Chapter 1

Object-Oriented Programming in Dependent Type Theory

Anton Setzer[1]

Abstract: We introduce basic concepts from object-oriented programming into dependent type theory based on the idea of modelling objects as interactive programs. We consider methods, interfaces, and the interaction between a fixed number of objects, including self-referential method calls. We introduce a monad like syntax for developing objects in dependent type theory.

1.1 INTRODUCTION

In the conference TYPES 2003, the author gave a talk on how to represent lambda-terms in Java ([Set03]). At the end of this talk, Martin-Löf asked the question: What you have done is to represent functional programming in object-oriented programming. Can we do it the other way around as well? What he meant was: Can we represent object-oriented programming in dependent type theory?

The author has developed with P. Hancock the notion of interactive programs in dependent type theory. Objects can be considered as interactive programs: they receive requests (method calls) from the outside, and return answers to these requests to the outside. It seems to be interesting to explore the use of interactive programs in order to model objects and classes in dependent type theory.

There are two reasons why we think it is useful to model concepts from object-oriented programming in dependent type theory. On one hand this would allow

[1] Department of Computer Science, University Of Wales Swansea, Singleton Park, Swansea SA1 4PZ, UK. Email: `a.g.setzer@swan.ac.uk`. This research was supported by EPSRC grant GR/S30450/01. Part of this work was done while the author was an invited research follow at the Graduate School of Science and Technology, Kobe University, Japan.

to reason about object-oriented programs in a theorem prover based on dependent types. For instance, one can write a verified compiler for an object-oriented language. On the other hand we belief that it is interesting to develop an object-oriented programming language based on dependent types. In such a language, two different powerful programming paradigms – object-orientation and dependent types – would merge. This would give rise to powerful new types – note that one of the main contributions of both concepts is that they substantially increase the types available. By using dependent types one could write more generic object-oriented programs.One could as well write object-oriented programs with a higher degree of correctness. For instance, pre-conditions of methods could be enforced by having extra arguments, which require a proof of that pre-condition; post-conditions could be treated similarly. In general we would obtain one uniform framework in which both computing and reasoning takes place. So there is no need to introduce separate axioms for reasoning about programs, since correctness proofs are carried out in the same framework as the program, and have therefore direct access to it.

Along that route one would have to add of course at some point new language constructs to dependent type theory rather than implementing all concepts from object-oriented programming directly in it. The current paper, in which we model some object-oriented concepts in dependent type theory, serves as a first step in determining which additional language constructs are needed in order to facilitate writing object-oriented programs in dependent type theory.

In this paper we will make first steps in exploring the idea of modelling objects as interactive programs. We will cover objects, interfaces, methods and interaction between a fixed number of objects. We will see that the modelling of self-referential method calls in dependent type theory results in rather complex interactive programs.

We will as well see that with dependent type theory we obtain a higher degree of expressibility. For instance classes can have dependently typed methods, which means that the result type of a method might depend on the arguments. Because of the use of dependent types, all methods can be merged into a single method. In type theory, the body of a method is an element of a data type, and we can write generic functions, which transform this data type, and which can be used to generate method bodies.

Content of this article. In Sect. 1.2 we introduce the basic concepts of dependent type theory. This will introduce as well our notations used for working in dependent type theory. In that section we will as well introduce coalgebraic types (codata). In Sect. 1.3 we introduce interactive programs. In Sect. 1.4 we explore the idea of an object as an interactive program. In Sect. 1.5 we show how to deal with combining objects without self-referential calls. In Sect. 1.6 we will extend this by allowing objects which directly or indirectly call themselves. We will as well introduce in this section a monad like syntax for representing object code, as it occurs in standard object-oriented languages like Java. This code will then be translated into objects in dependent type theory.

Related work. There is a rich literature on using object-calculi related to the λ-calculus and on using impredicative type theories in order to assign types to type theory. A lot of material can be found in [AC96]. Pierce and Turner [PT94] use impredicative existential types of $\mathcal{F}^{\omega}_{\leq}$ in order model objects. Jacobs [Jac95, Jac98] and Reichel [Rei95] have used coalgebras in order to model objects, but they do not deal with self-referential method calls in full, which, as we will see, results in rather complex structures. Meseguer [Mes93] has indicated how to formulate concurrent objects in a rewrite system. Kiselyov and Lämmel [KL05] have modelled object-oriented concepts in Haskell. We have not found any treatment of object-oriented programming in the context of predicative dependent type theory, which makes use of the expressive power of dependent types, and we belief that the use of interactive programs in this context is new.

Acknowledgements. We would like to thank the referees for valuable comments.

1.2 BASIC CONCEPTS OF DEPENDENT TYPE THEORY

The basic type constructions used from standard dependent type theory are as follows:

- Dependent function types, written as $(x : A) \to B$ for the type of functions, mapping an element $a : A$ to an element of $B[x := a]$. We use standard abbreviations, such as $A \to B$ for $(x : A) \to B$ for a variable x which does not occur in B, $(x : A, y : B) \to C$ for $(x : A) \to (y : B) \to C$, etc. Elements of $(x : A) \to B$ are created by λ-abstraction $\lambda x.t$, and eliminated by application to elements of A, where application is written in functional style $(f\ a)$.

- Dependent products. For convenience, we use record notation in this article. The product of types $A_1, \ldots, A_n$ is thus written as

  $$\Sigma(\mathsf{l}_1 : A_1, \ldots, \mathsf{l}_n : A_n)$$

  Here A_i might depend on $\mathsf{l}_j : A_j$ for $j < i$, and l_i are the record selectors (sometimes called labels). Projection is written as record selection, i.e. for an element c of the type A just introduced the ith projection is written as $c.\mathsf{l}_i : A_i$. Our notation for introducing elements of type A is

  $$\mathsf{record}(\mathsf{l}_1 = t_1, \ldots, \mathsf{l}_n = t_n) \ ,$$

  where $t_i : A_i$. This is record notation for the n-tuple $\langle t_1, \ldots, t_n \rangle$.

  Occasionally, we will use as well product notation, namely $(x : A) \times B$ for the product of A and B, where B might depend on $x : A$. Elements of this type are written as pairs $\langle a, b \rangle$. We use case distinction in order to unpack such pairs.

- Algebraic types are written as follows:

  $$A = \mathsf{data}\ \mathsf{C}_1(a^1_1 : A^1_1, \ldots, a^1_{n_1} : A^1_{n_1}) \ \mid\ \cdots\ \mid\ \mathsf{C}_m(a^m_1 : A^m_1, \ldots, a^m_{n_m} : A^m_{n_m})$$

Algebraic types correspond to the least set closed under those constructors. We have the usual condition that the types of the constructor are strictly positive in the type to be introduced. (This means that in the above definition A^i_j either do not make use of A, or are of the form $B_1 \to \cdots \to B_l \to A$, where B_i do not make use of A). Furthermore, we will sometimes omit a variable a^i_j, if A^i_k for $k > j$ do not depend on it.

