BRNO UNIVERSITY OF TECHNOLOGY

FACULTY OF INFORMATION TECHNOLOGY

Bozetechova 2
612 66 Brno
Czech Republic

Tel: +420 54114-1145
Email: research@fit.vutbr.cz
http://www.fit.vutbr.cz/research

Technical report UIFS FIT VUT/2006

Inheritance of specifications in the calculus of functional objects [preliminary report]

ONDREJ RYSAVY
Brno University of Technology, Faculty of Information Technology
Bozetechova 2, 612 00 Brno, Czech Republic
rysavy@fit.vutbr.cz

Abstract. Solid theoretical foundation of object-oriented paradigm have been developed for both functional and imperative programming languages. Although type theory contains functional programming language and offers rich specification and reasoning capabilities the similar foundation is not so evident despite the presence of flavor of object orientation in many other formal methods.

The idea of the present work is straightforward. Object type consists of object's interface signature specification and accompanied specification in form of logical proposition. An object value is a collection of operations working on an internal state and a proof of correctness of the implementation. In this manner, certain principles of object-orientation can be also applied to accompanied proofs, namely inheritance, late binding, and encapsulation.

Paper introduces a new calculus that features object notion as a primitive construction allowing for quite simple presentation. Object type constructor combines existential and recursive types akin to Self type of Abadi and Cardelli. To simplify the presentation and to avoid introducing general fixed point constructor a special sort of expressions is introduced – boxed expressions. A boxed expression is an expression annotated with a variable that with suitable defined substitution operation allows us to encode a restricted form of self application and positive methods.

Key words: Abstract data types, formal definitions and theory, object-oriented languages, object types, program construction, type theory.

Inheritance of specifications in the calculus of functional objects [preliminary report]

ONDREJ RYSAVY
Brno University of Technology, Faculty of Information Technology
Bozetechova 2, 612 00 Brno, Czech Republic
rysavy@fit.vutbr.cz

Abstract. Solid theoretical foundation of object-oriented paradigm have been developed for both functional and imperative programming languages. Although type theory contains functional programming language and offers rich specification and reasoning capabilities the similar foundation is not so evident despite the presence of flavor of object orientation in many other formal methods.

The idea of the present work is straightforward. Object type consists of object's interface signature specification and accompanied specification in form of logical proposition. An object value is a collection of operations working on an internal state and a proof of correctness of the implementation. In this manner, certain principles of object-orientation can be also applied to accompanied proofs, namely inheritance, late binding, and encapsulation.

Paper introduces a new calculus that features object notion as a primitive construction allowing for quite simple presentation. Object type constructor combines existential and recursive types akin to Self type of Abadi and Cardelli. To simplify the presentation and to avoid introducing general fixed point constructor a special sort of expressions is introduced – boxed expressions. A boxed expression is an expression annotated with a variable that with suitable defined substitution operation allows us to encode a restricted form of self application and positive methods.

Key words: Abstract data types, formal definitions and theory, object-oriented languages, object types, program construction, type theory.

ACM CCS Categories and Subject Descriptors: D.2.4 [Software/Program Verification]: Correctness proofs; D.3.3 [Programming Languages]: Language Constructs and Features; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; F.3.3 [Studies of Program Constructs]: Type structure; F.4.1 [Mathematical Logic and Formal Languages]: Proof theory.

1. Introduction

Pragmatic application of formal methods requires the ability to tackle large specifications. Many formal tools feature powerful structuring techniques including object-oriented ones that simplifies a problem by decomposing it into abstract data types, modules, or objects.

Studying formal foundation of object-oriented programming languages has deserved much attention and led to development of various formal calculi

October 6, 2006.

(e.g. Hofmann and Pierce [1992], Fisher *et al.* [1994], Pierce and Turner [1994], Abadi and Cardelli [1996], Crary [1999]). Significantly less such results are available for type theory as this formal system relies on very strict meta-theoretical properties that are often incompatible with usual properties of most object calculi. The present work attempts to develop a calculus of type theory that incorporates constructor for abstract data types that feature notion of encapsulation, inheritance, positive self methods, and late binding. It allows to exploit object-oriented structuring mechanism for specification and verification in the form of weak specifications in formal system of type theory. A complete specification consists of object signature and companion proposition stating the required properties. Objects are supplied with proofs of the correctness of their implementations that allows applying inheritance to proof components as well. This technique is known as deliverables (Burstall and McKinna [1993]). Novel type constructor of object types combines schemes of existential and recursive types. To express an object value with positive methods a boxed expressions are placed at positions where objects need to be reconstructed. A boxed expression is a term of calculus that wraps another term and displays a variable annotation. Adding a suitable substitution operation for this kind of terms allows to implement positive methods in an object.

