for system $F^{\omega}_{\leq}$. However, adding intersection types to $F^{\omega}_{\leq}$ is not immediate since intersection types have their associated reduction rules which interfere non-trivially in the process of subtyping.

All the models grounded in system F are based on Church typing. This means that types are provided by the programmer, and the role of the type system is mainly to do type-checking. In other words, terms always have to be written together with their type. There is however another possible approach, called Curry typing, where type information is automatically extracted from the terms of the languages. Consequently, programmers do not need to explicitly give the type of a certain expression, because the type can be implicitly deduced using a type inference algorithm, embodying a system of inference rules. An example of this alternative approach is the so-called "Curry System with Intersection Types" (Cardone & Coppo, 1990). Another extension of the basic Curry system is described in the same paper, by introducing recursive types, which produce a notation for recursive equations over types.

3.1.10. $\lambda N^{\downarrow}_{\&}$-calculus: lambda-calculus with names, combinations and alternations

The $\lambda N^{\downarrow}_{\&}$-calculus was proposed by Dami (1994) in his Ph.D.-thesis as a formal foundation for modelling object-oriented constructs. The main features of the calculus are: interaction by names, unification of types and values, and operators for combinations and alternations of terms. A large range of object-oriented features such as inheritance, self reference, subtyping and encapsulation can be modelled in this calculus.

In an early stadium, this calculus was called the HOP-calculus (Hierarchical Objects with Ports). Its syntax and operational semantics are presented, together with numerous programming examples, in (Dami, 1993). Later on, Laurent Dami showed that this calculus can be regarded as an extension of the "de Bruijn λ-calculus" (de Bruijn, 1972) with names, combinations and alternations.

3.1.11. Conclusion

An overview of all possible extensions of lambda-calculus discussed in this paper is depicted in figure 2. The most general ones are bounded by a thick shadowed rectangle.