Elimination is defined by case distinction: Assume $a : A$, $D : A \to \mathsf{Set}$ [2] and

$$a^i_1 : A^i_1, \ldots, a^i_{n_i} : A^i_{n_i} \Rightarrow t_i[a^i_1, \ldots, a^i_{n_i}] : D\;(\mathsf{C}_i\; a^i_1 \cdots a^i_{n_i})$$

Then

$$\begin{array}{llll}\mathsf{case}\ (a)\ \mathsf{of}\ \{ & (\mathsf{C}_1\; a^1_1 \cdots a^1_{n_1}) & \longrightarrow & t_1[a^1_1, \ldots, a^1_{n_1}];\ \cdots; \\ & (\mathsf{C}_m\; a^m_1 \cdots a^m_{n_m}) & \longrightarrow & t_m[a^m_1, \ldots, a^m_{n_m}]\}\end{array}$$

is of type $(D\,a)$. Note that $t_i[\cdots]$ has a type corresponding to the branch of the case distinction. Functions defined using case-distinction can be recursive, as long as recursive function calls are made to structurally smaller elements.

- We introduce a convenient notation for the disjoint union:

  $$\begin{array}{l} C_1(x^1_1 : A^1_1, \ldots, x^1_{n_1} : A^1_{n_1}) + \cdots + C_n(x^m_1 : A^m_1, \ldots, x^m_{n_m} : A^m_{n_m}) \\ \quad := \mathsf{data}\; C_1(x^1_1 : A^1_1, \ldots, x^1_{n_1} : A^1_{n_1}) \mid \cdots \mid C_n(x^m_1 : A^m_1, \ldots, x^m_{n_m} : A^m_{n_m}) \\ A + B := \mathsf{inl}(a : A) + \mathsf{inr}(b : B) \qquad 1 + A := \mathsf{inl}() + \mathsf{inr}(x : A) \end{array}$$

  Here A^i_j should not refer to the new set being introduced. We will as well omit brackets in case one of C_i does not have any arguments.

- Furthermore, we will add coalgebraic types. A discussion on coalgebraic types in dependent type theory can be found in [HS04] (see as well [BC04]). Coalgebraic types are written as

  $$A = \mathsf{codata}\; \mathsf{C}_1(a^1_1 : A^1_1, \ldots, a^1_{n_1} : A^1_{n_1}) \mid \cdots \mid \mathsf{C}_m(a_1 : A^m_1, \ldots, a^m_{n_m} : A^m_{n_m})$$

  with the same condition on strict positivity as for algebraic types.

  – The elimination principle is case distinction. However, after applying case distinction, one does not obtain a structural smaller term which can be used in order to write terminating recursive functions.

  – Introduction is formally written as guarded recursion. However, guarded recursion is a syntactic notation, and, as pointed out in [HS04], guarded recursion is not supposed to be evaluated in full. Instead definitions by guarded recursion represent syntactic expressions introduced by the introduction rule for coalgebraic types. Essentially, guarded recursion is only

[2] See the remarks on the logical framework at the end of this section on what is meant by Set.

evaluated, if the case-distinction construct is applied to a term introduced by guarded recursion. In this case, one step of the guarded recursion is evaluated, and then, depending on the case distinction, the term is reduced further. More details can be found in [HS04].

We will in this article make use of the *logical framework*. There we have two levels of types, called, as usual in dependent type theory, Set and Type. We have $\mathsf{Set} : \mathsf{Type}$ and if $A : \mathsf{Set}$, then $A : \mathsf{Type}$. But we do not have $\mathsf{Set} : \mathsf{Set}$, otherwise the resulting type theory would be inconsistent (Girard's paradox). Both Set and Type are closed under dependent function types and dependent products. Set is as well closed under the formation of strictly positive algebraic and coalgebraic types. Especially, Set will contain the well-behaved simple types, as they occur in non-dependent functional programming, and dependent versions of these. Set is what would outside dependent type theory be denoted as "type". With the logical framework we have one layer on top of it. This allows to assign for instance the type $\mathsf{Set} \to \mathsf{Set} : \mathsf{Type}$ to operations mapping elements of Set to Set, i.e. to type-manipulating operations.

1.3 INTERACTIVE PROGRAMS IN DEPENDENT TYPE THEORY

The main idea for representing object-oriented programming in dependent type theory is that objects are to be considered as interactive programs. Let us first review how interactive programs can be represented in dependent type theory.

In [HS99, HS00b, HS00a, HS04] the author has developed together with Peter Hancock a theory of interactive programs in dependent type theory (see as well [Han00, HH06] for some related work by Hancock and Hyvernat). An interface for a stateless (or non-state-dependent) interactive program[3] is given by a set of commands $\mathsf{C} : \mathsf{Set}$ and a set of responses depending on a command $c : \mathsf{C}$, i.e. we have

$$\mathsf{Interface} = \Sigma(\mathsf{C} : \mathsf{Set}, \mathsf{R} : \mathsf{C} \to \mathsf{Set}) : \mathsf{Type}$$

There are two kinds of interactive programs to be associated with an element $I := \mathsf{record}(\mathsf{C} = C, \mathsf{R} = R)$ of $\mathsf{Interface}$:

- The set of client side programs. These are programs which make a call $c : C$ to the real world, such as to output a string to the console, or requesting a string input from the keyboard. They then receive a response $r : R\ c$ from the real world (e.g. a success message in case of the writing of a string to console, or the string typed in by the user in case of the request for a string), and depending on it, execute the next command. This loop is then repeated until the program determines that it has terminated.

  If IO is the set of client side programs, then this loop can be represented in type theory by a function $f : \mathsf{IO} \to ((c : C) \times ((R\ c) \to \mathsf{IO}))$. Executing this loop is

[3] Stateless refers to externally observable states – interactive programs for stateless interfaces can have internal states.

necessarily an external procedure, which is outside type theory. If we execute this loop for $p : \mathsf{IO}$, this procedure will compute the command $c := \pi_0(f\ p)$ to be issued. It will then execute c in the real world, and receive a response r. Then it will compute $p' := \pi_1(f\ p)\ r$ and continue executing p'.

- The set of server side programs. These programs receive a command $c : C$ from the real world, and respond to it with a response $r : R\ c$. Then the program is ready to get the next command, etc.

There exist as well programs like proxy servers with a combination of a server-side and client-side behaviour. Such combinations will occur on the object level later in this article.

In a monadic version (see below for the monad operations), we obtain two sets. Both will depend on an answer set A and a fixed interface I as above:

- The set of client side programs, which possibly terminate, and if they terminate terminate by returning an element of type A.
- The set of server side programs, which possibly terminate as well by returning an element of type A.

However, we have not specified when a program terminates. There are two choices, and depending on these choices we end up with 4 different sets. All of them depend on the answer set A.

