In: *Proc. 4th Int'l Workshop on Database Programming Languages, New York City, Aug '93.*

A Functional Object Database Language

Christian Laasch, Marc H. Scholl
Faculty of Computer Science, Databases and Information Systems
University of Ulm, D 89069 Ulm, Germany
e-mail: {laasch, scholl}@informatik.uni-ulm.de

Abstract

The language BCOOL is formally defined using a denotational semantics approach. BCOOL is a functional object database language with a very flexible, yet strong and statically checked, type system. Its main source of flexibility is its support for object evolution, that is, dynamic type changes of existing objects. Originally, BCOOL was used as a formal basis for a more traditional (relational algebra-style) database language, COOL. In this paper, though, BCOOL is presented on its own. The purpose being to compare with other functional languages and discuss the virtues and limitations that BCOOL and these functional languages have w.r.t. each other in terms of (i) the above-mentioned flexibility in the type system, which we consider essential for *objects* and (ii) the orthogonality of the language.

1 Introduction

COOL is an object database query language developed in the COCOON project [20, 21]. In a nutshell, COOL is an object-flavored extension of a (nested) relational algebra. The object flavor is established by the inclusion of concepts such as abstract object types, functions (methods), type hierarchies, and classes. The effects of the algebraic query operators are pretty similar to their relational counterparts, except for the fact that they have to take into account that they now work with objects with an identity (which led to the notion of object-preserving queries) and with a much richer type/class system that requires typing (and classification) of query results. Informal presentations of COCOON and the COOL language have been given earlier [19, 21]. A formalization of COCOON and COOL using BCOOL was developed in [20].

In this paper, we discuss the functional object language BCOOL in detail. We present the formal semantics of query and update operations using a denotational approach. In contrast to [17] and [10], which also propose operations for object evolution, we also analyze the impact of dynamic type changes on the type system. Further, we compare our approach with functional database languages. While these typically offer more flexibility w.r.t. orthogonality and genericity of the type system, BCOOL provides objects and object evolution (i.e., dynamic type changes). Adding any of these extra capabilities to the other language, brings them close together. We show how this can be achieved and what the consequences are.

For this purpose, BCOOL serves as a formal basis for the integration of functional (database) languages supporting polymorphism, static type inferencing, and orthogonality (e.g., as in Machiavelli [16] or FAD [5]) with object-oriented data models including objects with subtyping and flexible update facilities (e.g., as in Iris [27], Melampus [17], or COCOON [12]). The key objectives are:

- an extension of the relational algebra to an object query language. We started from relational algebra in order to preserve the potential for and knowledge on query optimization.
- static type-checking that allows for early error detection and reduction of run-time effort.
- update operations that allow object evolution such that objects are not only added to and removed from collections ("classes"), but can also change their types dynamically.
- extensibility of the set of type constructors such that the model can be tailored to requirements of new applications.

In this paper, the main focus is on the type system; we define subtyping and the impact of object evolution on static-type checking. After introducing the syntax in Section 2, we define the semantics of types, the type inference rules for expressions, and the impact of object evolution operations in Section 3. We formalize the model by using a denotational semantics approach [24], in which the semantics of language elements (i.e., types, expressions, and updates) are defined by "denotation functions" that return elements of the "semantic domain". In Section 4 we further analyze the commonalities and differences between object database and pure functional languages by discussing extensions of both that narrow the gap between them. On the one hand, we sketch how to extend BCOOL to become fully orthogonal and include genericity, on the other hand, we discuss the integration of objects with their respective operations into Machiavelli as one well-known functional (database) language.

2 BCOOL Model: Concepts and Syntax

Initially, BCOOL was developed as a formal model for the definition of COOL, which has a number of concepts, such as predicative specification of collections (called classes) that can be derived from more basic constructs provided in BCOOL. Hereinafter we will focus on the essentials of the formal language BCOOL and sometimes use the more syntactically sugared COOL syntax for illustration only.

BCOOL consists of only the few following concepts, which can also be found in (almost) all object-oriented models:

Types can be basic or constructed. *Basic types* describe (pairwise) disjoint sets of instances. Besides concrete (or *printable* [11]) types describing **data**, or values, (such as integers, boolean, strings), there is a basic type denoting **objects**, on which *object types* can be defined by subtyping (see below). Objects are fully encapsulated. They can be used and manipulated only by means of their interface, a set of functions. *Constructed types* can be specified using the build-in type constructors set ({ }) and function ($\rightarrow$).[1]

Types serve several purposes: (i) they represent a "repository" of possible values (this will be called the *domain* of the type below, an intensional notion of type); (ii) they are used by the compiler for type checking (i.e., assuring that only "(type) valid" expressions are ever executed. For example, we would not allow to compute the square root of a string. Finally, (iii) types are often also used as containers (collections) for those values of that type, which are currently "in use" in the database (an extensional notion). Notice that we do *not* use this aspect of types in general, but only for *object* types (where this will be called the "active domain" of the type).

Subtyping is used to describe (sub)-sets of objects with common interfaces, such that type-checking becomes more meaningful. The COOL definition of a subtype consists of three parts: a set of supertypes, a set of local functions, and (possibly) a type name. Any instance of the subtype is also an instance of its supertypes (*substitutability*), and all functions defined on the supertypes are applicable to the instances of the subtype (*inheritance of the interface*), in addition to the locally defined functions.

As a running example let us consider persons and employees defined by the COOL definitions:

$$\begin{array}{ll}
\textbf{type } \mathit{Pers} \textbf{ isa } \mathit{Object} = & \mathit{name} : \textbf{string}, \\
 & \mathit{age} : \textbf{integer}, \\
 & \mathit{children} : \textbf{set of } \mathit{Pers}; \\
\textbf{type } \mathit{Empl} \textbf{ isa } \mathit{Pers} = & \mathit{sal} : \textbf{integer}, \\
 & \mathit{manager} : \mathit{Mngr}; \\
\textbf{type } \mathit{Mngr} \textbf{ isa } \mathit{Empl} = & \mathit{budget} : \textbf{integer};
\end{array}$$

The functions *name, age,* and the set-valued function *children* are applicable to instances of *Pers*, which is a subtype of the predefined type *Object*, which has no user-defined functions. Because *Empl* is a subtype of *Pers*, the functions of *Pers* are inherited by *Empl*, such that *name, age, children, sal*, and *manager* can be applied to *Empl*'s instances. Similarly, the function *budget* can be applied to managers (the instances of *Mngr*) in addition to the *Empl*'s functions. [2]

On the (formal) BCOOL level, however, we omit the names of object types. Instead, object types are identified by the set of applicable functions. Syntactically, *Pers* is referred to as $[name, age, children]$ and *Empl* as [*name, age, children, sal*]. Thus, the syntax of BCOOL type expressions can be described by the following list:

$$\begin{array}{lll}
\tau = & (\ \tau\ ) & /*\ \text{types}\ */ \\
 & \mid \beta_{Int} & /*\ \text{INTEGER}\ */ \\
 & \mid \beta_{Bool} & /*\ \text{BOOLEAN}\ */ \\
 & \ \vdots & \\
 & \mid \beta_{Object} & /*\ \text{Objects}\ */ \\
 & \mid [f_1, ..., f_n] & /*\ \text{object types}\ */ \\
 & \mid \{\ \tau\ \} & /*\ \text{set types}\ */ \\
 & \mid \beta_{Object} \rightarrow \tau & /*\ \text{function types}\ */
\end{array}$$

[1]In Sec. 4 we discuss how to integrate additional type constructors.