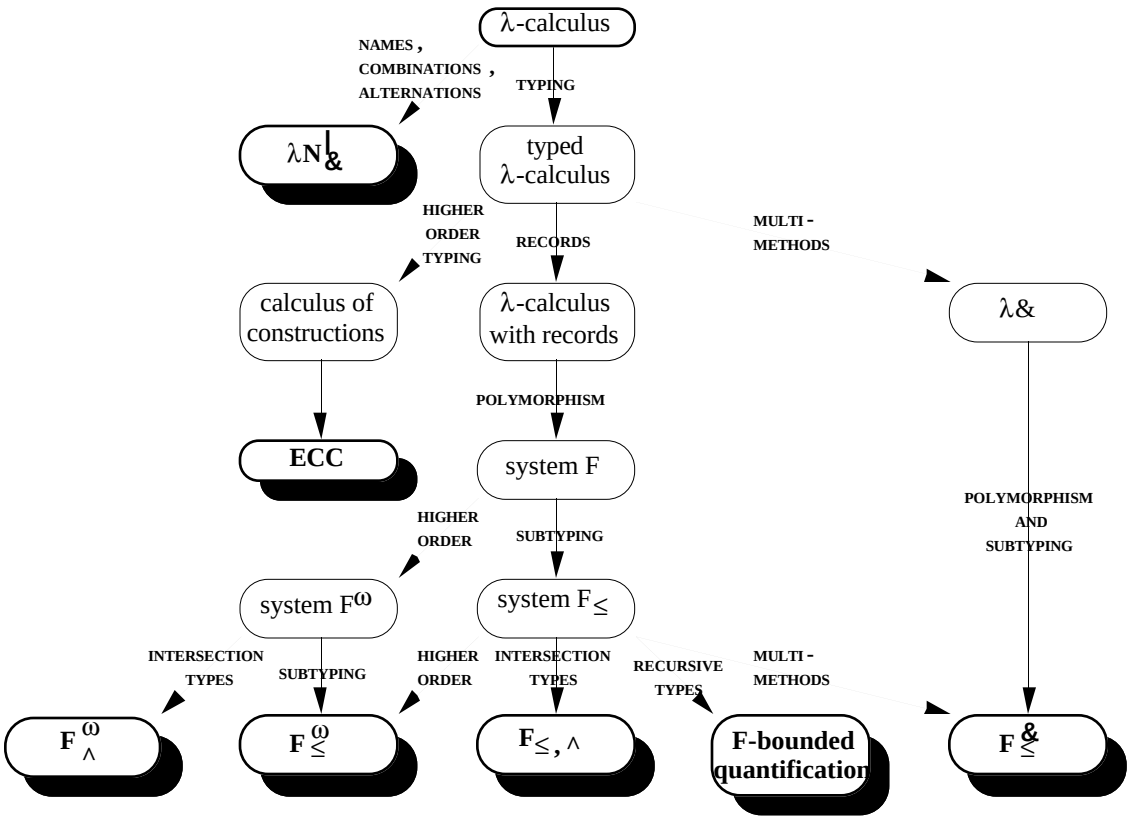

Figure 2: An overview of extensions of λ-calculus

First of all, we have system $F^{\omega}_{\leq}$, describing a higher-order polymorphic lambda-calculus with subtyping. This can be used directly as a model for OO, allowing to deal with features like objects, methods, encapsulation, message passing, subtyping, class-based inheritance, delegation, multiple inheritance etc. To make the system decidable, we could add intersection types. An alternative approach towards decidability is F-bounded quantification, but this is less general since it doesn't allow higher order functions.

An alternative model is system $F^{\&}_{\leq}$, in which we can deal with subtyping and two kinds of polymorphism, namely explicit parametric polymorphism and explicit "ad hoc" polymorphism. The latter is especially useful in a multi-methods-like approach.

Despite the importance of all of these models, I agree with Nierstrasz (1993b) on the following criticism: "While this view (the typed lambda-calculus approach) has the advantage of benefiting from a well-developed body of literature that has a great deal to say of relevance to OOP about polymorphism and subtyping, the fact that objects in real object-oriented languages change state is either ignored or dealt with in an indirect way."

There are two other interesting approaches that are relatively different of the earlier described models (but nevertheless suffer the same problems), namely the Extended Calculus of Constructions, and the $\lambda N^{|}_{\&}$-

calculus. Particularly the last one is interesting, since it can model all of the essential object-oriented features, but can also handle functional examples in an equally straightforward way, thus more or less unifying the functional and object-oriented framework.

3.2. Sequential models not based on λ-calculus

A lot of work has been carried out in attempting to explain object-oriented programming by using higher-order typed λ-calculus. These techniques are very powerful in their own right. However, according to numerous authors models based on λ-calculus are not suited for constructing a formal foundation of OO, since methods are essentially different from functions, and method execution is essentially different from function application:

> "... the standard techniques of semantics, based on higher order typed λ-calculus, are used to explain object oriented programming ... but there remains a lingering suspicion that these techniques, despite their power, somehow miss the point."
>
> (Grogono & Gargul, 1994)

> "We do not provide an operation to extract a method from an object as a function; such an operation is incompatible with object subsumption in typed calculi. Methods are inseparable from objects and cannot be recovered as functions."
>
> (Abadi & Cardelli, 1994)

> "Most of the current research on formal methods of OO consists of attempts to generalise lambda-calculus. A lot of these models suffer from difficulties in expressing the essential features of object-orientedness in a satisfying way."
>
> (Mens et. al., 1994b)

Indeed, many problems with models based on λ-calculus arise because these models are too functional. In an object-based model, functions should not be first class, since this compromises object-based encapsulation. For the same reason, message passing should be the only control structure, not composed out of more elementary building stones (such as method selection and function application). In other words, message passing should be atomic.

Most formal models of OO that are not *based on* lambda-calculus, do however *subsume* the lambda-calculus, since it is very easy to model functions in an object-oriented calculus, but not necessarily the other way round.

3.2.1. OPUS: an Object-oriented Programming calculUS

Mens et. al. (1994a, 1994b) have described a calculus for objects with delegation, encapsulation and incremental modification. An early version of the calculus can be found in the PhD-thesis of Patrick Steyaert (1994). The calculus is called OPUS - which is an acronym for Object-oriented Programming calculUS - and is intended to be used as a formal foundation for modelling object-oriented concepts, and for specifying and comparing object-oriented languages. OPUS models in a direct way crucial features of OO such as objects, names, message passing, encapsulation, and incremental modification. It is relatively easy to model booleans, numerals, inheritance, different kinds of recursion and self reference in the calculus using only the basic constructs.