- $(\mathsf{IO}^{*}_{\mathsf{client}}\ A)$, $(\mathsf{IO}^{*}_{\mathsf{server}}\ A)$, the set of client and server-side programs which are guaranteed to terminate eventually and to return an element of type A. These types are defined as algebraic types.
- $(\mathsf{IO}^{\infty}_{\mathsf{client}}\ A)$, $(\mathsf{IO}^{\infty}_{\mathsf{server}}\ A)$, the set of client and server-side programs which might run for ever or might terminate and return an element of type A. These types are defined as coalgebraic types.

All these data types might terminate immediately without having any interaction.

The types are defined as follows:

$$\begin{array}{rcllcl}
\mathsf{IO}^{*}_{\mathsf{client}}\ A & = & \mathsf{data} & \mathsf{return}(a : A) & \mid & \mathsf{do}(c : C, f : R\ c \to \mathsf{IO}^{*}_{\mathsf{client}}\ A) \\
\mathsf{IO}^{*}_{\mathsf{server}}\ A & = & \mathsf{data} & \mathsf{return}(a : A) & \mid & \mathsf{do}(f : (c : C) \to R\ c \times \mathsf{IO}^{*}_{\mathsf{server}}\ A) \\
\mathsf{IO}^{\infty}_{\mathsf{client}}\ A & = & \mathsf{codata} & \mathsf{return}(a : A) & \mid & \mathsf{do}(c : C, f : R\ c \to \mathsf{IO}^{\infty}_{\mathsf{client}}\ A) \\
\mathsf{IO}^{\infty}_{\mathsf{server}}\ A & = & \mathsf{codata} & \mathsf{return}(a : A) & \mid & \mathsf{do}(f : (c : C) \to R\ c \times \mathsf{IO}^{\infty}_{\mathsf{server}}\ A)
\end{array}$$

If we want to make the dependency on the interface explicit, we write the interface I as an additional subscript.

We can define the operations of a monad for all four of these types. For instance in case of $\mathsf{IO}^{*}_{\mathsf{client}}$, return is given by the constructor return. For $p : \mathsf{IO}^{*}_{\mathsf{client}}(A)$, $q : A \to \mathsf{IO}^{*}_{\mathsf{client}}(B)$, the monadic bind $p * q : \mathsf{IO}^{*}_{\mathsf{client}}(B)$[4] can be defined as follows: $(\mathsf{return}\ a) * q := q\ a$, $\quad (\mathsf{do}\ c\ f) * q := \mathsf{do}\ c\ (\lambda r. f\ r * q)$.

We introduce some simple operations:

[4] In Haskell this is written as $p\ >>=\ q$.

- If I, I' are two interfaces, we introduce their disjoint union as

  $$I \oplus I' := \mathsf{record}\{\mathsf{C} = I.\mathsf{C} + I'.\mathsf{C}, \mathsf{R} = [I.\mathsf{R}, I'.\mathsf{R}]\} : \mathsf{Interface}$$

  where

  $$\begin{array}{c} [I.\mathsf{R} + I'.\mathsf{R}] : I.\mathsf{C} + I'.\mathsf{C} \to \mathsf{Set} \\ [I.\mathsf{R} + I'.\mathsf{R}]\ (\mathsf{inl}\ c) \quad := \quad I.\mathsf{R}\ c \qquad [I.\mathsf{R} + I'.\mathsf{R}]\ (\mathsf{inr}\ c) \quad := \quad I'.\mathsf{R}\ c \end{array}$$

- If I,I' are interfaces, $f : I'.\mathsf{C} \to I.\mathsf{C}$, $g : (c : I'.\mathsf{C}, I.\mathsf{R}\ (f\ c)) \to I'.\mathsf{R}\ c$, then we can define $\mathsf{rename} : \mathsf{IO}^{\infty}_{I,\mathsf{server}}\ A \to \mathsf{IO}^{\infty}_{I',\mathsf{server}}\ A$ by

  $$\begin{array}{lll} \mathsf{rename}\ p = \mathsf{case}\ p\ \mathsf{of}\{ & (\mathsf{return}\ a) & \to \mathsf{return}\ a; \\ & (\mathsf{do}\ b) & \to \mathsf{do}\ (\lambda c.\mathsf{case}\ (b\ (f\ c))\ \mathsf{of} \\ & & \qquad \{\langle r, p' \rangle \to \langle g\ c\ r, \mathsf{rename}\ p' \rangle\})\} \end{array}$$

  So $(\mathsf{rename}\ p)$ operates as p, but translates the commands it receives into commands for p and responses from p back into its own responses. This operation allows hiding and renaming of an interface.

State-dependent interfaces. The notation of interface can be extended to state-dependent interfaces. A state-dependent interface is given by

- a set of externally observable states,
- a set of commands depending on the states,
- a set of responses depending on states and commands,
- and a next function, which determines the observable state one obtains after an interaction consisting of a command and a response to it has been carried out.

So we get

$$\begin{array}{rl} \mathsf{Interface}_{\mathsf{statedep}} = \Sigma(& \mathsf{S} : \mathsf{Set}, \\ & \mathsf{C} : \mathsf{S} \to \mathsf{Set}, \\ & \mathsf{R} : (s : \mathsf{S}, \mathsf{C}\ s) \to \mathsf{Set}, \\ & \mathsf{n} : (s : \mathsf{S}, c : \mathsf{C}\ c, \mathsf{R}\ s\ c) \to \mathsf{S})\ : \mathsf{Type} \end{array}$$

Let the following be fixed:

$$I := \mathsf{record}(\mathsf{S} = S, \mathsf{C} = C, \mathsf{R} = R, \mathsf{n} = n) : \mathsf{Interface}_{\mathsf{statedep}}$$

Assuming $A : S \to \mathsf{Set}$ and $s : \mathsf{S}$ we introduce the set of client/server-side interactive programs starting in state s and possibly terminating in a state s' with result $(A\ s')$ as follows:

$$
\begin{array}{l}
\mathsf{IO}^{*}_{\mathsf{statedep,client}}\, A\, s \\
\quad = \mathsf{data} \quad \mathsf{return}(a : A\, s) \\
\qquad\qquad \mid \quad \mathsf{do}(c : C\, s, f : R\, s\, c \rightarrow \mathsf{IO}^{*}_{\mathsf{statedep,client}}\, A\, (n\, s\, c\, r)) \\
\mathsf{IO}^{*}_{\mathsf{statedep,server}}\, A\, s \\
\quad = \mathsf{data} \quad \mathsf{return}(a : A\, s) \\
\qquad\qquad \mid \quad \mathsf{do}(f : (c : C\, s) \rightarrow R\, s\, c \times \mathsf{IO}^{*}_{\mathsf{statedep,server}}\, A\, (n\, s\, c\, r)) \\
\mathsf{IO}^{\infty}_{\mathsf{statedep,client}}\, A\, s \\
\quad = \mathsf{codata} \quad \mathsf{return}(a : A\, s) \\
\qquad\qquad\quad \mid \quad \mathsf{do}(c : C\, s, f : R\, s\, c \rightarrow \mathsf{IO}^{\infty}_{\mathsf{statedep,client}}\, A\, (n\, s\, c\, r)) \\
\mathsf{IO}^{\infty}_{\mathsf{statedep,server}}\, A\, s \\
\quad = \mathsf{codata} \quad \mathsf{return}(a : A\, s) \\
\qquad\qquad\quad \mid \quad \mathsf{do}(f : (c : C\, s) \rightarrow R\, s\, c \times \mathsf{IO}^{\infty}_{\mathsf{statedep,server}}\, A\, (n\, s\, c\, r))
\end{array}
$$

Execution of interactive programs is an external operation. For this we assume an interface corresponding to the real world (stateless or state-dependent).