[2]Range restrictions of functions within subtypes are regarded as different functions by unique function names that are prefixed by the COOL type name similar to C++.

Functions (denoted by the meta-variable f) are described by a unique name and a signature. We can distinguish retrieval (stored or computed) and update functions. Generic update operations can only be used for stored retrieval functions, which are uniform abstractions of "attributes" and "relationships" of classical data models. Direct updates of computed functions require type-specific methods, whereas indirect updates (i.e., updates to values used in the derivation) are automatically propagated.

Because of the desired flexibility (e.g., in case of projections), the domain of a function is not the COOL type on which the function is defined, but the (usually unnamed) object type on whose instances only the function itself is applicable. For example, the function *name* might typically be introduced in a type $Pers = [name, age, children]$. Formally, however, we use "$[name] \rightarrow$ **string**" as the signature of *name*. The reason for this is the semantics of object types, which allows to apply the *name* function not only to persons, but also, more generally, to objects that are contained in the active domain of the type $[name]$. Thus, it becomes possible that, e.g., the *children* function can be hidden by projecting only the *name* and *age* of persons. In other words, the substitutability of typed expressions is adapted to unnamed typed, such that not only specializations of types can be created, but also generalizations. Consequently, object types are arranged into a lattice (see Sec. 3.1).

Variables (denoted by x) are used as temporary names ("handles") for instances of any type (e.g., data values, objects, sets, or functions). They have to be declared with their type in the database language, for compile-time type checking.

Query Language. The query language on objects is an extension of a (nested) relational algebra [18] with *object-preserving* semantics. It contains a selection (**select**), a projection (**project**), and the set operations ($\cup$, $\cap$). Instead of join we use a macro mechanism known from functional languages (**let**) that allows to define additional (virtual) functions mapping objects onto their join partners. An informal presentation and rationale for this query language was given in [21]. Additionally, we provide the following constructors for building expressions and make use of the standard operations for integers (e.g., $+$ and $-$), boolean (e.g., $\wedge$, $\vee$, and $\neg$), and comparisons (e.g., $=$) that are not mentioned here any further:

$$\begin{array}{lll}
e \; = & (\; e \;) & /*\ \text{expressions}\ */ \\
 & \mid c & /*\ \text{constants}\ */ \\
 & \mid x & /*\ \text{variables}\ */ \\
 & \mid \textbf{new}\,() & /*\ \text{creating objects}\ */ \\
 & \mid \textbf{adom}\,([f_1, ..., f_n]) & /*\ \text{active domains}\ */ \\
 & \mid \lambda x : \tau.\ e & /*\ \text{function expression}\ */ \\
 & \mid f\ (e) & /*\ \text{function applications}\ */ \\
 & \mid \{\ e\ \} & /*\ \text{sets}\ */ \\
 & \mid \textbf{pick}\,(e) & /*\ \text{pick one element of a set}\ */ \\
 & \mid e_1 \in e_2 & /*\ \text{test for set–membership}\ */ \\
 & \mid e \cup e & /*\ \text{unions}\ */ \\
 & \mid e \cap e & /*\ \text{intersections}\ */ \\
 & \mid \textbf{select}\,[\lambda x.e]\ (e) & /*\ \text{selections}\ */ \\
 & \mid \textbf{project}\,[f_1, ..., f_n]\ (e) & /*\ \text{projections}\ */ \\
 & \mid \textbf{let}\ f = e\ \textbf{in}\ e\ \textbf{end} & /*\ \text{joins}\ */
\end{array}$$

Update Operations. In order to change the state of a database, assignments (:= and **set**) are provided to set the values of variables and functions. Additional operations allow for object evolution: objects can be created, deleted, and can change their types dynamically.

$$
\begin{array}{lll}
u = & (u) & /*\ \text{updates}\ */ \\
 & \mid x := e & /*\ \text{setting variables}\ */ \\
 & \mid \mathbf{set}\ [f := e](e) & /*\ \text{setting function values}\ */ \\
 & \mid \mathbf{new}\ () & /*\ \text{creating objects}\ */ \\
 & \mid \mathbf{gain}\ [f_1, ..., f_n](e) & /*\ \text{adding functions to objects}\ */ \\
 & \mid \mathbf{lose}\ [f_1, ..., f_n](e) & /*\ \text{removing functions from objects}\ */ \\
 & \mid \mathbf{destroy}\ (e) & /*\ \text{deleting objects}\ */
\end{array}
$$

The motivation for dynamic type changes is the longevity of objects. In contrast to programs, where data are valid only until the end of program, information stored in databases is usually valid over a couple of years, such that the "role" of objects might change. For example, if an object created with the type *Pers* is hired by a company, the type of this object must be changed to the type *Empl* in order to make functions *sal* and *manager* applicable, while still talking about the identical object. In case that this employee is fired, the functions that are applicable to employees, but not to persons, must not be applied anymore; i.e., the instance relationship between the object and the type *Empl* has to be removed.

However, there is a problem in combining dynamic type changes with (static) type-checking. Considering the above example, let us assume that the variable p of type *Pers* denotes a person that is to be hired. Of course, using p in **set** $[sal := \ldots](p)$ should cause a type error, because the *sal* function is not defined on the type *Pers*. Instead, we would use the COOL operation **gain** $[Empl](p)$ that makes the person instance of the type *Empl* such that the salary can be assigned. Notice, however, that we have to cope with undefined function values, if there is no mechanism for default values associated with type changes. In this paper, we only point out where and how to integrate such a mechanism, but leave the actual integration as future work.

As usual in object models, more complex updates can be performed by methods as combinations of the above operations. Again, we do not propose a full method language here, but only employ the two most basic constructs: besides sequencing, the iterator **apply_to_all** provides set-oriented application of update sequences with deterministic semantics [13]:

$$
\begin{array}{lll}
u = & u\ ;\ u & /*\ \text{composition of updates}\ */ \\
 & \mid \mathbf{apply\_to\_all}\ [u](e) & /*\ \text{set-oriented application of updates}\ */
\end{array}
$$

3 Semantics of BCOOL

In the denotational approach, the semantics of a language is defined by (higher order) functions [24]. A "semantic domain" represents interpreted constructs used as the "denotation" (semantics) of syntactic constructs. For each syntactic construct, such as constants, expressions and statements, a function is given that maps syntax to semantics. In particular, for updates the target of this denotation function is again a function (from an old to a new state).

- The state function σ represents the current database state. It captures the following information: (i) the current instances for each object type (the active domains), (ii) the values of all possible function applications (F denotes the set of functions defined in the schema below), and (iii) the values of all variables (X denotes the set of variables defined in the schema, which is a superset of F, because functions are regarded as variables over function types).

- The typing function A is used to represent the type declarations of variables and functions: for example, $A(x)$ returns the type expression τ for a variable declaration **var** $x : \tau$.

- The domain function $[\![\ ]\!]$ returns the semantics (the domains) of type expressions τ (see Sec. 3.1).

- The expression function $E[\![\ ]\!]$ returns the value of expressions in the current state σ (see Sec. 3.2).

- The update function $U[\![\ ]\!]$ turns the state functions for update expressions (see Sec. 3.3).