While the version described in (1994a) can only model a form of non-encapsulated inheritance, (1994b) shows how to deal with encapsulated inheritance. Moreover the latter paper discusses how updatable objects can be modelled via the incremental modification mechanism, since state is not directly supported in OPUS. Similarly, argument-passing and self-references can be modelled using incremental modification.

Although OPUS was developed totally independent of the $\lambda N^{\downarrow}_{\&}$-calculus (Dami, 1994), there seem to be some striking similarities. A superficial comparison of some connections and differences between both approaches has been performed by Mens (1994b).

3.2.2. A theory of primitive objects

Martin Abadi and Luca Cardelli (1994a, 1994b) have also investigated an object calculus that is patently object-oriented: it has built-in objects, methods with self, and the characteristic semantics of method invocation and override. Moreover, the calculus is very simple, because it contains only four syntactic forms (variables, objects, method invocation and method override). Nevertheless the calculus is very expressive: it can encode the lambda-calculus, and it can express all kinds of object-oriented examples in a direct way.

Just as objects subsume functions, the calculus subsumes the λ-calculus. The system does not support a notion of state. However, the system does support shallow subtyping: an object of a subtype can have more methods than objects of its supertype, but the types of the arguments and results of the common methods must be identical. Encoding the calculus in a recognised λ-calculus in a way that preserves subtyping remains an open problem.

Abadi & Cardelli (1994a) start with giving an untyped version of the calculus, with corresponding semantics. Next, the calculus is extended by adding first-order type systems and an equational theory of objects, and this extended calculus is powerful enough to encode the simply typed lambda-calculus with fixpoints. Later on, object subtyping and recursively defined object types are introduced. The latter leads

to some problems, but these are resolved by introducing a second-order type system in (1994b), containing a *Self* quantifier. Also for this second-order calculus a semantics is given.

3.2.3. A graph semantics for object-oriented programming

Grogono & Gargul (1994) have presented a formal model for OO based on the notion of graphs. Vertices of the graph represent objects, while edges represent links between the objects. The model proposed is class-based, and includes features like object identity, local state and dynamic binding. It is shown how this model can be used as semantics for object-oriented programming languages with state and recursion. More details can be found in the technical report of Grogono (1994).

3.2.4. An ontological foundation for OO

An entirely different approach towards formal models for OO is suggested by Yair Wand (1989). He proposes to deal with objects from an ontological standpoint. (Ontology is the branch of philosophy that deals with modelling the existence of things in the world.) The proposed formalism supports features such as encapsulation, independence, persistence and inheritance. However, the notions of methods and message passing are not present in this approach, since these are considered to be implementation-oriented constructs. Instead the conceptual-oriented notion of *laws* for capturing the dynamics of OO is advocated.

3.2.5. Comparison

The untyped sigma-calculus proposed by Abadi & Cardelli (1994a) contains a lot of similarities with the OPUS-calculus (Mens et. al., 1994b). Although OPUS contains a lot of interesting ideas, the sigma-calculus is definitively better from a typing point of view.

- In both calculi, an object is defined as a record of methods, and there is a notion of method invocation;
- Both models distinguish a special kind of methods: fields in sigma-calculus are methods that do not perform self-references, constant methods in OPUS are methods that do not make use of their implementation details and arguments;
- Both calculi do not include state as a primitive operation;
- In both models, objects can be specialised by other objects (object subsumption), via an incremental modification mechanism in OPUS, or via method override in the sigma-calculus;
- Object-oriented concepts such as classes and inheritance are not explicitly present in both calculi, but can be modelled using more primitive constructs.

The main differences are summarised below:

- In the sigma-calculus the emphasis lies on sound typing rules for objects, while typing is an issue that is as yet not explored in OPUS;
- In OPUS there is a notion of encapsulation, while in the untyped calculus of Abadi and Cardelli only simple objects (records) are allowed;

• After creation, objects in sigma-calculus have a fixed length: objects cannot be extended with new methods, only existing methods can be overridden. In OPUS, objects can always be extended with new methods via the incremental modification operator.

• Although both calculi can deal with recursive objects via self-references, there is an important difference between both approaches: in the sigma-calculus the notion of self is primitive in the syntax (by making a difference between ordinary names and self-variables), thus making the examples easier, while in OPUS it is shown how self-references can be simulated straightforwardly using incremental modification and encapsulation.

