A Semantics of Object-Oriented Sets

Herman Balsters
balsters@cs.utwente.nl

Chris C. de Vreeze*
vreeze@cs.utwente.nl

Department of Computer Science
University of Twente
PO Box 217, NL 7500 AE Enschede
The Netherlands

Abstract

An account is given of extending the well-known object-oriented type system of Luca Cardelli with set constructs and logical formalism. The system is based on typed λ-notation, employing a subtyping relation and a powertype construct. Sets in this system are value expressions and are typed as some powertype. Sets are built up in a very general manner; in particular, sets can be described by (first-order) predicates. The resulting system, called LPT, is statically typecheckable (in a context of multiple inheritance) and is provided with a set-theoretic semantics. LPT can be used as a mathematical foundation for an object-oriented data model employing sets and constraints.

1 Introduction and results

In object-oriented database systems, object-oriented programming is merged with object-oriented data-structuring, enabling the designer of a database to have all the advantages of a clean conceptual design, as well as the possibility of enforcing better software engineering. Cardelli [1] argues that in order to integrate database systems and programming languages, it is first necessary to unify the database system (or information system) concept of *data model* with the programming-language concept of *type system*. Type systems such as the Cardelli object-oriented system [2] are often rich as far as expressiveness is concerned, and they have the additional advantage of being statically and efficiently typecheckable (this in contrast to, for example, set theory). Also the use of *subtyping* offers a concise and clear way to deal with (multiple) inheritance—a very powerful tool for organizing data. Multiple inheritance in the Cardelli type system is described for a collection of data types constructed from basic types, labeled products, disjoint sums, and function spaces. One big lack in Cardelli's type system is, however, that it offers no set constructions, and this is one of the reasons why it cannot be considered as a suitable candidate (as yet) for a mathematical foundation of an object-oriented data model to be used for database specifications. Set constructions are extremely useful in describing data structures pertaining to data modelling in databases. To quote François Bancilhon [3], "I think that treating sets uniformly is one of the major challenges facing designers of object-oriented database systems". Without sets, and in particular sets built up by a comprehension scheme (i.e. predicatively described sets of the form $\{x \mid \varphi(x)\}$, where $\varphi(x)$ is some first-order formula), one has hardly any possibility to define a collection of record expressions (the most important kind of expressions in a Cardelli-like database) satisfying some constraint (a predicate). With set constructs one also has at hand a wide range of possibilities to describe abstract specifications of queries. There are hardly any examples of object-oriented database languages with a sound theoretical basis that support sets, particularly predicatively

*This work has been funded in part by the Dutch 'Nationale Faciliteit Informatica' of the 'Nederlandse organisatie voor Wetenschappelijk Onderzoek' NWO under project NF–74 ISDF.

described sets, as part of the actual language. Two theoretically sound proposals should be noted in this respect, namely the OODB programming language *Machiavelli* [4] and the language IQL [5]. Both of these languages have full first-order facilities for expressing general *queries* in terms of predicatively described sets. Set expressions in the actual language, i.e. *objects* denoting sets, are in the case of both of these languages limited to sets basically built up by enumeration.
It is the topic of this paper to show how the Cardelli type system can be extended with set objects and logical formalism and still make full use of subtyping facilities. The type system that we describe in this paper (called LPT, from Lambda Power Type) offers, via power types and logical formalism, set constructs as enumeration, comprehension, union, intersection, and set difference.

The rest of this paper is organized as follows. Section 2 offers an introduction to [6] (concerning an alternative set-theoretic semantics for the semantics described in the original Cardelli paper [2]). Section 3 describes the extension of the Cardelli system with logical formalism and introduces the concept of "semantical equivalence". Section 4 offers a further extension with power types and sets, and shows how we can use semantical equivalence to give our typing rules concerning set constructs an intuitive and formal basis. Semantical equivalence provides a sound basis for object-oriented sets satisfying the important general principle that a set is uniquely determined by its elements. The paper ends with short conclusions and some directions for further research.

2 Cardelli theory and alternative semantics

We shall not discuss details of the original Cardelli type system in this paper. The reason for this is twofold. Firstly, we shall hardly deviate from the syntax and typing rules as described in the well-known paper [2], so readers are referred to this paper for details in that direction. Secondly, we wish to concentrate on new aspects of a Cardelli-like type system, namely the introduction of an alternative simpler semantics (as described in [6]), introduction of typed first-order logical symbolism, and introduction via power types of various set constructions. New aspects regarding logical symbolism and sets are also described in full detail in [7].