3.1 Semantics of Type Expressions, Subtyping

The semantic domain of values is defined by the following recursive domain equations:

$\mathcal{V} = \mathcal{B}_{Bool} \cup \mathcal{B}_{Int} \cup \mathcal{B}_{String} \cup \mathcal{B}_{Object} \cup \mathcal{F} \cup \mathcal{S}$
$\mathcal{B}_{Bool} = \{\perp_{Bool}, true, false\}$,
$\mathcal{B}_{Int} = \{\perp_{Int}, 0, 1, -1, 2...\}$,
$\mathcal{B}_{String} = \{\perp_{String}, "a", "A", ...\}$,
$\mathcal{B}_{Object}$ contains countably infinite objects,
$\mathcal{F} = \mathcal{B}_{Object} \rightarrow_{fin} \mathcal{V}$,
$\mathcal{S} = P_{fin}(\mathcal{V})$.

$\mathcal{B}_i$ are domains of basic values (e.g., boolean and integer). $\mathcal{F}$ denotes the domain of finite mappings from $\mathcal{B}_{Object}$ to $\mathcal{V}$, and $\mathcal{S}$ all finite powersets over $\mathcal{V}$. The type specific bottom elements ($\perp_i$) denote undefined values. In order to improve readability we omit the type information and use $\perp$ instead.

In general, equality must be defined for types on which sets are constructed (e.g., for testing set-membership). Because the equality for function types is undecidable in general, the domains of function types are restricted to objects. The equality on these restricted functions would be still undecidable, because the domain $\mathcal{B}_{Object}$ is infinite. However, since all instances of functions that can ever occur in any database state are restricted to the *active domains* of the corresponding object types (which are *finite* sets), all functions can be regarded as finite sets of pairs, such that equality is decidable. Hence, we do not need to separate types with equality from those without equality, which would be necessary otherwise.

3.1.1 Basic and Constructed Types

Except for object types, our semantics of types and subtyping is quite usual and follows [3, 4, 14]: i.e., the denotations of basic types are given by the following equations:

Definition Semantic Domain:

$[\![\beta_i]\!] = \mathcal{B}_i$, in case that $\mathcal{B}_i$ is a summand of $\mathcal{V}$
$[\![\{\ \tau\ \}]\!] = \{x \in \mathcal{S} \mid x \subseteq [\![\tau]\!]\} \in \mathcal{V}$
$[\![\tau_1 \rightarrow \tau_2]\!] = \{f \in \mathcal{F} \mid x \in [\![\tau_1]\!] \Longrightarrow f(x) \in [\![\tau_2]\!]\ \} \in \mathcal{V}.$

The subtype relationship ($\preceq$) is based on set inclusion: i.e., if a type is defined as a subtype of another, then every instance of the subtype is also an instance of its supertype (which allows *substitutability*): $\tau_1 \preceq \tau_2 \Longrightarrow [\![\tau_1]\!] \subseteq [\![\tau_2]\!]$. This leads to the following inference rules for constructed types:[3]

Definition Subtyping:

$$[\text{SETS}]\ \frac{\tau_1 \preceq \tau_2}{\{\tau_1\} \preceq \{\tau_2\}} \qquad\qquad [\text{FUNS}]\ \frac{\tau_1^{dom} \preceq \tau_2^{dom},\ \tau_2^{rng} \preceq \tau_1^{rng}}{\tau_2^{dom} \rightarrow \tau_2^{rng} \preceq \tau_1^{dom} \rightarrow \tau_1^{rng}}$$

3.1.2 Object Types

Let us now focus on the semantics of object types. In Section 3.1 we argued to use the basic type $\mathcal{B}_{Object}$ as the domain of all functions, such that function signatures are homogeneous. Then, however, type-checking is less meaningful, because each function may be applied to any object. The more specific function domains are, the more errors are detected at compile-time. Therefore, we use subtypes $[f_1, ..., f_n]$ of $\mathcal{B}_{Object}$. In particular, there is a object type $[f_1, ..., f_n]$ for any subset of F that contains the functions defined in a database schema. In order to get concise type inference rules in Section 3.2, the object types are arranged to a lattice. Intuitively, applications of the function f on instances of $[f_1, ..., f_n]$ pass static type-checking, iff f is contained in $\{f_1, ..., f_n\}$.

Nonetheless, the *semantic domains* of all object types are the same, in order to allow for object evolution, such that objects can gain and lose instance relationships dynamically.

In contrast to instances of data types like integers, objects can be created and deleted. That is, it is not possible to refer to arbitrary instances of object types, rather only to those that have been created and not yet deleted. Therefore, the domains of object types (denoted by $[\![[f_1, ..., f_n]]\!]$) have to be distinguished from the *active domains* (denoted by $\sigma([f_1, ..., f_n])$) that contain the instances of these types in the current state σ.

Thus, we use the state function in order to refer to active domains already in this section, though its definition is given in Section 3.3 by the semantics of operations that manipulate the active domain. This is done, because the notion of active domains is not only important for the restriction of function types (as discussed above), but also for the definition of subtyping on object types. Notice that only the active domains of the type without functions ($[]$) and the types with only one function ($[f_i]$) are explicitly maintained. The other

[3] The horizontal bar corresponds to logical implication. Notice the antimonotonicity (contra-variance) in [FUNS], which is needed for the set-inclusion semantics of the subtype relationship.

active domains (of types with more functions $[f_i, f_j, \ldots]$) are derived from the former according to the type lattice (see below).

There are two requirements for the definition of the active domains:

- the active domain has to be a subset of the domain in any state: [4] $\sigma([f_1, ..., f_n]) \subset [\![[f_1, ..., f_n]]\!]$.
- the subtype relationship has to capture the substitutability of objects: $[f_1, ..., f_n] \preceq [f'_1, ..., f'_m] \implies \sigma([f_1, ..., f_n]) \subseteq \sigma([f'_1, ..., f'_m])$

The second requirement adapts the usual notion for subtyping to object types, which implies a subset relationship on the respective domains (see above). In particular, this adaptation is important, because the domains of object types have to be the same in order to allow for object evolution.

After this motivation of object types, let us now define the semantic domains and subtyping for object types.

Definition Semantic Domains:

$[\![[f_1, ..., f_n]]\!] = [\![[\,]]\!] = B_{Object}$, for $\{f_1, ..., f_n\} \subseteq F$.

The subtype relationship between object types is defined on the superset relationship between their function sets, as follows:[5]

Definition Subtyping:

$[f_1, ..., f_n] \preceq [f'_1, ..., f'_m] \iff \{f_1, ..., f_n\} \supseteq \{f'_1, ..., f'_m\}$.

In order to make type-inference more concise, object types are arranged in a lattice:

Definition Object Type Lattice:

The set of object types forms a lattice, where the subtype relationship is the partial order. The least upper bound ($\sqcup$), and the greatest lower bound ($\sqcap$) are defined as follows:

$$\begin{aligned}
[f_1, ..., f_n] \sqcup [f'_1, ..., f'_m] &= [f''_1, ..., f''_l], \\
\text{where } \{f''_1, ..., f''_l\} &= \{f_1, ..., f_n\} \cap \{f'_1, ..., f'_m\}. \\
[f_1, ..., f_n] \sqcap [f'_1, ..., f'_m] &= [f''_1, ..., f''_l], \\
\text{where } \{f''_1, ..., f''_l\} &= \{f_1, ..., f_n\} \cup \{f'_1, ..., f'_m\}.
\end{aligned}$$

Finally, let us define the active domains of object types containing more than one function dependent on the types with exactly one function. (The active domains of object types with none or one function are defined in Section 3.3).