Although the graph model of Grogono and Gargul (1994) is very interesting in its own right, it appears to contain more differences than similarities with the two approaches described above:

• The model is class-based instead of object-based;

• State is introduced as a primitive concept in the graph model, making it possible to reason about aspects like object identity, sharing and aliasing;

• Just like in the sigma-calculus, but in contrast to OPUS, there is no means to deal with encapsulation.

4. Acknowledgements

I express my gratitude to all persons that have contributed to this survey in a direct or indirect way. Special thanks go to Kim Mens who provided very useful information about system F and its extensions, and Marc Van Limberghen for proof-reading this article.

Although I tried to be as complete as possible, probably there will remain some models that are not included in this survey. My apologies for this.

5. References

Abadi, M & Cardelli, L (1994a), A Theory of Primitive Objects: Untyped and first-order systems, in TACS '94 Proceedings, LNCS, Springer-Verlag, pp. 296-320

Abadi, M & Cardelli, L (1994b), A Theory of Primitive Objects: Second-order systems, in Sannella, D, ESOP '94 Proceedings, LNCS 788, Springer-Verlag, pp. 1-25

Agha, G A (1986), Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press

Agha, G A & Hewitt, C (1985), Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism, MIT Press

Baeten, J C M & Weijland, W P (1990), Process Algebra, Cambridge University Press

Boudol, G (1989), Towards a lambda-calculus for concurrent and communicating systems, in Diaz & Orejas, TAPSOFT '89 Proceedings, LNCS 351, Springer-Verlag, pp. 149-161

Bruce, K B (1993), A Paradigmatic Object-Oriented Language: Design, Static Typing and Semantics, Journal of Functional Programming, Vol. 1, Cambridge University Press

Bruce, K B & Mitchell, J C (1992), PER models of subtyping, recursive types and higher-order polymorphism, in POPL '92 Proceedings, ACM Press, pp. 316-327

Bruce, K B & van Gent, R (1993), TOIL: A new type-safe object-oriented imperative language, to appear

Canning, P, Cook, W, Hill, W, Mitchell J C & Olthoff, W (1989), F-bounded quantification for object-oriented programming, in ACM Proceedings 4th Int. Conference on Functional Programming Languages and Computer Architectures, ACM Press, pp. 273-280

Cardelli, L (1988), A semantics of multiple inheritance, Information and Computation 76, Academic Press, pp. 138-164

Cardelli, L (1994), Extensible Records in a Pure Calculus of Subtyping, in Gunter, C A, Mitchell, J C, Theoretical Aspects of Object-Oriented Programming: Types, Semantics and Language Design, MIT Press, pp. 373-425

Cardelli, L & Wegner, P (1985), On understanding types, data abstraction, and polymorphism, ACM Computing Surveys, Vol. 17, No 4, ACM Press, pp. 471-522

Cardelli, L & Mitchell, J C (1991), Operations on records, Mathematical Structures in Computer Science 1, pp. 3-48

Cardelli, L, Martini, S, Mitchell, J C & Scedrov, A (1994), An extension of system F with subtyping, in TACS '91 Proceedings, Information and Computation, Vol. 109, Academic Press, pp. 4-56

Cardone, F & Coppo, M (1990), Two extensions of Curry's Type Inference System, Logic and Computer Science, Academic Press, pp. 19-75

Castagna, G (1993a), $F^{\&}_{\leq}$, Integrating parametric and 'ad hoc' second order polymorphism, in Proc. 4th International Workshop on Database Programming Languages, Springer-Verlag, pp. 335-355

Castagna, G (1993b), A meta-language for typed object-oriented languages, in Shyamasundar, R K, Proc. 13th Conf. on Foundations of Software Technology and Theoretical Computer Science, LNCS 761, Springer-Verlag

Castagna, G & Pierce, B C (1994), Decidable Bounded Quantification, in POPL '94 Proceedings, ACM Press

Castagna, G, Ghelli, G & Longo, G (1994), A calculus for overloaded functions with subtyping, Information and Computation, Academic Press, to appear

Clinger, W D (1981), Foundations of Actor Semantics, Tech Report AI-TR-633, MIT AI Lab

Compagnoni, A B (1994), Subtyping in $F^{\omega}_{\wedge}$ is decidable, Tech Report ECS-LFCS-94-281, University of Edinburgh

Compagnoni, A B & Pierce, B C (1993), Multiple Inheritance via Intersection Types, Tech Report ECS-LFCS-93-275, University of Edinburgh