2.1 Subtyping and semantical issues

Subtyping is a feature of a typing discipline that may control the automatic insertion of implicit operations; it may also be used to model the inheritance relation in object-oriented languages [2]. Roughly said, we speak of subtyping when

- a partial order $\leq$ exists on types, and for types σ, τ with $\sigma \leq \tau$ there exists a ("conversion") operation $cv_{\sigma \leq \tau}$ that behaves like a function mapping arguments of type σ into results of type τ
- an expression e of type σ is allowed to occur at a position where something of type τ is required, provided that $\sigma \leq \tau$ and that the operation $cv_{\sigma \leq \tau}$ is applied (implicitly) to the value of e

Reynolds [8] gives an excellent overview of various possibilities of typing and subtyping.
Our description of typing and subtyping, mentioned above, is of a syntactical nature. It goes without saying that the question arises quickly whether types themselves have a meaning, i.e. whether there exists a (mathematical) semantics for types (and subtyping). Let us denote the semantics of closed expressions e and types τ by $[\![e]\!]$ and $[\![\tau]\!]$, respectively. It would be nice if the semantics $[\![\tau]\!]$ of type τ is merely a set such that $[\![e]\!] \in [\![\tau]\!]$ whenever e has type τ. However, only for simple (so-called first-order, non-recursive) types such a simple set-theoretic semantics seems possible. Most often one finds types interpreted as "domains" (continuous lattices or the like) and sometimes a set -theoretic interpretation is proved to be impossible [9]. The semantics of *sub*typing is our prime concern in this section.
We set out to construct, by *simple set-theoretic* means, a semantics for types —in the presence of subtyping— such that

$$[\![\sigma]\!] \subseteq [\![\tau]\!] \quad \text{whenever} \quad \sigma \leq \tau$$

This poses serious semantical problems. Consider for example the following situation:

- Assume that $[\![(\sigma \rightarrow \tau)]\!]$ = some non-empty set of functions that have domain $[\![\sigma]\!]$ and co-domain $[\![\tau]\!]$
- Assume that $[\![int]\!] \subset [\![real]\!]$
- Assume that $int \leq real$. Then, according to subtyping of function types (cf. [2]), we have $(real \rightarrow \tau) \leq (int \rightarrow \tau)$.

We then find that the desire $[\![(real \rightarrow \tau)]\!] \subseteq [\![(int \rightarrow \tau)]\!]$ contradicts the following two observations:

- Functions $f \in [\![(real \rightarrow \tau)]\!]$ cannot belong to $[\![(int \rightarrow \tau)]\!]$ because the domain of f differs from $[\![int]\!]$
- The cardinality of $[\![(real \rightarrow \tau)]\!]$ is strictly larger than that of $[\![(int \rightarrow \tau)]\!]$

It turns out that not only functions give rise to semantical problems when subtyping is added, but also records (i.e. when record types are given an obvious straightforward semantics as (labeled) cartesian products).
Several authors have attacked this problem, and have solved it by non-simple (Scottery, categorical) domain constructions for $[\![\tau]\!]$ ([2,10,11]).
Our solution, on the contrary, is as simple as effective, and can be stated in a single line. Given a semantics $[\![\ ]\!]$ for the language without subtyping, we form a new semantics $\{\!\![\ ]\!\!\}$ when subtyping is added, by defining

$$\{\!\![\tau]\!\!\} = \bigcup_{\sigma \leq \tau} [\![\sigma]\!]$$

For now we have, when $\sigma \leq \tau$, that

$$\begin{array}{lll} \{\!\![\sigma]\!\!\} & = & \bigcup_{\rho \leq \sigma} [\![\rho]\!] \\ & \subseteq & \bigcup_{\rho \leq \tau} [\![\rho]\!] \quad (\textit{transitivity of } \leq,\ \sigma \leq \tau) \\ & = & \{\!\![\tau]\!\!\} \end{array}$$

Note that we have used only elementary set-theoretic constructions in the definition of $\{\!\![\ ]\!\!\}$ for types. However, this still leaves us with the problem of defining $\{\!\![\ ]\!\!\}$ for expressions in such a way that

- $\{\!\![e]\!\!\} \in \{\!\![\tau]\!\!\}$ whenever expression e has type τ, and
- $\{\!\![\ ]\!\!\}$ is in a natural way related to $[\![\ ]\!]$

The first part is not hard to achieve. The second part poses some technical problems: we would like to "define" $\{\!\![\ ]\!\!\}$ by certain equations – these equations, however, turn out to be ambiguous. We can only succeed in showing that the ambiguity is not harmful by defining a *minimal typing* (that is sound and complete with respect to the given typing), defining a semantics based on this minimal typing, and then proving that the desired equations hold for those semantics. Full proof details can be found in [6].