Definition Active Domains $[f_1, \ldots, f_n]$ **(for** $n \geq 2$**):**

$$\sigma([f_1, ..., f_n]) = \bigcap_{f \in \{f_1, ..., f_n\}} \sigma([f]).$$

Notice that this definition guarantees the subset relationship between the active domains of a subtype and its supertypes (see the second requirement for active domains above), because of the superset relationship between their function sets:

$$[f_1, ..., f_n] \preceq [f'_1, ..., f'_m] \implies \{f_1, ..., f_n\} \supseteq \{f'_1, ..., f'_m\}$$
$$\implies \sigma([f_1, ..., f_n]) \subseteq \sigma([f'_1, ..., f'_m]).$$

[4] Because all active domains are finite sets, the subset relationship is always proper.

[5] Notice that this definition captures inheritance of interfaces in the usual way.

3.2 Semantics of Expressions

In this section, we define the type inference rules and the semantics of expressions. First, expressions are reduced to syntactically correct ones by type inference rules. Secondly, the semantics of expressions is defined by the denotation function E, that returns the value $E[\![e]\!]_\sigma$, which is an element of a component of the semantic domain $\mathcal{V}$, by evaluating the expression e in the current state σ.

In order to improve readability, we often simplify the notation of type inference rules by omitting the assertions of variable declarations (included in the typing function A) and subtyping information, which are part of the premisses. Similarly, we simplify the semantic denotations by leaving out preconditions and result types that are covered by type inference rules, and omit the current state σ in cases where it is only used as an input parameter of the function E. Additionally, we use the domains instead of the active domains in the denotations. Notice, however, that the semantics of update operations guarantees that values are restricted to the active domains.[6]

Constants and Variables:

$$c :: \tau \qquad\qquad E[\![c]\!] = C, \text{ where } C \text{ is a constant } \in [\![\tau]\!].$$

$$\frac{A(x) = \tau}{x :: \tau} \qquad\qquad E[\![x]\!]_\sigma = \sigma(x) \in [\![\tau]\!].$$

New creates new objects. That is, the application of the operation **new** yields an object that does not yet occur in the current state of the database. Formally, the "invention" of objects is non-deterministic, such that $E[\![\ ]\!]$ is a mapping rather than a function (as claimed above). This non-determinism could be eliminated, for example, by assuming an order on objects (such that **new** yields the "smallest" not yet created object). However, since the object identities are not visible on the BCOOL level, we do not need to care about different states that are isomorphic up to renaming object identities (see the notion of O-isomorphisms in [2]). Therefore, we take the freedom to allow the non-deterministic invention of object identities: This is certainly desirable on the implementation level, furthermore it allows **new** operations to be executed in parallel within an **apply_to_all** statement [13].

$$\textbf{new}\,() :: \beta_{Object} \qquad\qquad E[\![\textbf{new}\,()]\!]_\sigma = o \textbf{ with } o \in [\![[\,]]\!] \wedge o \notin \sigma([\,]).$$

Active domains of object types are returned by the operation **adom**:

$$\frac{[f_1, ..., f_n] \preceq \beta_{Object}}{\textbf{adom}\,([f_1, ..., f_n]) :: \{[f_1, ..., f_n]\}} \quad E[\![\textbf{adom}\,([f_1, ..., f_n])]\!]_\sigma = \sigma([f_1, ..., f_n]).$$

Lambda abstractions define functions on the active domains of object types, where $\sigma' = \sigma\{v/x\}$ denotes the substitution of v for x, i.e., it is identical to σ except that $\sigma'(x) = v$. This form of abstractions does not cause problems w.r.t. to the equality test on functions, because the free variable x is restricted to the active domain of the respective object type. Thus, functions can be regarded as

[6] In particular, this is also true for values constructed on object types such as sets of persons.

finite sets of pairs[7]. The type inference rule means that $e :: \tau_2$ can be inferred under the assumption that x is a variable of the object type τ_1:

$$\frac{A(x) = \tau_1, \tau_1 \preceq \beta_{Object} \vdash \ e :: \tau_2}{\lambda x : \tau_1.e :: \ \tau_1 \ \rightarrow \ \tau_2} \quad E[\![\lambda x : \tau_1.e]\!]_\sigma = \{\langle v, E[\![e_{\{v/x\}}]\!]\rangle \mid v \ \in \ \sigma(\tau_1)\}.$$

Function applications return the function value if defined. Notice that there is no difference whether a tuple with the null value ($\perp$) as second component is included in the set of function tuples or not:

$$\frac{f :: \ \tau_1 \ \rightarrow \ \tau_2, \ e :: \tau_1}{f(e) :: \ \tau_2} \qquad E[\![f \ (e)]\!] = \begin{cases} v & \text{if } \langle E[\![e]\!], v\rangle \ \in \ E[\![f]\!], \\ \perp & \text{otherwise.} \end{cases}$$

Sets can be constructed by including an expression in braces:

$$\frac{e :: \tau}{\{e\} :: \ \{\tau\}} \qquad E[\![\{ \ e \ \}]\!] = \{E[\![e]\!]\}.$$

Pick is used to deconstruct sets, i.e., to get rid of the braces in case of singleton sets. Notice that **pick** is not deterministic in case of sets with more than one element. In order to avoid a non-deterministic semantics in case that the set contains more than one object, we could restrict the applicability (which, however, can only be checked at run-time) or assume an order on objects (see above).

$$\frac{e :: \{\tau\}}{\textbf{pick}\,(e) :: \ \tau} \qquad E[\![\textbf{pick}\,(e)]\!] = \begin{cases} v \ \in \ E[\![e]\!] & \text{if } E[\![e]\!] \ \neq \emptyset, \\ \perp & \text{otherwise.} \end{cases}$$

Set membership can be tested by the predicate $\in$:

$$\frac{e_1 :: \tau, e_2 :: \{\tau\}}{e_1 \in e_2 :: \ \textbf{bool}} \qquad E[\![e_1 \in e_2]\!] = \begin{cases} \mathit{true} & \text{if } E[\![e_1]\!] \ \in E[\![e_2]\!], \\ \mathit{false} & \text{otherwise.} \end{cases}$$

Unions of two sets result in a set that is associated to the least upper bound of the element types:

$$\frac{e_1 :: \{[f_1, ..., f_n]\}, \ e_2 :: \{[f_1', ..., f_m']\}}{e_1 \cup e_2 :: \{[f_1, ..., f_n] \ \sqcup \ [f_1', ..., f_m']\}} \qquad E[\![e_1 \cup e_2]\!] = E[\![e_1]\!] \ \cup \ E[\![e_2]\!].$$

Intersections of two sets are related to the greatest lower bound of the element types:

$$\frac{e_1 :: \{[f_1, ..., f_n]\}, \ e_2 :: \{[f_1', ..., f_m']\}}{e_1 \cap e_2 :: \{[f_1, ..., f_n] \sqcap [f_1', ..., f_m']\}} \qquad E[\![e_1 \cap e_2]\!] = E[\![e_1]\!] \ \cap \ E[\![e_2]\!].$$

Selections are used to specify subsets of e_2 according to the predicate $\lambda x.e_1$. Notice that this is a restricted case of lambda abstractions, because the free variable x, which may occur in e_1, ranges over the elements in e_2: [8]

[7] One might argue that functions are total, because they yield the $\perp$ value except for a finite set of arguments. Notice, however, that there is no means to refer to instances of β_{Object} that are not contained in the active domain.