Coquand, T & Huet, J (1988), The Calculus of Constructions, Information and Computation 76, Academic Press, pp. 95-120

Dami, L (1993), The HOP Calculus, in Tsichritzis, D, Visual Objects, Centre Universitaire d'Informatique, University of Geneva, pp. 151-212

Dami, L (1994), Software Composition: Towards an Integration of Functional and Object-Oriented Approaches, Ph.D.-thesis, University of Geneva

de Bruijn, N G (1972), Lambda-calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation with application to the Church-Rosser theorem, Indag. Mat., Vol. 34, No 5, pp. 381-392

Engberg, U & Nielsen, M (1986), A Calculus of Communicating Systems with Label-passing, Report DAIMI PB208, Computer Science Department, University of Aarhus

Girard, J-Y (1972), Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur, These D'Etat, Université Paris VII

Grogono, P (1994), Graph Semantics for Object Oriented Programming, Technical Report, Department of Computer Science, Concordia University

Grogono, P & Gargul, M (1994), A Graph Model for Object Oriented Programming, ACM SIGPLAN Notices, Vol. 29, No 7, ACM Press, pp. 21-28

Hailpern, B & Nguyen, V (1987), A Model for Object-Based Inheritance, Research Directions in Object-Oriented Programming, MIT Press, pp. 147-164

Hennessy, M (1988), Algebraic Theory of Processes, MIT Press

Hewitt, C (1977), Viewing Control Structures as Patterns of Passing Messages, Journal of Artificial Intelligence, Vol. 8, pp. 323-364

Hoare, C A R (1978), Communicating Sequential Processes, Communications of the ACM, Vol. 21, No 8, ACM Press, pp. 666-677

Hoare, C A R (1985), Communicating Sequential Processes, Prentice Hall

Hofmann, M & Pierce, B C (1993), A Unifying Type-Theoretic Framework for Objects, Journal of Functional Programming, Cambridge University Press

Honda, K & Tokoro, M (1991), An Object Calculus for Asynchronous Communication, in America, P, ECOOP '91 Proceedings, LNCS 512, Springer-Verlag, pp. 133-147

Katiyar, D (1992), Subtyping {F}-bounded types, in ANSA Workshop on F-bounded quantification, Cambridge

Katiyar, D & Sankar, S (1992), Completely bounded quantification is decidable, ACM SIGPLAN Workshop on ML and its Applications, ACM Press, pp. 68-77

Katiyar, D, Luckham, D & Mitchell, J C (1994), A type system for prototyping languages, in POPL '94 Proceedings, ACM Press, pp. 138-150

Luo, Z (1989), ECC, an extended calculus of constructions, in Proc. 4th IEEE Symposium on Logic in Computer Science

Mens, K (1994a), An introduction to Polymorphic Typed lambda-calculus, Tech Report vub-tinf-94-01, Computer Science Department, Vrije Universiteit Brussel

Mens, K (1994b), A Comparison of Formal Object-oriented Models, Graduate Thesis, Computer Science Department, Vrije Universiteit Brussel

Mens, T, Mens, K & Steyaert, P (1994a), OPUS: A Formal Approach to Object-Orientation, in Naftalin, M, Denvir, T, Bertran, M, FME '94: Industrial Benefits of Formal Methods, LNCS 873, Springer-Verlag, pp. 326-345

Mens, T, Mens, K & Steyaert, P (1994b), OPUS: a Calculus for Modelling Object-Oriented Concepts, in OOIS '94 Proceedings, Souhth Bank Press

Milner, R (1980), A Calculus of Communicating Systems, in Goos, G, Hartmanis, J, LNCS 92, Springer-Verlag

Milner, R (1989), Communication and concurrency, Prentice Hall

Milner, R (1992), Action Structures, Tech Report ECS-LFCS-92-249, Computer Science Department, University of Edinburgh

Milner, R (1993a), Elements of Interaction, Communications of the ACM, Vol. 36, No 1, ACM Press, pp. 78-89

Milner, R (1993b), The Polyadic Pi-calculus: a Tutorial, Logic and Algebra of Specification, Springer-Verlag, pp. 203-246

Milner, R (1993c), Action Structures for the π-calculus, Tech Report ECS-LFCS-93-264, Computer Science Department, University of Edinburgh