The method described above, viz. constructing the required semantics $\{\!\![\ ]\!\!\}$ from a given semantics $[\![\ ]\!]$, is demonstrated by means of a simple language containing representatives for quite a number of practical programming language constructs. One concept that we do not take into account is (general) recursion; as a consequence the given semantics $[\![\ ]\!]$ can be kept quite simple. (If one adds recursion, the semantics $[\![\tau]\!]$, for types τ, should be a complete partial order (c.p.o.) or an even more complex structure. But even then our technique of defining $\{\!\![\tau]\!\!\} = \bigcup_{\sigma \leq \tau} [\![\sigma]\!]$ works, even though the resulting $\{\!\![\tau]\!\!\}$ is not a c.p.o. - and note that there is also no need for it to be a c.p.o.).

We shall offer a short overview of semantics of expressions and types in our alternative system. Since we wish to concentrate on new aspects of a Cardelli-like type system, we have refrained from taking into account various other expressions (and types) occurring in [2], such as if-then-else expressions, variants and case expressions. These expressions are, however, taken into full account in [6].

2.2 An overview of the semantics

In this section we offer a short overview of an alternative semantics as described in [6]. Below, the set B denotes the set of basic types (such as *int*, *real*, *bool* and *string*) and the set T denotes the set of types. T consists of basic types, record types and function types. We let β vary over B, and we let σ and τ vary over T. The set E denotes the set of expressions and E_τ denotes the set of expressions of type τ. We let e vary over E.
Function types are denoted by $(\sigma \to \tau)$, and functions are written in λ-notation. Record types are denoted by $\langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle$, and records are denoted by $\langle a_1 = e_1, \ldots, a_m = e_m \rangle$. We recall that the order in which field components occur in records and record types is irrelevant.
We postulate for each $\tau \in T$ a (possibly empty) set C_τ (of *constants*), mutually disjoint. C_{bool}=$\{$`true`,`false`$\}$. We let c vary over the C_τ.
Furthermore, we postulate for each $\tau \in T$ a set X_τ (of *variables*), mutually disjoint, countably infinite and disjoint from the sets C_σ $(\sigma \in T)$. We let x vary over the X_τ. We remark that the postulation that *variables are typed* eliminates the need for introducing a type assignment (that assigns a type to a variable), and therefore simplifies the presentation slightly. We refer to [6] for various proof details.

Definition 1 *For each $\tau \in T$ a set $[\![\tau]\!]$ is defined by induction on the structure of τ as follows*

1. *$[\![\beta]\!]$ is postulated for every $\beta \in B$*
2. *$[\![(\sigma \to \tau)]\!] \;=\; [\![\sigma]\!] \to [\![\tau]\!] \;=$ the set of all total functions from $[\![\sigma]\!]$ to $[\![\tau]\!]$*
3. *$[\![\langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle]\!] \;=$ the set of total functions with domain $\{a_1, \ldots, a_m\}$ that map a_i into $[\![\tau_i]\!]$ for all $i \in m$. We shall denote such a function f by its "graph" $\{(a_1, d_1), \ldots, (a_m, d_m)\}$*

□
We let d vary over any $[\![\tau]\!]$. The construction $[\![\tau]\!]$ is called the *pre-semantics* of the type τ. The eventual definition of semantics of types is given in definition 7.