[8] Notice that differences between sets can be expressed by a selection predicate that checks whether an object is not contained in a set.

$$\frac{\lambda x : \tau.e_1 :: \tau \rightarrow \mathbf{bool}, e_2 :: \{\tau\}}{\mathbf{select}[\lambda x.e_1](e_2) :: \{\tau\}} \quad E[\![\mathbf{select}[\lambda x.e_1](e_2)]\!] = \{v \in E[\![e_2]\!] \mid E[\![e_{1\{v/x\}}]\!]\}.$$

Projections restrict the interfaces of set elements as in the relational algebra (see also transformational filters in [1]). In contrast to projections in [9, 23, 25] that generate objects or values, **project** is defined with object preserving semantics [21]. Therefore, it can be used for hiding information, similar to assignments of instances to variables of a supertype:

$$\frac{e :: \{[f'_1, ..., f'_m]\},\ [f'_1, ..., f'_m] \preceq [f_1, ..., f_n]}{\mathbf{project}\,[f_1, ..., f_n]\ (e) :: \{[f_1, ..., f_n]\}} \quad E[\![\mathbf{project}\,[f_1, ..., f_n]\,(e)]\!] = E[\![e]\!].$$

Qualified Expressions allow for a macro mechanism, such that the expression e_2 can be substituted in e_1 by f. In particular, this macro mechanism can be used, for example, to express joins by virtual functions:[9]

$$\frac{e_2 :: \tau_2, A(f) = \tau_2\ \vdash e_1 :: \tau_1}{\mathbf{let}\ f = e_2\ \mathbf{in}\ e_1\ \mathbf{end}\ :: \tau_1} \qquad E[\![\mathbf{let}\ f = e_2\ \mathbf{in}\ e_1\ \mathbf{end}\ ]\!]_\sigma = E[\![e_{1\{e_2/f\}}]\!].$$

Usually, operations of an algebra are orthogonal to each other, such that non of them can be defined by the others. However, up to now the operation **project** can be defined by other operations as follows:

$\mathbf{project}\,[f_1, ..., f_n](e) \equiv \mathbf{select}\,[\lambda x.x \in e](\mathbf{adom}\,([f_1, ..., f_n]))$

The reason for this lack of orthogonality is that casting the interface of objects has been bundled together with set-orientation. The extension of the relational algebra with object types (that are arranged in a lattice) needs active domains. These can be used together with operations like $\cap$ and $\cup$ that are defined with respect to the type lattice. Therefore type inference is more powerful than in relational algebra, in which set operations require the same schema on the input relations. That is, the problem originates from the extension of the relational algebra, in which operations are defined on sets, because of the potential for optimization. However, if not only sets but also other type constructors are integrated into the model, **project** is decoupled from sets, such that it only casts the type of a single object (see Sec. 4.1).

Examples: Finally, let us illustrate how joins can be expressed in BCOOL by the following two examples. In the first example, we are looking for employees that are managed by their parents. In order to improve readability, we use COOL type names (such as *Empl*) instead of the respective BCOOL function sets ($[name, age, children, sal, manager]$). In the relational algebra this query would involve the join operation (if the schema fulfills at least third normal form). However, in object algebras composition of functions can be explored:

$\mathbf{select}\,[\lambda x.x \in children(manager(x))](\mathbf{adom}\,(Empl))$.

In the second example we make use of a virtual function for a more complex join.[10] All employees that have the same manager as x are collected in the

[9] The type inference rule means that $e_1 :: \tau_1$ can be inferred under the assumption that f is a variable of type τ_2.

[10] See [21] for a discussion of different alternatives for joins: symmetric tuple/object generating; asymmetric as functions.

function *colleagues*, whose scope is limited by the **let** operation. If we are interested in all employees who have at least one colleague, the virtual function *colleagues* can be used in the subsequent selection:

$$\begin{aligned}&\textbf{let}\ colleagues = \lambda x : [manager].\ \textbf{select}\ [\lambda y.\ manager(y) = manager(x) \wedge \\ &\qquad\qquad\qquad\qquad\qquad\qquad x \neq y](\textbf{adom}\ (Empl)) \\ &\textbf{in}\,\textbf{select}\ [\lambda x.\ colleagues(x) \neq \emptyset](\textbf{adom}\ (Empl))\ \textbf{end}\ .\end{aligned}$$

3.3 Semantics of Update Operations

Update operations are defined by a function U that maps the old state σ onto the new one: $U[\![\, upd\text{-}op \,]\!]_\sigma$. The definitions require that certain typing restrictions are fulfilled, which are notated as preconditions. Since these preconditions can be verified by the static type-checker already at compile-time, only update statements that fulfill the restrictions are executed. In the COCOON model the semantics of update operations is defined w.r.t. model inherent constraints (e.g., sub-/typing, class membership, and class predicates) [12]. That is, applying an update operation to a consistent database state returns a consistent state. Similarly, the update operations of BCOOL are defined w.r.t. typing and subtyping constraints. For example, if an object is deleted, "dangling references" are avoided, and removing or adding instance relationships is propagated to subtypes or supertypes, respectively.

Assignment. Variables can be bound to new values by an assignment (:=). The new state is the same as the old one for all variables, types, and functions except for the variable x. The precondition ensures the substitutability of x's value (ϕ ranges over variables, functions, and object types):

Precondition: $x \in X$ and $e :: \tau$ and $\tau \preceq A(x)$.

$$U[\![\, x := e \,]\!]_\sigma(\phi) = \begin{cases} E[\![\, e \,]\!]_\sigma & \text{if } \phi = x, \\ \sigma(\phi) & \text{otherwise.} \end{cases}$$

Partial Assignments. Function values can be changed by partial assignments (**set**) for a single argument:

Precondition: $f :: [f] \rightarrow \tau_r$ and $e' :: \tau'$ and $\tau' \preceq [f]$ and $e :: \tau$ and $\tau \preceq \tau_r$.

$$U[\![\, \textbf{set}\ [f := e](e') \,]\!]_\sigma(\phi) = \begin{cases} f' & \text{if } \phi = f, \\ \sigma(\phi) & \text{otherwise,} \end{cases}$$

$$\textbf{with}\ f'(\psi) = \begin{cases} E[\![\, e \,]\!]_\sigma & \text{if } \psi = e', \\ \sigma(f)(\psi) & \text{otherwise.} \end{cases}$$

The **set** operation only affects the value of the function f. It is substituted by a new function value f' that differs from f only for the argument designated by the expression e', for which the result is e.

In the following, we define the semantics of update operations that manipulate the active domains of object types. The existence of objects can be manipulated by **new** and **destroy** , and is captured by the membership in the set $\sigma([\,])$, which is the active domain of the most general object type. The active domains of object types including a single function are manipulated by the operations **gain** and **lose**.[11]

[11]The active domains of types with more than one function have been defined in Sec. 3.1.