- In case of a client side program, commands correspond to interactive commands the program can demand from the real world like writing a string to console, demanding some user input from the keyboard, or manipulating a GUI. Responses correspond to responses the real world makes to such a command. For instance, in case of the writing a string this would be a simple success element $x : \{*\}$, in case of reading a string, it would be the string typed in. Running a program means that case distinction is applied to the program. If one obtains $(\mathsf{return}\ a)$, the program stops and returns a. If one obtains $(\mathsf{do}\ c\ f)$, then command c is carried out in the real world. Once one has obtained a real world response r, the program continues by executing $(f\ r)$.
- In case of server side programs, commands are requests the real world can make to the program. Responses are answers the program can give in response to such a request. The execution of such a program is carried out as follows: First case distinction is applied to the program. If one obtains $(\mathsf{return}\ a)$, the program stops and returns a. Otherwise, it is of the form $(\mathsf{do}\ f)$. Then the program waits for a request by the real world. If it receives a request c, $(f\ c)$ is evaluated to a pair $\langle r, p\rangle$ consisting of a response r and a next program p. This response r is sent back as answer to the real world and the program continues by carrying out the execution loop with program p.

1.4 SIMPLE OBJECTS

The basic idea of our approach to object-oriented programming in dependent type theory is that an object is considered as an interactive program, and that classes are functions which generate objects. Let us restrict ourselves first to a simple class, which has methods which take input from one set and return in response to this input an answer which is an element of another set. These methods might change the internal state of an object, but do not receive or return elements from other classes or interact with other objects. Later we will indicate how to deal with the situation in which a method might call methods of other objects, including the object itself. In the current simple situation we have methods $(i = 1, \ldots, m)$

$$\mathsf{method}_i : (x^i_1 : A^i_1, \ldots, x^i_{n_i} : A^i_{n_i}) \to R_i[x^i_1, \ldots, x^i_{n_i}]$$

Note that these are not functions, but each method call depends on the internal state of an object, and changes the state of the object. In dependent type theory we can allow the set $R_i[x^i_1, \ldots, x^i_{n_i}]$ to depend on $x^i_1, \ldots, x^i_{n_i}$. Public instance variables $x : A$ can be modelled by having methods $\mathsf{setx} : A \to \{*\}$ for setting the variable to the value, and $\mathsf{getx} : \{*\} \to A$ for obtaining the value of this variable.

The interface definition of methods as introduced above corresponds in dependent type theory to the stateless interface

$$I = \mathsf{record}\{\mathsf{C} = C, \mathsf{R} = R\}$$

where

$$\begin{array}{l} C = \mathsf{data}\ \mathsf{method}_1(x^1_1 : A^1_1, \ldots, x^1_{n_1} : A^1_n) \ \mid\ \cdots\ \mid\ \mathsf{method}_m(x^m_1 : A^m_1, \ldots, x^m_{n_m} : A^m_{n_m}) \\ R\ (\mathsf{method}_i\ x^i_1\ \cdots\ x^i_{n_i}) \quad = \quad R_i[x^i_1, \ldots, x^i_{n_i}] \end{array}$$

An object of this interface is an element of

$$\mathsf{Object}\ I := \mathsf{IO}^{\infty}_{\mathsf{server}, I}\ \emptyset$$

It is a server-side program, which receives requests (message-calls) $c : C$, and depending on them, computes a response $r : R\ c$ and changes its internal state. Note that by using dependent type theory we have encoded several methods method_1, $\ldots$, method_m into one of type $(c : I.\mathsf{C}) \to I.\mathsf{R}\ c$.

C as given above models the notion of an interface as it occurs for instance in Java.[5] In addition to interfaces, classes have constructors, each of which constructs an object of this class. This means that a constructor Constr with arguments $(x_1 : A_1, \ldots, x_n : A_n)$ is a function

$$\mathsf{Constr} : (x : A_1, \ldots, x_n : A_n) \to \mathsf{Object}\ I$$

As an example we represent a very simple example in dependent type theory, namely that of a memory cell holding one element $x : A$. Such a class has methods

$$\mathsf{setx} \quad : \quad A \to \{*\} \qquad \mathsf{getx} \quad : \quad \{*\} \to A$$

This means that the interface $I_A = \mathsf{record}\{\mathsf{C} = C_A, \mathsf{R} = R_A\}$ is of the form

$$\begin{array}{lcl} C_A & = & \mathsf{data}\ \mathsf{setx}\ (a : A)\ \mid\ \mathsf{getx} \\ R_A\ (\mathsf{setx}\ a) & = & \{*\} \qquad R_A\ \mathsf{getx} \quad = \quad A \end{array}$$

[5] More precisely one should say: interfaces without static constants; because of such constants Java interfaces are a hybrid between pure interfaces and abstract classes.

The standard implementation of a memory cell has constructor $f : A \rightarrow \mathsf{Object}\ I_A$ which is defined by guarded recursion as

$$\begin{array}{llll} f\ a = \mathsf{do}\ (\lambda c.\mathsf{case}\ (c)\ \mathsf{of}\ \{ & (\mathsf{setx}\ a') & \longrightarrow & \langle *, f\ a' \rangle; \\ & (\mathsf{getx}) & \longrightarrow & \langle a, f\ a \rangle \}) \end{array}$$

Manipulation of objects. The operation "rename" introduced in Sect. 1.3 allows to take an interface and hide and rename its methods. One can introduce as well operations which extend an object in order to deal with an extended interface.

1.5 COMBINING OBJECTS WITHOUT SELF-REFERENTIAL CALLS

We will now consider how to deal with objects, which might call methods of other objects. Let us take as an example 3 objects o_1, o_2, o_3, which can be called from the outside world using interfaces I_1, I_2, I_3. The method calls are $I_i.\mathsf{C}$, and the response to a method call $c : I_i.\mathsf{C}$ is $(I_i.\mathsf{R}\ c)$. We call I_i the *receiving interface* of object o_i. In a first step we will exclude self-referential calls. Then o_1 might call o_2, o_3. The outside world as presented to this object is therefore given by the interface $O := I_2 \oplus I_3$ (one might extend O by an additional external interface corresponding to communication with the real world, e.g. communications with GUIs or the console). We call O the *outside interface* of the object.