Definition 2 $U = \bigcup_{\tau \in T} [\![\tau]\!]$, *the universe in which the semantics of both types and expressions shall find their place, both with and without subtyping.*

□

Definition 3 *The relation $:$ on $E \times T$ ($e : \tau$ is pronounced as:* e is well-typed and has type τ*) is defined inductively as follows*

1. *$c : \tau$, whenever $c \in C_\tau$*
2. *$x : \tau$, whenever $x \in X_\tau$*
3. *$(\lambda x.e) : (\sigma \to \tau)$, whenever $x \in X_\sigma,\ e : \tau$*
4. *$e(e') : \tau$, whenever $e : (\sigma \to \tau),\ e' : \sigma$*
5. *$\langle a_1 = e_1, \ldots, a_m = e_m \rangle : \langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle$, whenever $e_i : \tau_i \quad (1 \leq i \leq m)$*
6. *$e.a : \tau$, whenever $e : \langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle,\ a = a_j,\ \tau = \tau_j$ for some $j \ \ (1 \leq j \leq m)$*

□

Theorem 4 *For any $e \in E,\ \tau \in T$, there is at most one way to derive $e : \tau$.*

□

Definition 5 *An* assignment A *is a family of functions* $A_\tau \in X_\tau \to [\![\tau]\!]$, $(\tau \in T)$. *For assignment* A, $\tau \in T$, $x \in X_\tau$, $d \in [\![\tau]\!]$ *we define the assignment* $A[x \mapsto d]$ *for all* $\sigma \in T$, $y \in X_\sigma$ *by*

$$\begin{array}{rcll} (A[x \mapsto d])_\sigma(y) & = & A_\sigma(y) & , \textit{if} \quad y \neq x \\ & = & d & , \textit{if} \quad y = x \end{array}$$

□

Definition 6 *Let* A *be an assignment. Functions* $[\![\ ]\!]^\tau_A \in E_\tau \to U$ *are defined by induction on the derivation of their typing as follows*

1. $[\![c]\!]^\tau_A$ *is postulated for every constant* $c \in C_\tau$
2. $[\![x]\!]^\tau_A \ = \ A_\tau(x)$, *whenever* $x \in X_\tau$
3. $[\![(\lambda x.e)]\!]^{\sigma\to\tau}_A \ = \ \lambda d \in [\![\sigma]\!].\ [\![e]\!]^\tau_{A[x\mapsto d]}$, *whenever* $x \in X_\sigma$, $e : \tau$.
   (On the right hand side we have used λ *as a notation on the meta-level for functions.)*
4. $[\![e(e')]\!]^\tau_A \ = \ f(d)$, *where* $f = [\![e]\!]^{\sigma\to\tau}_A$, $d = [\![e']\!]^\sigma_A$, *whenever* $e : (\sigma \to \tau)$, $e' : \sigma$
5. $[\![\langle a_1 = e_1, \ldots, a_m = e_m\rangle]\!]^{\langle a_1:\tau_1,\ldots,a_m:\tau_m\rangle}_A \ = \ \{(a_i, [\![e_i]\!]^{\tau_i}_A) \mid 1 \leq i \leq m\}$, *whenever* $e_i : \tau_i \ (1 \leq i \leq m)$
6. $[\![e.a]\!]^\tau_A \ = \ f(a)$, *where* $f = [\![e]\!]^{\langle a_1:\tau_1,\ldots,a_m:\tau_m\rangle}_A$, *whenever* $e : \langle a_1 : \tau_1, \ldots, a_m : \tau_m\rangle$, $a = a_j$, $\tau = \tau_j$ *for some* j, $1 \leq j \leq m$

□

Theorem 7 *For each* $\tau \in T$, $e \in E_\tau$ *and assignment* A : $[\![e]\!]^\tau_A \in [\![\tau]\!]$.

Adding subtyping

Subtyping has already been discussed in section 2.1; here we discuss the subtyping relation $(\leq)$ and related matters in more detail.

We postulate a relation $\leq_B$ on $B \times B$. Also, for each $\beta, \beta' \in B$ with $\beta \leq_B \beta'$, we postulate a function $cv_{\beta\leq\beta'}$ in $[\![\beta]\!] \to [\![\beta']\!]$. We assume that the following properties hold true

$(\leq_B)$ $\quad \leq_B$ is a partial order

(ID_B) $\quad cv_{\beta\leq\beta} \ = \ identity_\beta \ \in \ [\![\beta]\!] \to [\![\beta]\!]$

(TR_B) $\quad cv_{\beta'\leq\beta''} \circ cv_{\beta\leq\beta'} \ = \ cv_{\beta\leq\beta''}$, for $\beta \leq \beta' \leq \beta''$, where the operation $\circ$ denotes function composition.