New. The creation of an object by **new** () instantiates the top element of the object type lattice (β_{Object}) with a new object. That is, the active domain of this type is extended by the object that is the return value of the operation:[12]

$$U[\![\,\mathbf{new}\,()\,]\!]_\sigma(\phi) = \begin{cases} \sigma(\phi) \cup \{E[\![\,\mathbf{new}\,()\,]\!]_\sigma\} & \text{if } \phi = \beta_{Object} \\ \sigma(\phi) & \text{otherwise.} \end{cases}$$

Gain. An existing object can be made instance of additional object types by the operation **gain** $[f_1, ..., f_n](e)$ that makes each function in $[f_1, ..., f_n]$ applicable to the object e:[13]

Precondition: $e :: \tau'$and $\tau' \preceq \beta_{Object}$ and $\{f_1, ..., f_n\} \subseteq F$.

$$U[\![\,\mathbf{gain}\,[f_1, ..., f_n](e)\,]\!]_\sigma(\phi) = \begin{cases} \sigma(\phi) \cup \{E[\![\,e\,]\!]_\sigma\} & \text{if } \phi = [f] \wedge f \in \{f_1, ..., f_n\} \\ \sigma(\phi) & \text{otherwise.} \end{cases}$$

That is, the object becomes instance of the active domains $\sigma([f])$ for all f in $\{f_1, ..., f_n\}$, which propagates to object types with more than one function according to the definition in Sec. 3.1. In other words, if the type $[f'_1, ..., f'_m]$ has been the least upper bound of all types the object has been instance of, then the object becomes instance of $[f_1, ..., f_n] \sqcap [f'_1, ..., f'_m]$ and all its supertypes.

Example: Let us consider a person object denoted by the variable p that is of type *Pers*[14]. The person p can be made instance of the type *Empl* by the following **gain** operation that makes the functions *sal* and *manager* applicable to p:

gain $[sal, manager](p)$

Up to now the function values $sal(p)$ and $manager(p)$ are undefined ($\perp$). However, a mechanism for providing default values could be integrated here very easily by a corresponding extension of syntax. Later on the employee might become a manager in a similar way.

Lose. In contrast to the **gain** operation, instance relationships can be deleted by **lose**. The effect of **lose** $[f_1, ..., f_n](e)$ is that all functions in $\{f_1, ..., f_n\}$ are no longer applicable to the object denoted by e.

Precondition: $e :: \tau'$ and $\tau' \preceq \beta_{Object}$ and $[f_1, ..., f_n] \preceq \beta_{Object}$.

$$U[\![\,\mathbf{lose}\,[f_1, ..., f_n](e)\,]\!]_\sigma(\phi) = \begin{cases} \sigma(\phi) \setminus \{E[\![\,e\,]\!]_\sigma\} & \text{if } \phi = [f] \wedge f \in \{f_1, ..., f_n\} \\ \sigma(\phi) & \text{if } \phi = [f] \wedge f \notin \{f_1, ..., f_n\} \\ nv(\sigma(\phi), A(\phi)) & \text{otherwise.} \end{cases}$$

Thus, the state after an operation **lose** $[f_1, ..., f_n](e)$ can be derived in two steps. First, the object denoted by the expression e is excluded from the active

[12]Notice that $E[\![\mathbf{new}()]\!]_\sigma$ denotes the return value of the operation (which is the newly created object) whereas $U[\![\mathbf{new}()]\!]_\sigma$ denotes the intermediate state after the evaluation of $E[\![\mathbf{new}()]\!]_\sigma$ and before the execution of the statement in which **new**() is used.

[13]In contrast to **new**, the operation **gain** is defined as a statement that does not return a value. However, one can define a macro **gain'** that returns the input object with the new type by the following definition: $\mathbf{gain}'[f_1, ..., f_n](e) \equiv$ **pick** (**select** $[\lambda x.x = e]$(**adom** $([f_1, ..., f_n])$)).

[14]Again we use the COOL type names instead of the respective function sets in order to improve readability.

domains of the types $[f_i]$ $(i = 1, ..., n)$; see the first case). This automatically propagates to the other object types.

The second step of the definition refers the occurrences of the object within values of variables, sets, and functions. Due to strong typing we have, for example, to exclude that the function f_i is applied to the object e, which might be the value of a variable. In general, this constraint is assured by removing each occurrence of the object denoted by e from variables, sets and functions, if these are related to a type that contains at least one function in $\{f_1, ..., f_n\}$. Therefore, the values of all variables (and functions) are recursively derived from the new active domains by the function nv.

The function nv is applied to an old value v and its type τ and returns the new value $nv(v, \tau)$. The new value is different form the old one only if the old one would not fulfill the type constraints. Therefore, the function is defined as follows:

$$nv(v,\tau) = \begin{cases} \perp & \text{if } \tau \preceq \beta_{Object} \\ & \wedge\, v \notin U[\![\mathbf{lose}\,[f_1, ..., f_n](e)]\!]_\sigma(\tau) \\ \bigcup\limits_{v' \in v} nv(v', \tau') & \text{if } \tau = \{\tau'\} \\ \{\langle x, nv(v', \tau_2)\rangle \mid \langle x, v'\rangle \in v\, \wedge & \text{if } \tau = \tau_1 \rightarrow \tau_2 \\ \qquad\qquad x \in U[\![\mathbf{lose}\,[f_1, ..., f_n](e)]\!]_\sigma(\tau_1)\} & \\ v & \text{otherwise.} \end{cases}$$

The idea of the derivation is to use the structure of types in order to reduce the problem of specifying the new value of sets and functions to easier cases. In the first case, the old value, which is not element of the type τ anymore, is replaced by the null value ($\perp$). This case together with the last case, in which all remains the same, are the anchors of the derivation. If the old value is a set (the second case), the derivation is evaluated for each element of this set recursively. The return value of the set is constructed by the union over all elements[15]. Similarly, the value of each function is also checked recursively (the third case).

Notice that the **lose** operation and the deletion of objects have a strong impact on constructed values. Without objects, instances of constructed types such as sets and functions are regarded as values that can not be created or deleted [6]. However, if object types are used to construct sets of functions, the constructed domains become dynamic as well, since the existence of constructed values depends on the active domains of object types.

Example: Assume the variable declarations

$p ::$	$Pers;$
$e ::$	$Empl;$
$mgr ::$	$Mngr;$
$jones\_pers ::$	$\{Pers\};$
$jones\_empls ::$	$\{Empl\};$

and a state in which an object named "Jones", which is manager of an employee denoted by e. After the following assignments, the variables p and mgr denote the same object Jones, which is also contained in both set variables $jones\_pers$ and $jones\_empls$:

[15]The union ignores null values as elements, i.e., $S \cup \{\perp\} = S$.

$jones\_pers :=$ **select** $[name = \text{"}Jones\text{"}](\textbf{adom}\,(Pers))$;
$jones\_empls :=$ **select** $[name = \text{"}Jones\text{"}](\textbf{adom}\,(Empl))$;
$p :=$ $manager(e)$;
$mgr :=$ $manager(e)$;

If Jones retires or is fired, the functions associated to the type *Empl* and its subtypes must not applied to him/her anymore. This can be achieved by the operation

lose $[sal, manager, budget](p)$

that removes Jones from the respective active domains. The propagation to variables, sets, and functions leads to the state in which Jones is still contained in *jones_pers*, but is removed from *jones_empls* (because of the type declaration that would allow to apply e.g., the *sal* function to elements of *jones_empls*). Similarly, the variable p still denotes Jones, whereas mgr is undefined as well as the function application $manager(e)$.