The object o_1 would in this situation receive a command $c : I_1.\mathsf{C}$. It can then carry out a possibly unbounded sequence of communications with its outside world, which possibly terminates, and if it terminates, returns an element $r : I_1.\mathsf{R}\ c$ and a program o_1' (i.e. a call-back) for interfaces I_1 and O. Then it continues with executing o_1'. So o_1 is an element of

$$\mathsf{Object}_{\mathsf{simple}}\ I_1\ O = \mathsf{codata}\ \mathsf{do}(f : (c : I_1.\mathsf{C}) \rightarrow \mathsf{IO}^{\infty}_{\mathsf{client},O}\ (I_1.\mathsf{R}\ c \times \mathsf{Object}_{\mathsf{simple}}\ I_1\ O))$$

The interpretation loop for this object would be as follows: Assume the object is $o = \mathsf{do}\ f$. Then o waits for a command $c : I_1.\mathsf{C}$ from interface I_1. When it has received c, it computes $p := f\ c$. It executes p as a client side program, which issues commands to O and receives responses from it until it terminates with an element $\langle r, o' \rangle$. Then the object returns the answer r on interface I_1, and the interpretation loop is iterated starting with object o' (the call-back).

In order to exclude indirect self-referential calls (that for instance o_1 calls o_2 which in turn calls o_1), o_2 would only be allowed to communicate with o_3 (i.e. have outside interface I_3) and o_3 would not be allowed to communicate with any other object at all. We can combine o_1, o_2, o_3 into one interactive program with interface $I := I_1 \oplus I_2 \oplus I_3$, and define from the resulting program an interactive program which has the following behaviour: it receives a call c from $I.\mathsf{C}$, passes it on to one of o_1, o_2, o_3, and simulates the communication between o_1, o_2, o_3. If this communication terminates it will return an answer $r : I.\mathsf{R}\ c$, and wait for the next method call. We will not go into details and instead move to a more complex situation, in which self-referential method calls are allowed. There we will introduce the resulting combined program in more detail.

1.6 COMBINING OBJECTS WITH SELF-REFERENTIAL CALLS

Complications arise if we want to extend the above in order to include self-referential calls.[6] For instance, we might replace $O = I_2 \oplus I_3$ as the outside world for o_1 by $O' := I_1 \oplus I_2 \oplus I_3$ and interpret a method call to the I_1 component of O' as a method call to o_1 itself. Or one might allow indirect self-referential calls, i.e. that o_1 calls o_2 and o_2 in turn calls o_1. The consequence of allowing such self-referential calls is that when an object has issued a command to the outside world, a method call to it might be made, before it has received the answer.

An element of $\mathsf{Object}_{\mathsf{simple}}$ has two phases: a phase in which it receives a command from the outside, and a phase where it issues commands to the outside. We need to extend this in such a way that we have objects, which always can receive a command from the outside, even if a method call made by them has not been answered yet. Note that these self-referential calls might affect the state of the object. Consider for instance a class with methods `method1` and `method2`, where `method1` is defined using Java-like code as follows (`y`, `u` are instance variables of the class):

```
A method1(B x){u = 0;
               y = method2(x);
               return y + u;};
```

The self-referential call to `method2` will interrupt the execution of `method1`, but will refer to the state of the class which has been changed by the line `u=0`. `method2` might change `u`, so the result returned by `method1` need not be `y+0`.

The consequence of the above considerations is that an object o with receiving interface I, external interface O, and the possibility of self-referential calls refers to a list[7] icl of elements of $I.\mathsf{C}$, namely the method calls to o, which it has not answered yet, and a list ocl of elements of $O.\mathsf{C}$, the set of external commands, for which o has made a request without having received an answer yet.

Let us fix some notations for dealing with lists: $(l)_i$ is the ith element of the list l, $(\mathsf{delete}_i\ l)$ is the result of deleting the ith element from the list l; $(\mathsf{insert}_i\ l\ x)$ is the result of inserting into list l at position i element x. Using these notations, we obtain the following definition, where $icl : \mathsf{List}\ I.\mathsf{C}$ and $ocl : \mathsf{List}\ O.\mathsf{C}$:

$$
\begin{array}{l}
\mathsf{Object}_{\mathsf{server}}\ I\ O\ icl\ ocl = \\
\qquad \Sigma(\mathsf{receive\_request}\colon (ic : I.\mathsf{C}, i \leq \mathsf{length}\ icl) \\
\qquad\qquad\qquad\qquad\qquad \to \mathsf{Object}_{\mathsf{client}}\ I\ O\ (\mathsf{insert}_i\ icl\ ic)\ ocl, \\
\qquad\quad \mathsf{receive\_answer} : (i < \mathsf{length}\ ocl, r : O.\mathsf{R}\ (ocl)_i) \\
\qquad\qquad\qquad\qquad\qquad \to \mathsf{Object}_{\mathsf{client}}\ I\ O\ icl\ (\mathsf{delete}_i\ ocl)) \\
\mathsf{Object}_{\mathsf{client}}\ I\ O\ icl\ ocl = \\
\qquad \mathsf{codata}\ \mathsf{send\_answer}(i < \mathsf{length}(icl), r : I.\mathsf{R}\ (icl)_i, \\
\qquad\qquad\qquad\qquad\qquad o : \mathsf{Object}_{\mathsf{server}}\ I\ O\ (\mathsf{delete}_i\ icl)\ ocl) \\
\qquad\qquad\quad \mid\ \mathsf{send\_request}(oc : O.\mathsf{C}, i \leq \mathsf{length}\ ocl, \\
\qquad\qquad\qquad\qquad\qquad o : \mathsf{Object}_{\mathsf{server}}\ I\ O\ icl\ (\mathsf{insert}_i\ ocl\ oc))
\end{array}
$$

[6] Note that this paper does not treat the dynamic creation of objects.

[7] As it is the default in functional programming, lists are 0-based.

An element of $(\mathsf{Object}_{\mathsf{server}}\ I\ O\ icl\ ocl)$ can receive method calls (elements of $I.\mathsf{C}$) and receive an answer for any of its pending requests to the outside world (elements of *ocl*).[8] It then switches to client mode. In that mode it either sends an answer to one of the requests made to it (elements of *icl*) or it sends requests $oc : O.\mathsf{C}$ to the outside using its external interface O. It then switches back to server mode.

Note that we have made our definition in such a way that an object can receive answers for any of its open requests in *ocl* and can answer any of its requests in *icl*, not only the last one. Otherwise for instance the definition of $o_0 \oplus o_1$ below would become rather complicated. Furthermore, we are allowed to insert new elements to *icl* and *ocl* at arbitrary positions, not only in a stack-like way at front. This makes programming much easier, since it allows to keep *icl* and *ocl* synchronised.