The remainder of the paper is organized as follows: Next section contains motivation for development of a formal calculus. It reviews object-oriented concepts that are considered thorough the paper. These all are standard and very basic ones from the viewpoint of object-oriented programming. Nevertheless, the aim is to prepare an intuitive informal framework for further formal development. Section 3 gives the definition of a formal calculus of functional objects including type inference rules and a list of important meta-theoretical properties. In the concluding Section 4 related and further work is discussed.

2. Object-oriented programs

In this section some motivating examples are presented that helps to describe the intended form of object data structures formalized later in the calculus. The developed calculus is rather simple but allows to express a spectrum of object-oriented notions. For the purpose of simple presentation the syntax of a notional object-oriented functional programming language is used for given examples. Roughly characterized, this language mixes the syntax of pseudo-notations used by Abadi and Cardelli [1996, ch.1], fragment of OCaml language (Rémy [2002]) and Coq vernacular (Huet *et al.* [2006]).

2.1 Object Types

Object types represent signatures for object constructions. They either can only exploit its interface in the form of collection of methods (Example 1), or

they can be constructed such that also an internal state is advertises (Example 2) to public. In the context of the present work, an interface is considered to be a collection of methods that can only be called. An update operation is possible through the call of an appropriate method that internally performs update of the object's state during the regular evaluation. The difference between concrete and abstract representation is similar to that between concrete and abstract data types (Mitchell and Plotkin [1988]). Tn the present work an abstract object representation is due to Pierce and Turner [1993], where existential types are used to hide internal state. In the original work existential types allows for eliminating recursive types, which is not the case here as it is to be seen later.

Example 1. (Abstract cell object type) An object type describing storage-cell objects equipped with two methods named get and set is declared as following:

```
ObjectType Cell is
  method get : int;
  method set : int -> Cell;
end;
```

Example 2. (Concrete cell object type) A variation of storage-cell object type exploiting an integer field named `contents` that represents its representation state is declared in this example.

```
ObjectType Cell' is
  var contents : int;
  method get : int;
  method set : int -> Cell';
end;
```

In the rest of the paper, the attention is to be paid mainly to abstract version of object types. Next subsection outlines a form of object values.

2.2 Objects

Object terms consist of implementations of interface and definitions of their internal structures such that they conform to the given object signature. Example 3 shows cell structure which representation type is `int`. Only the body of `set` method deserves providing a bit of information here. The construction `{< x.contents := n >}` expresses that the result of the method is modified object `cell` in which contents is updated to a value carried by variable `n`. This construction is almost identical to update construction in language OCaml.

Example 3. An object expression describing a possible implementation of a storage-cell objects having an integer field named contents, and two methods named get and set, can have the following declaration:

```
object cell : Cell is
  val contents : int := 0;
  method get : int := this.contents;
  method set (n : int) := {< this.contents := n >};
end;
```

Type checker should be able to infer type annotation for the object such that `cell : Cell`. In this simple example it is a trivial task.

2.3 Binary Methods

Representation of binary methods is always tricky part in the settings of typed programming languages (e.g. Mitchell and Plotkin [1988], Hickey [1996], Müller [2001], Kopylov [2004]). In many systems it is impossible to formally define a mechanism for binary method representation that would work reasonably well mainly because of the presence of inheritance or privileged access. In the case of type theory this is a significant problem that often leads to the elimination of binary methods from the system (c.f. Hofmann *et al.* [1998]).

Basically, there are two kinds of problems with binary methods. Firstly, privilege access to object members can avoid binary method to read internal structure of the objects. Secondly, the subtyping relation in inherited objects may be easily broken if attempt to adjust subtyping for binary methods is made. The present object representation naturally suffers with both kind of obstacles toward binary method adoption. Moreover it seems that the calculus works satisfiable for objects only with positives methods, i.e. methods where object type appears only in the result position. Example 4 demonstrates the case if one tries to include `equal` function as a binary method.