Destroy. In contrast to **lose** , the **destroy** operation has an effect on the existence of objects. The operation **destroy** (e) removes e from the active domains of all object types, which propagates to variables, sets, and functions:

Precondition: $e \;::\; \tau'$ and $\tau' \;\preceq\; \beta_{Object}$.

$$U[\![\, \textbf{destroy}\ (e)\,]\!]_\sigma(\phi) = \begin{cases} \sigma(\phi) \setminus \{E[\![\, e\,]\!]_\sigma\} & \text{if } \phi = [f] \,\wedge\, f \,\in\, F \text{ or } \phi = \beta_{Object} \\ nv(\sigma(\phi), A(\phi)) & \text{otherwise.} \end{cases}$$

Notice that the semantics of these operations is defined with respect to the requirements for active domains in Sec. 3.1. That is, **new** and **gain** are defined such that the subset relationship between active domains and their respective domain is valid. The definitions of **lose** and **destroy** guarantee that no "dangling references" occur. That is, the procedure nv "removes invalid objects from values". Therefore the state only contains objects that are elements in active domains. Finally, the subset relationship between subtypes and their supertypes is guaranteed by the definition of active domains for object types with more than one function (in Sec. 3.1) together with the definitions in **gain** and **destroy** .

4 Relationship to functional models

In this section we discuss the similarities between object algebras (such as BCOOL) and functional models (such as ML [15], and Machiavelli [7, 16]). On the one hand they have enough in common that a combination of both is possible; on the other hand the challenge is whether the advantages of either model can be preserved in the integration. We show (i) how to add full orthogonality and genericity to BCOOL and (ii) discuss how to extend Machiavelli by the concept of objects, which is more than just a reference.

4.1 Extending Object Query Languages by Orthogonality and Genericity

Up to now most operations of object query algebras (including BCOOL) center around sets. This has been done in order to take advantage of the optimization capabilities that arise from descriptive set-oriented queries and have already been used in relational systems. However, since many applications need other type constructors than sets (such as lists, tuples, and arrays), these constructors should also be integrated into object data models. Moreover, instead of a fixed collection of constructors, the variety of constructors should be extensible.

Therefore, let us discuss as a first extension, how BCOOL can be extended such that type constructors or generic types can be defined. That is, in this paper we omit other extensions of BCOOL, e.g., extensions of the algebra towards a programming language by providing operations such as loop and conditional instruction.

In the previous sections, we have concentrated on sets. Therefore, some of the proposed query operations are not orthogonal w.r.t. to type constructors other than sets. That is, operations that work on sets of objects have to be separated into elementary operations associated to type constructors from operations on objects, for example. This is similar to the separation of update operations into the **apply_to_all** iterator [13] and a few elementary update operations defined in the previous section. The iterator is used to apply sequences of elementary update operations to all elements of a set deterministically.

Consequently, the BCOOL set-oriented query operations **project** should be redefined to work on single objects (denoted by $'$) such that they can be used not only within sets:

$$\frac{e :: [f'_1, ..., f'_m]}{\mathbf{project}'\,[f_1, ..., f_n]\ (e) :: [f_1, ..., f_n]} \qquad E[\![\,\mathbf{project}'\,[f_1, ..., f_n]\ (e)\,]\!] = E[\![\,e\,]\!]$$

Then, if a set-iterator **map** is provided (akin to *hom* in Machiavelli [7], *replace* [1] or *pump* in FAD [5]), the previous set-oriented operations can, for example, be derived as [16]:

$$\mathbf{project}\,[f_1, ..., f_n](S) \ \equiv\ \mathbf{map}[\mathbf{project}'\,[f_1, ..., f_n](e)](e : S).$$

Now we can easily extend BCOOL to work on lists of objects. After defining the constructors and deconstructors of lists, as well as an iterator on lists (for example, **lmap**), a projection on lists can be defined by the combination of **project$'$** and **lmap**. Similarly we can add subtyping rules for lists (akin to [SETS] and [FUNS] in Sec. 3.1) if necessary.

In general, the inclusion of a new type constructor requires the specification of subtyping rules and operations on the types' instances (including constructors and deconstructors). Notice that there are no update operations on instances of constructed types, because—as values—they are not created, updated, or deleted explicitly, but are rather constructed from components. However, if the type variable of constructors can be instantiated by object types, a mapping is required how constructed values "including" an object are mapped onto another value, if this object is deleted (similar to the mapping for sets and functions in the definition of the **lose** operation in Sec. 3.3).

Notice that the gain of orthogonality does not necessarily decrease performance, if we use overriding in the implementations. For example, efficient implementations of the relational algebra or any object algebra could be used

[16] Similarly, **select** can be defined by an iterator such as *hom*, if an **if-then-else** construct is provided in the model.

for the all instances of sets of tuples (sets of objects), respectively. Thus, the system can still make use of the optimization techniques provided for standard database models.

Another lack of orthogonality is the restriction that functions can only be defined on object types. In an orthogonal type system, one would expect that any type can serve as function domain. This extension, however, causes problems, since equality of functions is undecidable in general. Then, constructing sets over function types has to be prohibited, because the test for set-membership requires the equality test on the elements. A possible solution is the separation of types with an without equality known from functional languages (e.g., ML, Machiavelli), which could easily be employed in BCOOL.

4.2 Integrating objects in functional data models

One important objective in functional languages is to be purely descriptive and avoid imperative statements. Therefore, no explicit update operations are provided in purely functional languages (e.g., Miranda [26]). Consequently, sharing is also no important issue, because it becomes relevant only through updates. Nevertheless, there are also functional languages in which references are provided in order to express sharing with a rudimentary update capability (e.g., ML [15] or its derivative Machiavelli [7]).

Before discussing the lacks of modeling objects by references, let us briefly illustrate the concepts that are essential for this modeling. In Machiavelli, references are used as pointers to typed data items with the following three generic operations: creating new references (**new**), de-referencing ("!"), and assignments (":="). In contrast to Cardelli and Wegner's type system, in which values might be instances of multiple types [8], values in Machiavelli are instances of *one unique* type. However, kinds are provided to describe sets of types with common properties, such that type-checking in fact refers to kinds. That is, all instances of the subtypes of $Pers$ are regarded as instances of a set of types that is described by the kind that allows the application of the $Pers$ functions. Therefore, the semantics of subtyping is captured by kinds.

However, because references have not been considered when the type system has been extended to kinds, the following problems arise, if references are used to model objects:

- An object might be instance of different types (that are usually related by the subtype relationship). Therefore, references to the same object with different types must be provided. This can be done either by using the most specific type in the declaration of the referenced type or by using different references for different records that describe the properties of the same object. In the first approach, the bottom type of the type lattice is required in order to allow for object evolution. The second approach is similar to "object specialization" [22], in which the same real world entity is represented by several objects. There, the problem is, that the system has to keep track, which references belong to the same object. If the real world object is deleted, this has to be propagated to all references; the substitutability might require that a reference must be mapped onto another; and equality of references to objects needs more than comparing two object identities. That is, the advantages of objects according to sharing and substitutability are lost.

- Considering the longevity and dynamics of persistent objects, object evolution becomes necessary. Machiavelli has no explicit means to delete objects, nor operations to add or remove objects to or from the active domain of types. Notice, however, that the lack of operations for object evolution is not particular to functional models, similar deficiencies are found in most object-oriented programming languages.