Constructing an interactive program from several objects with internal communications. We will show how to construct an interactive program from the combination of several objects, which possibly call each other. In a first step we combine two objects o_1 and o_2 with receiving interfaces I_1, I_2 and the same outside interface O into one object $o_1 \oplus o_2$ with interfaces $I_1 \oplus I_2$ and O. If we consider our main example, namely having objects o_1, o_2, o_3 with receiving interface I_1, I_2, I_3, respectively and the same outside interface $O = I_1 \oplus I_2 \oplus I_3$, we see that $o_1 \oplus o_2 \oplus o_3$ is an object, for which the receiving and the outside interface are the same, namely O. In a second step we will then show how to simulate a program for which the receiving and outside interface coincides – let it be I – by an interactive program which has no outside interface and I as receiving interface.

Step 1: Definition of $o_0 \oplus o_1$. Let o_i have receiving interface I_i and the same outside interface O. Let $I := I_0 \oplus I_1$, and $O' := O \oplus O$. For $icl : \mathsf{List}\ I.\mathsf{C}$ we define $(\mathsf{proj}_0\ icl)$ to be the list of elements $ic : I_0.\mathsf{C}$ s.t. $(\mathsf{inl}\ ic)$ occurs in *icl* (taken in the same order as they occur in *icl*). Similarly, we define $\mathsf{proj}_1\ icl : \mathsf{List}\ I_1.\mathsf{C}$ and $\mathsf{proj}_i\ ocl : \mathsf{List}\ O.\mathsf{C}$, where $ocl : \mathsf{List}\ O'.\mathsf{C}$. Furthermore, if $ocl : \mathsf{List}\ O'.\mathsf{C}$ then let $\mathsf{unify}\ ocl : \mathsf{List}\ O.\mathsf{C}$ be the result of replacing $(\mathsf{inl}\ oc)$ and $(\mathsf{inr}\ oc)$ occurring in *ocl* by *oc*. One can define from elements $icl : \mathsf{List}\ I.\mathsf{C}$, $ocl : \mathsf{List}\ O'.\mathsf{C}$, $o_i : \mathsf{Object}_{\mathsf{server}}\ I_i\ O\ (\mathsf{proj}_i\ icl)\ (\mathsf{proj}_i\ ocl)$ an element

$$o := o_0 \oplus o_1 : \mathsf{Object}_{\mathsf{server}}\ (I_0 \oplus I_1)\ O\ icl\ (\mathsf{unify}\ ocl)\ .$$

[8] Two referees commented that in most object-oriented languages, an object is only able to receive an answer to its most recent open request. Similarly it only sends an answer to the most recent open request it has received. One could modify the behaviour of the objects defined in this section so as to have this behaviour. However, we think that our current version is more flexible, since it allows to deal with concurrency. In a concurrent situation, an object may send several request and receive answers to it in a different order. If we want that an object in our setting only is able to deal with the most recent open request, we can implement such an object as follows: If it receives an answer which is not due yet, it stores this answer internally. It then delays dealing with this answer until it has become an answer to the most recent open request.

We compute the component $o' := o.\mathsf{receive\_request}\ (\mathsf{inl}\ ic)\ i$ and leave the other cases to the reader. Let $icl' := \mathsf{insert}_i\ icl\ (\mathsf{inl}\ ic)$, $o'_0 := (o_0.\mathsf{receive\_request}\ ic\ i')$ for the index i' corresponding to i in $(\mathsf{proj}_0\ icl)$. If $o'_0 = (\mathsf{send\_answer}\ j\ r\ o''_0)$, then $o' = (\mathsf{send\_answer}\ j'\ r\ (o''_0 \oplus o_1))$, and if $o'_0 = (\mathsf{send\_request}\ oc\ j\ o''_0)$, then $o' = (\mathsf{send\_request}\ oc\ j'\ (o''_0 \oplus o_1))$. Here j' is the index corresponding to j in icl' and ocl, respectively.

Step 2: Simulating the internal communications. The second step is to consider one object for which both receiving interface and outside interface coincide, let it be I. We want to obtain from such an object an interactive program which has only a receiving interface I. This program receives calls via I and passes them to its corresponding object o. If o makes calls to its outside interface I, these are then passed back to o itself as a request from its receiving interface I. Any answers o returns via its receiving interface in response to such requests are passed back to o itself as an answer from its outside interface in response to its original request.

A more general situation, which we do not consider here because of lack of space, would be to have an object with receiving interface $I \oplus X$ and outside interface $O \oplus X$, which can receive requests from the outside via I, send requests to the outside via O, and for which any calls via X to the outside are bounced back to the object in question via its receiving interface part X.

There is one complication, namely that the internal communications between the objects might not terminate. The way of dealing with it is to simulate this program by a state-dependent interactive program. That program receives a request via I. Then its command set changes to $\{\mathsf{continue}\}$. Whenever it receives $\mathsf{continue}$ it will carry out one more step of its internal communication, until an answer to the original request is obtained. Then the answer is given back, the program switches back to its original state where it can receive requests via $I.\mathsf{C}$.

Therefore we define a function

$$\mathsf{simulate} : \mathsf{Object}_{\mathsf{server}}\ I\ I\ \mathsf{nil}\ \mathsf{nil} \longrightarrow \mathsf{IO}^{\infty}_{\mathsf{statedep,server}}\ I'\ \mathsf{s}_0$$

for a suitable state-dependent interface I' and $\mathsf{s}_0 : I'.\mathsf{S}$.

$I'.\mathsf{S}$ has one state in which it can receive commands from $I.\mathsf{C}$, and a second state in which its set of commands is $\{\mathsf{continue}\}$. In the latter case we need to store the command $c : I.\mathsf{C}$ it has received but not answered. So the set of states is $1 + I.\mathsf{C}$. Let $\mathsf{s}_0 := \mathsf{inl}$. In state inl, the program can receive commands $c : I.\mathsf{C}$. It either replies with response $r : I.\mathsf{R}\ c$, or answers with delay and switches into state $(\mathsf{inr}\ c)$. In state $(\mathsf{inr}\ c)$, it can only receive request $\mathsf{continue}$. It replies with an answer to the original request, or with delay and continues in state $(\mathsf{inl}\ c)$. We obtain the interface $I' := \mathsf{Interface}_{\mathsf{statedep}}(\mathsf{S} = S, \mathsf{C} = C, \mathsf{R} = R, \mathsf{n} = n)$, where

$$\begin{array}{lcllcl}
S & = & 1 + I.\mathsf{C} & & & \\
C\ \mathsf{inl} & = & I.\mathsf{C} & C\ (\mathsf{inr}\ c) & = & \{\mathsf{continue}\} \\
R\ \mathsf{inl}\ c & = & R\ (\mathsf{inr}\ c)\ \mathsf{continue} & & = & \mathsf{delay} + \mathsf{reply}(r : I.\mathsf{R}\ c) \\
n\ \mathsf{inl}\ c\ \mathsf{delay} & = & n\ (\mathsf{inr}\ c)\ \mathsf{continue}\ \mathsf{delay} & & = & \mathsf{inr}\ c \\
n\ \mathsf{inl}\ c\ (\mathsf{reply}\ r) & = & n\ (\mathsf{inl}\ c)\ \mathsf{continue}\ (\mathsf{reply}\ r) & & = & \mathsf{inl}
\end{array}$$