Example 4. (Cell with binary method) A cell object type representation with binary method `equal`. Method `equal` contains argument of type `CellEq` in a non-positive position that leads to several consequences.

```
ObjectType CellEq :=
    method get : int
    method set : int -> CellEq
    method equal : CellEq -> bool
  end.
```

One immediate consequence of non-positive methods can be seen on `cellEq` object. From type `Cell` we cannot provide information that `contents` is a member of `c` object.

```
Definition cellEq : CellEq is
    val contents : int := 0
    method get : int := this.contents
    method set (n : int) := {< this.contents := n >}
    method equal(c: Cell) := equal c.contents this.contents
end.
```

The extended discussion on problem of binary methods was given by Bruce *et al.* [1996]. The possible solutions introduced therein can be with some effort adopted for the presented language as well. A detailed study of the issue related to binary methods is left open for further research.

2.4 Classes

The weak form of specifications (e.g. Bertot and Casteran [2004, ch.9]) seems to be a suitable declarative representation for classes. This kind of specifications takes an object signature and defines additional properties by means of a predicate on the type of object signature. Basically, it is possible to either put the specification part inside the object in the style of Hofmann *et al.* [1998], or it can stand beside it akin the style used by Luo [1994] for specifications of abstract data types. Second version of class representation is considered in this text. Using dependent record types (c.f. Betarte [1998]) the class declaration has the following form:

$$\begin{array}{l}\texttt{Sig} : \texttt{Type} \to \texttt{Type}\\ \texttt{Spec} : \texttt{ObjType}(\texttt{X} : \texttt{Type}).\texttt{Sig}(\texttt{X}) \to \texttt{Prop}\\ \texttt{Class}(\texttt{Sig}, \texttt{Spec}) \triangleq \sum[\texttt{str} : \texttt{ObjType}(\texttt{X} : \texttt{Type}).\texttt{Sig}(\texttt{X}), \texttt{ax} : \texttt{Spec}(\texttt{str})]\end{array}$$

Construction $\texttt{ObjType}(\texttt{X} : \texttt{Type}).\texttt{Sig}(\texttt{X})$ denotes an object type with signature $\texttt{Sig}(\texttt{X})$ that may contain type variable `X` representing self type.

Example 5. The following class declaration consists of an object type of cell objects and accompanied specification component.

```
ClassType CellClassType :=
  ObjectType Type is Cell
  ObjectSpec Spec is
    forall c : Cell, forall n : int, (x.set n).get = n
end;
```

Example 6. Class is an implementation of objects accompanied with proofs that guarantees that this implementation conforms to prescribed specification.

```
Class CellClass is
  Object of Cell is
    val contents : int := 0
    method get : int := x.contents
    method set (n : int) := {< x.contents := n >}
  Proof is
    fun  (c : Cell) => fun (n : int) => refl_eq n
end;
```

Instantiation of object is encoded as operation that extracts the first component of the class definition. According to the used syntax, an object instantiation can be written as following:

```
cellClassType.objectSig c = cellClass.Object
```

2.5 Subtyping and Inheritance

Subtyping relation and subsuming mechanism are the key factors that has to be considered when implementing inheritance feature in typed languages. The usual way of defining subtyping relation known for various lambda based calculi (e.g. Longo *et al.* [1995],Betarte [1998], Luo [1999], Luo and Soloviev [1999]) is considered and directly extended to include also object types.

EXAMPLE 7. Object type `ReCell` is a subtype of `Cell` type. The checking of this relation requires the presence of rules of record type subtyping and a variation of Sub Self rule as defined by Abadi and Cardelli [1996, p.182].

```
ObjectType ReCell is
  method get : int
  method set : int -> ReCell
  method restore : unit -> ReCell
end.
```

For the discussed object representation the usual inheritance relation provides a mechanism for defining new classes by means of extension mechanisms such that proofs of correctness given for superclasses may be utilized in subclasses too. It is known that inheritance is not equivalent to subtyping (e.g. Leavens [1989], Cook *et al.* [1990]). To simplify the situation, it is acceptable to consider that if class `A` is a subclass of class `B` then `TypeOf(a) <: TypeOf(b)` for all objects `a` instantiated from class `A` and all objects `b` instantiated from class `B`, respectively. The usual condition stating that specification of subclass implies the specification of superclass $\forall(\texttt{b} : \texttt{B}), \texttt{B.spec(b)} \implies \texttt{A.spec(b)}$, is intended to ensure that the implementation of operations in subclass `A` satisfies the specification of that operations in superclass `B` (c.f. Meyer [1997]).