As an example, whether $[\![int]\!] \subseteq [\![real]\!]$ actually holds or not, one may choose $int \leq real$, provided that $cv_{int\leq real}$ is defined as some function from $[\![int]\!]$ to $[\![real]\!]$ satisfying the requirements listed above.

Definition 8 *We define a relation* $\leq$ *on* $T \times T$ *and, simultaneously, for each pair* $\sigma, \tau \in T$ *with* $\sigma \leq \tau$, *a function* $cv_{\sigma\leq\tau} \in [\![\sigma]\!] \to [\![\tau]\!]$, *by induction as follows*

1. *if* $\beta \leq_B \beta'$ *then:*
   - $\beta \leq \beta'$
   - $cv_{\beta\leq\beta'}$ *has been postulated above*
2. *let* $\sigma = (\sigma_1 \to \sigma_2)$ *and* $\tau = (\tau_1 \to \tau_2)$; *if* $\tau_1 \leq \sigma_1$ *and* $\sigma_2 \leq \tau_2$ *then:*
   - $\sigma \leq \tau$

- $cv_{\sigma\leq\tau}(f) \;=\; cv_{\sigma_2\leq\tau_2} \circ f \circ cv_{\tau_1\leq\sigma_1}$ *for* $f \in [\![\sigma_1 \to \sigma_2]\!]$

3. let $\sigma = \langle a_1 : \sigma_1, \ldots, a_m : \sigma_m\rangle$ *and* $\tau = \langle a_{j_1} : \tau_{j_1}, \ldots, a_{j_n} : \tau_{j_n}\rangle$*; if* $j_1, \ldots, j_n$ *is a (not necessarily contiguous) sub-sequence of* $1, \ldots, m$ *and* $\sigma_{j_i} \leq \tau_{j_i}$ $(1 \leq i \leq n)$ *then:*

- $\sigma \leq \tau$
- $cv_{\sigma\leq\tau}(\{(a_i, d_i) \mid 1 \leq i \leq m\}) = \{(a_{j_i}, cv_{\sigma_{j_i}\leq\tau_{j_i}}(d_{j_i})) \mid 1 \leq i \leq n\}$

□

The relation $\leq$ and functions $cv_{\sigma\leq\tau}$ satisfy the following properties (see [6] for proof details)

($\leq$) $\leq$ is a partial order on $T \times T$

(ID) $cv_{\tau\leq\tau} \;=\; identity_\tau \;\in\; [\![\tau]\!] \to [\![\tau]\!]$

(TR) $cv_{\sigma\leq\tau} \circ cv_{\rho\leq\sigma} = cv_{\rho\leq\tau}$, for $\rho \leq \sigma$.

Remark It was our intention to show that subtyping should, somehow, imply set inclusion; however, for the semantics for types defined thus far this is not yet the case – i.e., $\sigma \leq \tau$ does not imply $[\![\sigma]\!] \subseteq [\![\tau]\!]$, for arbitrary types σ, τ. For example, take $\sigma = \langle a : int,\ b : real\rangle$ and $\tau = \langle a : int\rangle$ (the reader can easily verify that in this case $\sigma \leq \tau$ holds, but *not* $[\![\sigma]\!] \subseteq [\![\tau]\!]$). Also we can take (cf. Section 2.1) $\sigma = (real \to int)$ and $\tau = (int \to int)$ to yield a contradiction for the statement $\sigma \leq \tau \Rightarrow [\![\sigma]\!] \subseteq [\![\tau]\!]$. This motivates to define a new semantics, written $[\{\ \}]$.

Definition 9 *For* $\tau \in T$ *we define*

$$[\{\tau\}] = \bigcup_{\sigma\leq\tau} [\![\sigma]\!]$$

□

Theorem 10 *For* $\sigma, \tau \in T:\ \sigma \leq \tau \Rightarrow [\{\sigma\}] \subseteq [\{\tau\}]$.
Proof

$$\begin{array}{rcll} [\{\sigma\}] & = & \bigcup_{\rho\leq\sigma}[\![\rho]\!] & \\ & \subseteq & \bigcup_{\rho\leq\tau}[\![\rho]\!] & \textit{(by transitivity of} \leq \textit{and } \sigma \leq \tau\textit{)} \\ & = & [\{\tau\}]\ . & \end{array}$$