We consider one case of the definition of $p := \mathsf{simulate}\, o$. Assume p receives a request $c : C\, \mathsf{inl} = I.\mathsf{C}$. Then we make case distinction on $o' := o.\mathsf{receive\_request}\, c\, 0$. If we obtain $(\mathsf{send\_answer}\ 0\ r\ o'')$, then the response is $(\mathsf{reply}\ r)$ and we continue with $(\mathsf{simulate}\ o'')$. If we obtain $(\mathsf{send\_request}\ c'\ 0\ o'')$ then p returns delay. We bounce back c' as a request to o'' by computing $o''' := o''.\mathsf{receive\_request}\ c'\ 0$, from which we compute the next execution steps of p. The full details will be given in a followup paper.

Translation of standard object-oriented code into objects of dependent type theory. We show how to translate object-oriented code, e.g. written in Java, into elements of $(\mathsf{Object}_{\mathsf{server}}\ \mathsf{I}\ \mathsf{O}\ icl\ ocl)$. We are able to deal with objects which communicate with a fixed number of other objects and do not create new objects dynamically on the heap. So, when constructed, the object receives references to a fixed number of other objects and is then allowed to communicate with them without modifying them. The code can be represented by the following data:

- We have a global state $\mathsf{G} : \mathsf{Set}$ of the system which determines the state of those global instance variables, which are not objects. Instance variables which are objects will be treated as defining an outside interface O.
- We have methods with their arguments and result types, which can be given as a stateless interface I.
- We have an outside interface O, which is obtained as the union of receiving interfaces of all the objects, to which the object can send method calls.
- The body of method $ic : \mathsf{I.C}$ is an interactive program which operates as follows: Depending on the global state $g : \mathsf{G}$, it computes a new global state, and computes either an answer $r : \mathsf{I.R}\ ic$, or it makes a call to its outside interface, i.e. sends $oc : \mathsf{O.C}$. Depending on the response $r : \mathsf{O.R}\ oc$ it returns a new program of the same form. The updating of G is best dealt with by making use of the state monad $\mathsf{M_G}\ X = \mathsf{G} \to \mathsf{G} \times X$. Then the method body is an element of

  $$\begin{array}{rcl} \mathsf{MethodBody}\ ic & = & \mathsf{codata\ do}\ (f : \mathsf{M_G}\ (\mathsf{Action}\ ic))\ , \quad \text{where} \\ \mathsf{Action}\ ic & = & \mathsf{return}(r : \mathsf{I.R}\ ic) \\ & & +\mathsf{call}(oc : \mathsf{O.C}, f' : \mathsf{O.R}\ oc \to \mathsf{MethodBody}\ ic) \end{array}$$

  The methods are then given as an element

  $$\mathsf{methodBody} : (ic : \mathsf{I.C}) \to \mathsf{MethodBody}\ ic\ \ .$$

The complete code is given as a tuple

$$\langle \mathsf{G}, \mathsf{I}, \mathsf{O}, \mathsf{methodBody} \rangle$$

which we call a *class code*.

Example: We consider as an example a class which computes the Fibonacci numbers efficiently by memorising values it already has computed. We use a Java-like syntax with some functional additions:

```
class Fib{Map mem;
          nat fib_aux(nat n){
            if (n <= 1) {return 1;}
            else {nat k = fib (n-1);
                  nat l = fib (n-2); return k+l;}};

           nat fib (nat n){case (lookup(mem,n)){
                           (just(k)) -> {return k;};
                           (nothing) -> {nat k = fib_aux(n);
                                         put(mem,n,k);
                                         return k;}}}};
```

Map is the data type of finite maps from nat to nat, where nat stands for the type of natural numbers. $\mathsf{lookup}(mem,n) : \mathsf{Maybe}(\mathsf{nat})$ returns $\mathsf{just}(k)$ if n is in the domain of mem and $mem(n) = k$, and $\mathsf{nothing}$ otherwise. $\mathsf{put}(mem,n,k)$ updates mem so that it returns on argument n value k. We have not defined Map as an object with method calls but as a value parameter, in order to obtain a non-trivial global state.

Then the class code for `Fib` is $\langle \mathsf{G}, \mathsf{I}, \mathsf{O}, \mathsf{MethodBody} \rangle$. Here $\mathsf{G}, \mathsf{I}, \mathsf{O}$ are defined as follows:

$$\begin{array}{lcl}
\mathsf{G} & := & \mathsf{Map} \\
\mathsf{I.C} & := & \mathsf{fib\_aux}(n : \mathsf{nat}) + \mathsf{fib}(n : \mathsf{nat}), \\
\mathsf{I.R}\ c & := & \mathsf{nat} \\
\mathsf{O} & := & \mathsf{I}
\end{array}$$

$\mathsf{O} = \mathsf{I}$, since all method calls to the outside are to the object itself. If we had defined mem to be an object with receiving interface O', then we would have $\mathsf{G} = 1$ and $\mathsf{O} = \mathsf{I} \oplus \mathsf{O}'$ instead.

Before defining $\mathsf{methodBody}$ we introduce some convenient syntax for dealing with $(\mathsf{M_G}\ X)$: Let $\mathsf{return_G} := \lambda x.\lambda g.\langle g, x \rangle : X \to \mathsf{M_G}\ X$, and
$\mathsf{do_G} : \mathsf{Action}\ ic \to \mathsf{MethodBody}\ ic$, $\mathsf{do_G}\ x := \mathsf{do}\ (\mathsf{return_G}\ x)$. Then we define

$$\begin{array}{l}
\mathsf{methodBody}\ (\mathsf{fib\_aux}\ n) \\
\ = \mathsf{if}\ (n \leq 1)\ \mathsf{then}\ (\mathsf{do_G}(\mathsf{return}\ 1)) \\
\qquad\qquad \mathsf{else}\ \ (\mathsf{do_G}\ (\mathsf{call}\ (\mathsf{fib}\ (n-1)) \\
\qquad\qquad\qquad\qquad (\lambda k.\mathsf{do_G}\ (\mathsf{call}\ (\mathsf{fib}\ (n-2)) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\lambda l.\mathsf{do_G}\ (\mathsf{return}\ (k+l)))))))
\end{array}$$

We leave the definition of $\mathsf{methodBody}\ (\mathsf{fib}\ n)$ to the reader.