EXAMPLE 8. Class `ReCellClass` defines an object that keeps a previous value and which state can be revert by calling `restore` method. A specification of this class is extended to accompany this new behavior.

```
ClassType ReCellClassType is
ObjectType type is ReCell;
ObjectSpec spec is
  forall (c:ReCell),
  CellClassType.Spec(c) /\ forall (n : int), (c.set n).restore = c;
end.
```

Class definition uses `CellClass` definition in two ways. Firstly, it takes the cell object specification and adds `restore` method and override `set` method. Secondly, it reuses proof from the superclass definition and supplement it with a proof term for the second conjunct of `ReCellClassType.spec` specification.

```
Class ReCellClass is
Object of ReCell is CellClass.Object with
```

```
  val backup : int := 0;
  method set (n:int) := {< this.backup := this.contents;
                           this.contents := n >}
  method restore : ReCell := {< this.contents := this.backup >}
Proof is
  fun  (c : ReCell) =>
    conj (CellClass.Proof c) (fun (n:int) => refl_eq c)
end.
```

Note that Coq syntax is used in proof component. Declaration `conj` comes from Coq standard library (Barras *et al.* [2006]). The considered language is expressible enough to allow us to formally check the subclass condition. In this case it is easy to see that

$$\lambda(\mathtt{c} : \mathtt{ReCell})(\mathtt{p} : \mathtt{ReCellClassType.Spec}(\mathtt{c})) \Rightarrow \mathtt{first\ p}$$
$$\in \forall(\mathtt{c} : \mathtt{ReCell}), \mathtt{ReCellClassType.Spec}(\mathtt{c}) \longrightarrow \mathtt{CellClassType.Spec}(\mathtt{c})$$

is a proof that `ReCellClass` guarantees properties of `CellClass`.

3. Calculus of functional objects

Common terminology considers functional objects to be non-mutable objects that means when an object is to be modified it returns the modified object instead of doing it in place. The more precise name for the calculus presented in this section would be a calculus of strong normalizing functional objects as the intention is to develop a typed calculus with consistent internal logic suitable for specifying and reasoning about programs, which inevitable relies on strong normalization property.

3.1 Term calcululus

The calculus of functional objects (CFO for short) is quite similar to Calculus of constructions (Coquand and Huet [1988]) and Extended calculus of construction (Luo [1989]). CFO consists of an underlaying term calculus and a set of interference rules of judgements. A term of the calculus is an expression generated by the following grammar.

A, B, M, N	$::=$	terms
	x	variable
	$Prop$	kind of propositions
	$Type_i$	kind of constructions ($i \in \omega$)
	$\Pi(x : A).B$	product type
	$\lambda(x : A).M$	functional abstraction
	$(M\ N)$	application
	$[l_i : A_i^{i\in 1..n}]$	record type
	$[l_i = M_i^{i\in 1..n}]$	record
	$(M\#l)$	field selection
	$\varsigma(x : A).B$	object type
	$\varsigma(x : A, M).N$	object abstraction
	$\langle M\rangle_x$	box
	$M!$	object use

Notions of free-variable and variable renaming are defined as usual. Further, α-convertible terms are identified. Variable substitution operation, denoted $[N/x]M$, works as expected and requires no explanation if binding in novel kinds of terms is clarified. In object abstraction $\varsigma(x : A, M).N$ variable x is bound in N but not in M. Variable binding in object type follows binding pattern of product type and strong sum type. In addition to variable substitution operation the calculus features box substitution operation, denoted $\{N/x\}M$. The novel and interesting rules of variable and box substitution operations deal with boxed terms and have the following definition:

$$\{M/x\}\langle N\rangle_y \triangleq \begin{cases} \langle\{M/x\}N\rangle_y & (x \neq y) \\ [N/x]M & (x = y) \end{cases} \qquad [M/x]\langle N\rangle_y \triangleq \begin{cases} \langle[M/x]N\rangle_y & (x \neq y) \\ [M/x]N & (x = y) \end{cases}$$

It can be seen that the meaning of a boxed term is to store a value that is used for instantiating variables in substituted term when box substitution operation is processed. The variable substitution rule only eliminates a box term leaving a content of the box. This rather technical rule is required for proof of subject reduction property.