Therefore, let us discuss how objects could be integrated into Machiavelli akin to BCOOL objects. The general idea is to consider objects as references to kinded records. The extension of references to kinds of records simplifies substitutability and subtyping. The more powerful update facility results from adjusting Machiavelli's type constructor **ref**.

In more detail, the following two kinding rules are added to those in [7], such that the kind assignment $\mathcal{K}$ that maps types to kinds can be adapted to object types and their respective subtyping:

$$\mathcal{K} \vdash t :: \mathbf{ref}'([\![\, l_1 : \tau_1, ..., l_n : \tau_n \,]\!]) \quad \text{if } t \in domain(\mathcal{K}), \mathcal{K}(t) = \mathbf{ref}'([\![\, l_1 : \tau_1, ..., l_n : \tau_n, ...]\!])$$

$$\mathcal{K} \vdash \mathbf{ref}'([l_1 : \tau_1, ..., l_n : \tau_n, ...]) :: \mathbf{ref}'([\![\, l_1 : \tau_1, ..., l_n : \tau_n \,]\!])$$

Machiavelli's type constructor **ref** can be changed to **ref′** as follows: the generic operation **new** is adopted from BCOOL, such that it returns a newly created instance of $\mathbf{ref}'([\![\ ]\!])$. The operation for dereferencing is either adopted from Machiavelli, or dereferencing is done implicitly, if the dot operation is applied (which corresponds to BCOOL's function application). Similarly, assignments can be specific to components of objects (such as Machiavelli's *modify*) or common to all designators including variables and functions.

However, a fundamental consequence of using objects or references is that domains become dynamic. Without objects there is no need to separate the active domain from the domain, and no instances of any type could be created or deleted. Further, the domains of types that are constructed using objects become dynamic as well. However, it is not possible to invent an instance of a constructed type. That is, the domains of constructed types are static with respect to the active domains of their component types. Notice, however, that deletions cause problems: for example, a transformation is needed that maps a set containing object o to a set without o, if o is deleted. Therefore, we have to keep track of the active domains, similar to BCOOL. Then we can provide additional operations for **ref′** that allow for the evolution of objects:

- The semantics of $\mathbf{gain}\,[l : \tau](e)$ is to remove e from its current type $\mathbf{ref}'([l_1 : \tau_1, ..., l_n : \tau_n])$ and add it to the type $\mathbf{ref}'([l : \tau, l_1 : \tau_1, ..., l_n : \tau_n])$.

- The semantics of $\mathbf{lose}\,[l](e)$ is defined conversely. The object e is removed from its current type $\mathbf{ref}'([l : \tau, l_1 : \tau_1, ..., l_n : \tau_n])$ and added to the "supertype" $\mathbf{ref}'([l_1 : \tau_1, ..., l_n : \tau_n])$ that does not contain the component l.

5 Summary

The intended contribution of this paper is twofold: First, we proposed generic update operations for an object-oriented model that cope with object sharing and typing constraints of variables and functions, and allow to dynamically change the types of objects. Secondly, we discussed how functional models and object models could be combined such that object models gain flexibility by more orthogonality and functional models are extended by objects and their corresponding update operations.

Acknowledgements. We thank the referees, especially Limsoon Wong and Leonid Libkin, for their remarks on an earlier version of this paper.

References

[1] S. Abiteboul and C. Beeri. On the power of languages for the manipulation of complex objects. Technical Report 846, INRIA, Paris, May 1988.

[2] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 159–173, Portland, June 1989. ACM, New York.

[3] H. Balsters and C. C. de Vreeze. A semantic of object-oriented sets. In *Proc. of 3rd Intl. Workshop on Database Programming Languages*, pages 187–200, Nafplion, Greece, August 1991.

[4] H. Balsters and M. M. Fokkinga. Subtyping can have simple semantics. *Theoretical Computer Science*, 87:81–96, 1991.

[5] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. Int. Conf. on Very Large Databases*, pages 97–105, Brighton, September 1987.

[6] C. Beeri. Formal models for object-oriented databases. In W. Kim, J.-M. Nicolas, and S. Nishio, editors, *Proc. 1st Int'l Conf. on Deductive and Object-Oriented Databases*, pages 370–395, Kyoto, December 1989. North-Holland. Revised version appeared in "Data & Knowledge Engineering", Vol. 5, North-Holland.

[7] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 1993. to appear.

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

[9] K.C. Davis and L.M.L. Delcambre. A denotational approach to object-oriented query language definition. In *Proc. Int'l. Workshop on Specifications of Database Systems*, Glasgow, Scotland, June 1991. Workshops in Computing, Springer.

[10] D.H. Fishman, J. Annevelink, D. Beech, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Shan, and W.K. Wilkinson. Overview of the iris dbms. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 10, pages 371–394. ACM Press, Addison-Wesley, New York, 1989.

[11] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[12] C. Laasch and M.H. Scholl. Generic update operations keeping object-oriented databases consistent. In *Proc. of 2. GI Workshop Information Systems and Artificial Intelligence*, pages 40–55, Ulm, Germany, February 1992. IFB 303, Springer Verlag, Heidelberg.

[13] C. Laasch and M.H. Scholl. Deterministic semantics of set-oriented update sequences. In *Proc. of the IEEE Conf. on Data Engineering*, pages 4–13, Vienna, Austria, April 1993.

[14] M.V. Mannino, I.J. Choi, and D.S. Batory. The object-oriented functional data language. *IEEE Transactions on Software Engineering*, 16(11):1258–1272, November 1990.

[15] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Mass, 1990.

[16] A. Ohori, P. Buneman, and B. Breazu-Tannen. Database programming in Machiavelli a polymorphic language with static type inference. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 46–57, Portland, OR, May-June 1989.

[17] J. Richardson and P. Schwarz. Aspects: Extending objects to support multiple, independent roles. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 298–307, Denver, CO, May 1991.

[18] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137–147, jun 1986.

[19] M. H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proc. Int. Conf. on Deductive and Object-Oriented Databases (DOOD)*, pages 189–207, Munich, Germany, December 1991. LNCS 566, Springer Verlag, Heidelberg.

[20] M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, and M. Tresch. The COCOON object model. Technical Report 193, ETH Zürich, Dept. of Computer Science, 1992.

[21] M.H. Scholl and H.-J. Schek. A relational object model. In S. Abiteboul and P.C. Kanellakis, editors, *ICDT '90 – Proc. Int'l. Conf. on Database Theory*, pages 89–105, Paris, December 1990. LNCS 470, Springer Verlag, Heidelberg.

[22] E. Sciore. Object specialization. *ACM Trans. on Information Systems*, 7:103–122, April 1989.

[23] G.M. Shaw and S.B. Zdonik. A query algebra for object-oriented databases. In *Proc. of the IEEE Conf. on Data Engineering*, pages 154–162, Los Angelos, CA, February 1990.

[24] J. E. Stoy. *The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge (Mass.), 1977.

[25] D.D. Straube and M.T. Özsu. Queries and query processing in object-oriented databases. *ACM Transactions on Office Information Systems*, 8(4):387–430, October 1990.

[26] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proc. IFIP Int'l Conf. on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. LNCS 201, Springer.

[27] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris architecture and implementation. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):63–75, March 1990. Special Issue on Prototype Systems.