Milner, R (1993d), Action calculi, or syntactic action structures, in MFCS '93 Proceedings, LNCS, Springer-Verlag

Milner, R (1994), Pi-nets: a graphical form of π-calculus, in Sannella, D, ESOP '94 Proceedings, LNCS 788, Springer-Verlag, pp. 26-42

Milner, R, Parrow, J & Walker, D (1992), A Calculus of Mobile Processes, Parts I and II, Information and Computation, Vol. 100, Academic Press, pp. 1-77

Mitchell, J C (1990), Toward a Typed Foundation for Method Specialization and Inheritance, in POPL '90 Proceedings, ACM Press, pp. 109-124

Mitchell, J C, Honcell, F & Fisher, K (1993), A lambda-calculus of objects and method specialization, in Proceedings 8th IEEE Symp. Logic in Computer Science, pp. 26-38

Nierstrasz, O (1992), Towards an Object Calculus, in Tokoro, M, Nierstrasz, O, Wegner, P, Proceedings of the ECOOP '91 workshop on object-based concurrent computing, LNCS 612, Springer-Verlag, pp. 1-20

Nierstrasz, O (1993a), Composing Active Objects - The next 700 concurrent object-oriented languages, Research Directions in Concurrent Object Oriented Programming, MIT Press, pp 151-171

Nierstrasz, O (1993b), Regular Types for Active Objects, in Paepcke, A, OOPSLA '93 Proceedings, ACM SIGPLAN Notices, Vol. 28, No. 10, ACM Press, pp. 1-15

Pierce, B C (1991a), Bounded Quantification is Undecidable, Tech Report, Computer Science Department, University of Edinburgh

Pierce, B C (1991b), Programming with Intersection Types and Bounded Polymorphism, Ph.D.-thesis CMU-CS-91-295, Carnegie Mellon University

Pierce, B C (1991c), Programming with Intersection Types, Union Types and Bounded Polymorphism, Tech Report, Carnegie Mellon University

Pierce, B C (1993a), A Model of Delegation Based on Existential Types, Working draft, Computer Science Department, University of Edinburgh

Pierce, B C (1993b), Intersection Types and Bounded Polymorphism, in Proc. Conference on Typed Lambda-Calculi and Applications, LNCS 664, Springer-Verlag

Pierce, B C, Remy, D & Turner, D N (1993), A Typed Higher-Order Programming Language Based on the Pi-Calculus, Tech Report, Computer Science Department, University of Edinburgh

Pierce, B C & Sangiorgi, D (1993), Typing and Subtyping for Mobile Processes, Technical Report, Computer Science Department, University of Edinburgh

Pierce, B C & Turner, D N (1994), Simple Type-Theoretic Foundation for Object-oriented Programming, Journal of Functional Programming

Reynolds, J C (1990), Polymorphic lambda-calculus, Logical foundations of Functional Programming, Addison-Wesley, pp. 75-86

Sangiorgi, D (1992), Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms, Ph.D.-Thesis, Computer Science Department, University of Edinburgh

Steffen, M & Pierce, B C (1994), Higher-Order Subtyping, Tech Report ECS-LFCS-94-280, University of Edinburgh. Also in: IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET), 1994

Stein, L A, Lieberman, H & Ungar, D (1993), A Shared View of Sharing: The Treaty of Orlando, in Kim, W, Lochovsky, F H, Object-Oriented Concepts, Databases and Applications, ACM Press, pp. 31-48

Steyaert, P (1994), Open Design of Object-Oriented Languages, a Foundation for Specialisable Reflective Language Frameworks, Ph.D.-thesis, Computer Science Department, Vrije Universiteit Brussel

Thomsen, B (1990), Calculi for higher-order communicating systems, PhD thesis, Imperial College, London University

van Gent, R (1993), TOIL: An imperative type-safe object-oriented language, Williams College Senior Honors Thesis

Walker, D (1990), Pi-calculus Semantics of Object Oriented Programming Languages, Tech Report ECS-LFCS-90-122, Computer Science Department, University of Edinburgh

Wand, Y (1989), A Proposal for a Formal Model of Objects, in Object-Oriented Concepts, Databases and Applications, ACM Press, pp. 537-559

Wand, M (1991), Type Inference for Record Concatenation and Multiple Inheritance, in 4th IEEE Symposium on Logic in Computer Science, Information and Computation, Vol. 93, Academic Press, pp. 1-15