3.2 Conversion rules

Computation semantics of the calculus relies on the contraction schemes. These are given by the following definitions:

$$(\lambda(x : A).M)\ N \rightsquigarrow [N/x]M \qquad (\beta)$$

$$[l_i = M_i^{i\in 1..n}]\#l_j \rightsquigarrow M_j \quad (j \in 1..n) \qquad (\pi)$$

$$\varsigma(x : A, M).N! \rightsquigarrow [M/x]\{\varsigma(x : A, x).N/x\}N \quad (\gamma)$$

Reduction ($\triangleright$) and conversion ($=$) relations are defined with respect to the $\beta\pi\gamma$-contraction schemes. These relations are designed such they satisfies Church-Rosser property.

PROPOSITION 1. (CHURCH-ROSSER) *Let* $M_1 = M_2$ *then there exists* M *such that* $M_1 \triangleright M$ *and* $M_2 \triangleright M$.
PROOF. *The proof of this property uses modified version of standard Martin-Lof's method. (cf. Martin-Löf [1975] or Hindley and Seldin [1987] for details).* □

EXAMPLE 9. (CONTRACTING OBJECT USE) Object use operation is evaluated by γ contraction. Both substitution operations are employed to in order to disseminate the internal state and recreate objects in positions where necessary.
Let $M \equiv [get = x, set = \lambda(n : int).\langle n \rangle_x]$ in

$$\begin{gathered}\varsigma(x : int, 5).M! \rightsquigarrow_\gamma [5/x]\{\varsigma(x : int, x).M/x\}M \equiv \\ [5/x][get = x, set = \lambda(n : int).\varsigma(x : int, n).M] \equiv \\ [get = 5, set = \lambda(n : int).\varsigma(x : int, n).M]\end{gathered}$$

3.3 Type inference rules

Formalization of the type inference system of FCO relies on general notation of context, judgement, and derivations, which provide basis for understanding the concrete inference rules of the calculus. Type rules assert the validity of a judgement on the basis of other judgements that are already known to be valid. Figure 3.1 shows the inference rules of FCO. Subtyping on terms of the calculus is encoded by subsume rule and cumulative relation. Type cumulative relation ($\preceq$) is defined separately by the following rules:

(Sub Prod)
$$\frac{A_1 = A_2 \qquad B_1 \preceq B_2}{\Pi(x : A_1).B_1 \preceq \Pi(x : A_2).B_2}$$

(Sub Record)
$$\frac{\Gamma \vdash A_1 \preceq B_1^{i \in 1..n}}{\Gamma \vdash [l_i : A_i^{i \in 1..n+m}] \preceq [l_i : B_i^{i \in 1..n}]}$$

(Sub Object)
$$\frac{A_1 \preceq A_2 \qquad B_1 \preceq B_2}{\varsigma(x : A_1).B_1 \preceq \varsigma(x : A_2).B_2}$$

Deep investigation of proof-theoretical properties of CFO has to be done. It may be possible to follow the proofs of these properties given for ECC and CC as there is a similarity of FCO with these calculi. The interesting properties are listed bellow.

PROPOSITION 2. (CORRECTNESS OF TYPING) *If* $\Gamma \vdash M \in A$, *then* $\Gamma \vdash A \in Type_i$.

Subject reduction signalizes soundness of the type systems as it gurantees that the result of a computation preserves the type.