□

Typing is now enhanced with the following rule

$$e : \tau, \text{whenever } e : \sigma \text{ and } \sigma \leq \tau.$$

Definition 11 *Let* A *be an assignment. Functions* $[\{\ \}]^\tau_A \in E_\tau \to U$ *are defined by induction on the derivation of the argument's type*

1.– 6. *as for the functions* $[\![\ ]\!]^\tau_A \in E_\tau \to U$ *in Definition 4 (replacing* $[\![\ ]\!]$ *by* $[\{\ \}]$*)*

7. $[\{e\}]^\tau_A \;=\; cv_{\sigma\leq\tau}([\{e\}]^\sigma_A)$ *whenever* $e : \sigma$ *and* $\sigma \leq \tau$.

□

Remark For given e and τ there may exist several distinct derivations of $e : \tau$ and therefore we have to show that this *syntactic ambiguity does not lead to semantic ambiguity.* In principle, we cannot claim that Definition 9 defines functions $\{\![\ ]\!\}^{\tau}_{A}$ but only relations "$\{\![\ ]\!\}^{\tau}_{A} = \ldots$"; we are faced with the problem to prove directly that the relations are functions, i.e.

$$\{\![e]\!\}^{\tau}_{A} = d \ \wedge\ \{\![e]\!\}^{\tau}_{A} = d' \ \Rightarrow\ d = d'$$

Another problem is that functions $\{\![\lambda x.e]\!\}^{\sigma\to\tau}_{A}$ become -in principle- nondeterministic and, compared to Definition 4, the structure of the universe changes drastically. We are faced with some serious technical problems here, which have been resolved in [6]. The solution to this problem lies in determining a unique type, the so-called *minimal type*, for each correctly typed expression. Then, by showing that the semantics as defined in Definition 9 are the result of a certain function application on the minimal semantics (cf. Theorem 16 below), we have shown that Definition 9 indeed always defines functions.

Minimal typing

We shall now shortly discuss a language with subtyping, that is called a system with *minimal typing.* A minimal type of an expression can be derived in at most one way, ensuring that we can safely base a definition of a semantics $[\![\ ]\!]^{*}_{A}$ on the derivation of an expression's minimal type. In terms of $[\![\ ]\!]^{*}_{A}$ we can express the unique solution of the equations for $\{\![\ ]\!\}^{*}_{A}$.
Our definition of minimal typing (cf. [6]) is basically the same as the one given in [8].

Definition 12 *The relation* $::$ *on* $E \times T$ *(*$e :: \tau$ *is pronounced as* τ is the minimal type of e*) is defined inductively as follows*

1. $c :: \tau$*, whenever* $c \in C_\tau$
2. $x :: \tau$*, whenever* $x \in X_\tau$
3. $(\lambda x.e) :: (\sigma \to \tau)$*, whenever* $x \in X_\sigma,\ e :: \tau$
4. $e(e') :: \tau$*, whenever* $e :: (\sigma \to \tau),\ e' :: \sigma'$ *and* $\sigma' \leq \sigma$
5. $\langle a_1 = e_1, \ldots, a_m = e_m\rangle :: \langle a_1 : \tau_1, \ldots, a_m : \tau_m\rangle$*, whenever* $e_i :: \tau_i\ (1 \leq i \leq m)$
6. $e.a :: \tau$*, whenever* $e :: \langle a_1 : \tau_1, \ldots, a_m : \tau_m\rangle,\ a = a_j,\ \tau = \tau_j$ *for some* j*,* $1 \leq j \leq m$

□

We say that e is *minimally typable* if $e :: \tau$ for some $\tau \in T$. (We note that every typable expression has a minimal type.)