Translation of a class code into an object of dependent type theory. The intermediate state of an object determined by a class code

$$\langle \mathsf{G}, \mathsf{I}, \mathsf{O}, \mathsf{methodBody} \rangle$$

is given by $g : \mathsf{G}$ and an element $lmc : \mathsf{List\ OpenMethodCall}$, where

$$\mathsf{OpenMethodCall} := (ic : \mathsf{I.C}) \times (oc : \mathsf{O.C}) \times (f : \mathsf{O.R}\ oc \to \mathsf{MethodBody}\ ic)$$

An element $\langle ic, oc, f \rangle : \mathsf{OpenMethodCall}$ consists of the original method call ic, the last outside request oc done by the method, and a function f, which determines, depending on a response r to oc the next step in the evaluation of the method. We define a function

$$\mathsf{translate} : (g : \mathsf{G}, lmc : \mathsf{List\ OpenMethodCall}) \to \mathsf{Object}_{\mathsf{server}}\ \mathsf{I}\ \mathsf{O}\ (\mathsf{icl}\ lmc)\ (\mathsf{ocl}\ lmc)$$

which depends on a suppressed class code and computes from an intermediate state of that program given by g and lmc an object of dependent type theory. Here, $(\mathsf{icl}\ lmc)$ and $(\mathsf{ocl}\ lmc)$ are the results of projecting the elements of lmc to $\mathsf{I.C}$ and $\mathsf{O.C}$, respectively. Then the constructor for the object, which depends on the initial internal state $g : \mathsf{G}$ is the function

$$\lambda g.\mathsf{translate}\ g\ \langle\rangle : \mathsf{G} \to \mathsf{Object}_{\mathsf{server}}\ \mathsf{I}\ \mathsf{O}\ \langle\rangle\ \langle\rangle \quad .$$

The definition of $o := \mathsf{translate}\ g\ lmc$ is by guarded recursion, and we compute only $o' := (o.\mathsf{receive\_request}\ ic\ i)$, i.e. the case when o receives a method call ic: Let $\mathsf{methodBody}\ ic = \mathsf{do}\ f,\ f\ g = \langle g', m \rangle$. We make case distinction on m. If we obtain $(\mathsf{call}\ oc\ f')$, $o' := \mathsf{send\_request}\ oc\ i\ (\mathsf{translate}\ g'\ lmc')$ where $lmc' = \mathsf{insert}_i\ lmc\ \langle ic, oc, f' \rangle$. If we obtain $(\mathsf{return}\ r)$, $o' := \mathsf{send\_answer}\ i\ r\ (\mathsf{translate}\ g'\ lmc)$.

1.7 CONCLUSION

We have reviewed the basics of interactive programs in dependent type theory. Then we have introduced a notion of an object which is isolated (no interaction with other objects). We have seen how to combine objects with and without self-referential calls. Finally we have shown how to translate standard object-oriented class code into dependently typed objects.

The above deals only with some aspects of object-oriented programming. We have touched hiding and renaming, but we have not dealt yet in full with inheritance. Already subtyping is known to be quite complicated in the context of dependent type theory, and inheritance is even more sophisticated. However, it seems not to be too complicated to translate an object from one interface to a restricted one, which gives some notion of subtyping.

The most difficult problem seems to be to deal with a heap and pointers, in order to be able to construct for instance linked lists. We have some ideas based on the IORef monad in Haskell, but we do not have space to discuss these in this article.

What is of course missing is to translate typical object-oriented programs into this language and see how they execute. For this it is necessary to introduce an improved syntax for representing object-oriented programs in dependent type theory. The class code introduced in Sect. 1.6 seems to be pretty close to a satisfactory solution.

REFERENCES

[AC96] Martín Abadi and Luca Cardelli, editors. *A Theory of Objects*. Springer, 1996.

[Alt01] Thorsten Altenkirch. Representations of first order function types as terminal coalgebras. In *Typed Lambda Calculi and Applications, TLCA 2001*, number 2044 in LNCS, pages 8 – 21, 2001.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[Bru02] Kim B. Bruce. *Foundations of object-oriented languages: types and semantics*. MIT Press, Cambridge, MA, USA, 2002.

[Coq94] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 62–78, 1994.

[Geu92] Herman Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Informal proceedings of the 1992 workshop on Types for Proofs and Programs, Bastad 1992, Sweden*, pages 183 – 207, 1992.

[Gim94] E. Gimenéz. Codifying guarded definitions with recursive schemes. In *Proceedings of the 1994 Workshop on Types for Proofs and Programs*, pages 39–59. LNCS No. 996, 1994.

[Gor94] A.D. Gordon. *Functional programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[Han00] Peter Hancock. *Ordinals and interactive programs*. PhD thesis, LFCS, University of Edinburgh, 2000.

[HH06] Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. *Ann. Pure Appl. Logic*, 137(1-3):189–239, 2006.

[HS99] Peter Hancock and Anton Setzer. The IO monad in dependent type theory. In *Electronic proceedings of the workshop on dependent types in programming, Göteborg, 27-28 March 1999*, 1999. Available via http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/dtp99.html.

[HS00a] Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Proceedings of CSL 2000*, volume 1862 of *LNCS*, pages 317–331, 2000.

[HS00b] Peter Hancock and Anton Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming, Portugal, 7 July 2000*, 2000. Electronic proceedings, http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.

[HS04] Peter Hancock and Anton Setzer. Interactive programs and weakly final coalgebras (extended version). In T. Altenkirch, M. Hofmann, and J. Hughes, editors, *Dependently typed programming*, number 04381 in Dagstuhl Seminar Proceedings, 2004. Available via http://drops.dagstuhl.de/opus/.

[Jac95] Bart Jacobs. Objects and classes, co-algebraically. In Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Jörg Schek, editors, *Object Orientation with Parallelism and Persistence*, pages 83–103. Kluwer, 1995.

[Jac98] Bart Jacobs. Coalgebraic reasoning about classes in object-oriented languages. *Electronical Notes in Computer Science*, 11:231 – 242, 1998. Special issue on the workshop Coalgebraic Methods in Computer Science (CMCS 1998).

[KL05] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. Submitted, 2005.

[Mes93] José Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In *Research directions in concurrent object-oriented programming*, pages 314–390, Cambridge, MA, USA, 1993. MIT Press.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Logic in Computer Science Conference*, 1989.

[MS05] Markus Michelbrink and Anton Setzer. State dependent IO-monads in type theory. *Electronic Notes in Theoretical Computer Science, Elsevier*, 122:127 – 146, 2005.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Clarendon Press, 1990.

[Pie92] Benjamin C. Pierce. Bounded quantification is undecidable. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 305–315, New York, NY, USA, 1992. ACM Press.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[PW93] S. L. Peyton Jones and Philip Wadler. Imperative functional programming. In *20'th ACM Symposium on Principles of Programming Languages*, Charlotte, North Carolina, January 1993.

[Rei95] H. Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.

[Set03] Anton Setzer. Java as a functional programming language. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, pages 279 – 298. LNCS 2646, 2003.

[Wad97] Philip Wadler. How to declare an imperative. *ACM Comput. Surv.*, 29(3):240–263, 1997.