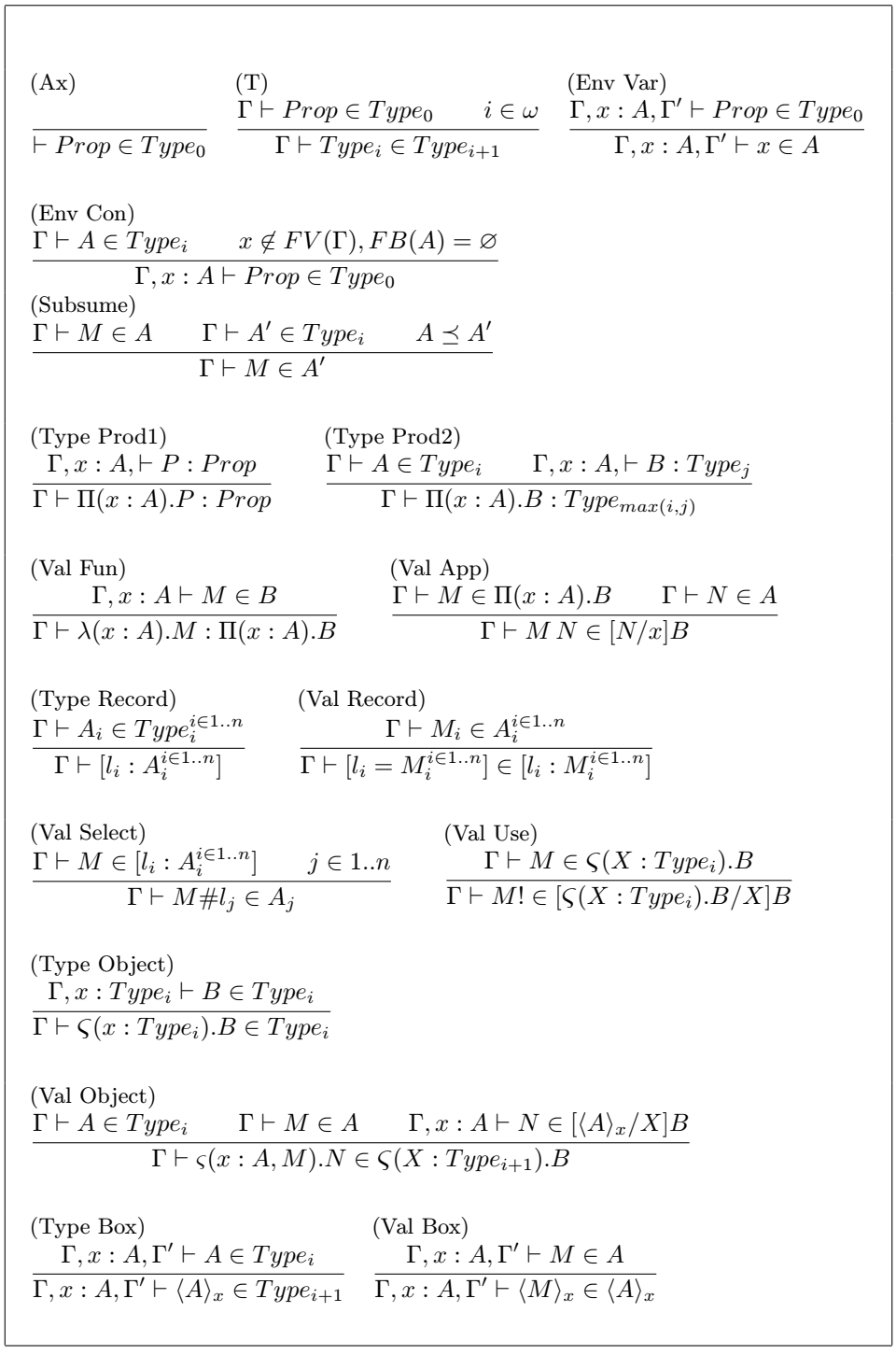

Fig. 3.1: Type inference rules of FCO

PROPOSITION 3. (SUBJECT REDUCTION) *If* $\Gamma \vdash M \in A$ *and* $M \rhd N$, *then* $\Gamma \vdash N \in A$.

A principal time is "the most precise" type that can be assigned to a term. In particular, for record type it means the type in which none of the fields was forgotten.

PROPOSITION 4. (EXISTENCE OF PRINCIPAL TYPE) *If* $\Gamma \vdash M : A$ *then exists a principal type* A' *of* M *such that* $\Gamma \vdash M : A'$ *and for any* $A'', \Gamma \vdash M : A''$ *holds* $A' \preceq A''$.

Contrary to calculi for programming, strong normalization is a key property that is required in a consistent system of type theory.

PROPOSITION 5. (STRONG NORMALIZATION) *If* $\Gamma \vdash M \in A$ *then* M *and* A *are strongly normalizable.*

Decidability of judgement inference procedure guarantees that there is an algorithm of typecheking for the calculus.

PROPOSITION 6. (DECIDABILITY) *For arbitrary* M, A, Γ *it is decidable whether* $\Gamma \vdash M \in A$.

In particular, proving strong normalization is a key basis to study the consistency of internal logic and related logical issues such as decidability and character of equality relation.

4. Conclusion

The present paper showed the possible approach to including object data types into the system of type theory with the aim of preservation of consistency of the internal intuitionistic logic. The idea of this attempt is to have object data structures as first class citizens in the calculus and to use their properties for structuring and reuse of specifications and proofs. Work in progress was presented with the following partial achievements:

- Simple object representation based on the new primitive language construct of the underlaying calculus is formalized. The new construct called boxed terms provides the key mechanism that allows for simple encoding positive methods.
- The object representation is suitable for both strong and weak specifications. Two styles for weak object specifications have been sugessted.
- In comparison to representation by Hofmann *et al.* [1998] there is no need of generic methods for manipulating with objects, and also late binding is naturally provided by the used representation that radically simplifies the whole representation. On the other side, underlaying calculus required for this encoding is more complex than the original ECC calculus.

It has to be noted that there are unenforced options that one must to choose from during the design of the calculus. The present work is just a one trace in the design space that follows the approach taken in related work to show how to construct a calculus with the primitive notion of object constructions and object types. The interesting questions might arise when one tries to take more adventure way and explore other combinations as well.

4.1 Related Work

Although there is a large body of work about semantic foundations of object-oriented programming and type systems for object-oriented programming languages the similar idea focusing on the representation of object data types in type theory is not so extensively developed:

- Hickey [1996] introduces a new type constructor, called very dependent function type, which represents certain form of recursive record type accompanied with well-foundedness conditions to avoid circularity.
- Hofmann *et al.* [1998] extends the object-model of Pierce and Turner [1993] with proof component and encodes it in ECC type theory.
- Recently, a different approach of synthesizing object calculus in logical framework has been suggested by Liquori and A.Spiwack [2004]. The aim of proposed logical framework combining logically sound first-order system and object calculus at the different level is to help with managing and reusing of proofs in logical environment rather than to develop "object-oriented" type theoretical system.

4.2 Future Work

Besides the open questions stated above there are several tasks that attract attention for further research:

- Justification of claimed properties of the calculus needs to be provided. It includes to perform classical proof work, but there is also an attempt to do some proofs formally in Coq. A calculus has been encoded as Coq theory using higher-order abstract syntax (c.f. Despeyroux *et al.* [1995], Honsell *et al.* [2002]) and several lemmas has been proved for substitution operations.
- The study of models for the presented theory is completely untouched so far. Set theoretical model of ECC is constructible in framework of ω-sets. As there is certain similarity to this calculus it is question whether it is possible to find ω-set model for FCO as well.
- Currently, an experimental computer aided tool is being implemented by the author. Its further development is desirable as it can help to better grasp the issue of abstract program development in type theory featuring object-oriented mechanisms. Having such tool is inevitable for checking large examples and counter examples.

References

Abadi, Martin and Cardelli, Luca. 1996. *A Theory of Objects*. Springer-Verlag New York, Inc.

Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J.C., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., Parent, C., Paulin, C., Sabi, A., and Werner, B. 2006. The Coq Proof Assistant Reference Manual – Version V8.1. Tech. report, INRIA.

Bertot, Yvs and Casteran, Pierre. 2004. *Interactive Theorem Proving and Program Development*. European Association for Theoretical Computer Science. Springer-Verlang, Berlin Heidelberg.

Betarte, Gustavo. 1998. *Dependent Record Types and Formal Abstract Reasoning*. PhD thesis, Chalmers University of Technology.

Betarte, Gustavo. 1998. Dependent Record Types, Subtyping and Proof Reutilization. In *Subtyping, inheritance and modular development of proofs*.

Bruce, Kim B., Cardelli, Luca, Castagna, Giuseppe, Leavens, Gary T., and Pierce, Benjamin. 1996. On binary methods. *Theory and Practice of Object Systems 1*, 3, 221–242.

Burstall, Rod and McKinna, James. 1993. Deliverables: a categorical approach to program development in type theory. In *Eighteenth Mathematical Foundations of Computer Science*, Borzyszkowski, A. M. and Sokolowski, S., Editors. Volume 711 of *Lecture Notes in Computer Science*. Springer Verlang, 32–67.

Cook, William R., Hill, Walter, and Canning, Peter S. 1990. Inheritance is not subtyping. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 125–135.

Coquand, Thierry and Huet, Gerard. 1988. The calculus of constructions. *Inf. Comput. 76*, 2-3, 95–120.

Crary, Karl. 1999. A Simple, Efficient Object Encoding using Intersection Types. In *International Conference on Functional Programming*, 82–89.

Despeyroux, Joëlle, Felty, Amy, and Hirschowitz, André. 1995. Higher-Order Abstract Syntax in Coq. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, Volume 902 of *LNCS*. Springer-Verlag, Edinburgh, Scotland, 124–138.

Fisher, Kathleen, Honsell, Furio, and Mitchell, John C. 1994. A lambda calculus of objects and method specialization. *Nordic Journal of Computing 1*, 1 (Spring), 3–37.

Hickey, Jason J. 1996. Formal Objects in Type Theory Using Very Dependent Types. In *FOOL 3: Foundations of Object Oriented Languages 3*.

Hindley, J.R. and Seldin, J.P. 1987. *Introduction to Combinators and λ-calculus*. Cambridge University Press.

Hofmann, Martin, Naraschewski, Wolfgang, Steffen, Martin, and Stroup, Terry. 1998. Inheritance of proofs. *Theor. Pract. Object Syst. 4*, 1, 51–69.

Hofmann, Martin and Pierce, Benjamin. 1992. An abstract view of objects and subtyping. Tech. Report ECS-LFCS-92-225, University of Edinburgh.

Honsell, Furio, Miculan, Marino, and Scagnetto, Ivan. 2002. The Theory of Contexts for First Order and Higher Order Abstract Syntax. *Electronic Notes in Theoretical Computer Science 62*, 116–135.

Huet, G., Kahn, G., and Paulin-Mohring, C. 2006. The coq proof assistant : A tutorial. Tech. report, Institut National de Recherche en Informatique et en Automatique.

Kopylov, Alexei Pavlovich. 2004. *Type Theoretical Foundations For Data Structures, Classes, and Objects*. PhD thesis, Cornell University.

Leavens, Gary Todd. 1989. *Verifying Object-Oriented Programs that Use Subtypes*. PhD thesis, MIT Laboratory for Computer Science.

Liquori, L. and A.Spiwack. 2004. An Object-Oriented Logical Framework [Preliminary Report]. Manuscript.

Longo, Milsted, and Soloviev. 1995. A Logic of Subtyping. In *LICS: IEEE Symposium on Logic in Computer Science*.

Luo, Zhaohui. 1989. ECC: an Extended Calculus of Constructions. In *Proceedings of IEEE 4th Annual Symphosium on Logic in Computer Science (LICS'89)*, 386–395.

Luo, Zhaohui. 1994. *Computational and Reasoning: A Type Theory form Computer Science*. Clarendon Press.

Luo, Zhaohui. 1999. Coercive Subtyping. *Journal of Logic and Computation 9*, 1, 105–130.

Luo, Zhaohui and Soloviev, Sergei. 1999. Dependent Coercions. In *Proceedings of 8th conference on Category Theory and Computer Science (CTCS'99)*.

Martin-Löf, Per. 1975. An intuitionistic theory of types: Predicative part. In *Logic colloquium '73*, H. E. Rose, J. C. Shepherdson, Editor. Amsterdam, North-Holland.

Meyer, Bertrand. 1997. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Mitchell, John C. and Plotkin, Gordon D. 1988. Abstract types have existential type. *ACM Trans. Program. Lang. Syst. 10*, 3, 470–502.

Müller, Peter. 2001. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, University of Hagen.

Pierce, Benjamin C. and Turner, David N. 1993. Object-Oriented Programming Without Recursive Types. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Charleston, South Carolina*, 299–312.

Pierce, Benjamin C. and Turner, David N. 1994. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming 4*, 2 (April), 207–247.

Rémy, Didier. 2002. Using, Understanding, and Unraveling the OCaml Language. In *Applied Semantics. Advanced Lectures. LNCS 2395.*, Barthe, Gilles, Editor. Springer Verlag, 413–537.