We have the following important lemma

Lemma 13 *For any* $e \in E$ *there is at most one* $\tau \in T$ *such that* $e :: \tau$ *and there is at most one derivation of* $e :: \tau$*.*
Proof *Easy by induction on the structure of* e*.*

□

Definition 14 *Let* A *be an assignment. A partial function* $[\![\ ]\!]^{*}_{A} \in E \hookrightarrow U$ *is defined, for minimally typable expressions, as follows by induction on the derivation of the minimal type of its argument*

1. $[\![c]\!]^{*}_{A} = [\![c]\!]$ *as postulated in 2.17, whenever* $c \in C_\tau$

2. $[\![x]\!]^*_A = A_\tau(x)$, *whenever* $x \in X_\tau$
3. $[\![\lambda x.e]\!]^*_A = \lambda d \in [\![\sigma]\!].[\![e]\!]^*_{A[x \mapsto d]}$, *whenever* $x \in X_\sigma,\ e :: \tau$
4. $[\![e(e')]\!]^*_A = f(cv_{\sigma' \leq \sigma}(d))$, *where* $f = [\![e]\!]^*_A,\ d = [\![e']\!]^*_A$, *whenever* $e :: (\sigma \to \tau),\ e' :: \sigma'$ *and* $\sigma' \leq \sigma$
5. $[\![\langle a_1 = e_1, \ldots, a_m = e_m \rangle]\!]^*_A \ = \ \{(a_i, [\![e_i]\!]^*_A) \mid 1 \leq i \leq m\ \}$, *whenever* $e_i :: \tau_i\ (1 \leq i \leq m)$
6. $[\![e.a]\!]^*_A \ = \ f(a)$, *where* $f = [\![e]\!]^*_A$, *whenever* $e :: \langle a_1 : \tau_1, \ldots, a_m : \tau_m \rangle,\ a = a_j$ *for some* $j,\ 1 \leq j \leq m$

□

Note that, due to clause 4 in Definition 12, minimal semantics $[\![e]\!]^*$ is different from pre-semantics $[\![e]\!]^\tau$ for expressions e, where $e :: \tau$.

Theorem 15 *For* $e \in E,\ \tau \in T$*:*

$$e :: \tau \ \Rightarrow \ [\![e]\!]^*_A \in [\![\tau]\!]$$

Proof *Easy induction on the derivation of* $e :: \tau$.

□

Corollary 16 *For* $e \in E,\ \tau \in T$*:*

$$e :: \tau \Rightarrow [\![e]\!]^*_A \in [\{\tau\}].$$

□

Thus we have succeeded in designing semantics $[\![\ ]\!]^*_A$ and $[\{\ \}]$ such that

- $\sigma \leq \tau \Rightarrow [\{\sigma\}] \subseteq [\{\tau\}]$, and
- $e :: \tau \Rightarrow [\![e]\!]^*_A \in [\{\tau\}]$

However, the well-formedness of Definition 9 has yet to be shown.

Theorem 17 *(cf. [6,8])*

1. *(soundness)* $\quad e :: \sigma \ \Rightarrow \ e : \sigma$
2. *(completeness)* $e : \tau \ \Rightarrow \ e :: \sigma$, *for some* $\sigma \in T$
3. *(minimality)* $\quad e : \tau,\ e :: \sigma \ \Rightarrow \ \sigma \leq \tau$

□

Theorem 18 *Definition 9 defines functions* $[\{\ \}]^\tau_A \in E_\tau \to [\{\tau\}]$ *given by*

$$[\{e\}]^\tau_A \ = \ cv_{\sigma \leq \tau}([\![e]\!]^*_A)$$

where σ *is the existent and unique type such that* $e :: \sigma$ *and* $\sigma \leq \tau$ *(by Theorem 15).*

□

(The reader is referred to [6] for proof details.)

Remark The results of this section can be summarized in a nutshell by the following diagram

$$\begin{tikzcd}
\llbracket e\rrbracket^* = [\{e\}]^{\sigma}_A \arrow[r, dash] \arrow[d, "cv_{\sigma\leq\tau}"] & \in \arrow[r, dash] & \llbracket\sigma\rrbracket \arrow[r, dash] & \subseteq \arrow[r, dash] & [\{\sigma\}] \arrow[d, dash, "\subseteq" description] \\
[\{e\}]^{\tau}_A \arrow[r, dash] & \in \arrow[r, dash] & \llbracket\tau\rrbracket \arrow[r, dash] & \subseteq \arrow[r, dash] & [\{\tau\}]
\end{tikzcd}$$

where $e :: \sigma \leq \tau$.

3 Adding Predicates and Logical Formalism

The actual extension of the language that we would like to achieve, is predicatively described sets, of which the general format is $\{\ x \mid \varphi(x)\ \}$ where $\varphi(x)$ is a boolean expression. It is the purpose of this section to describe what the expressions $\varphi(x)$ look like.
Let e, e' be boolean expressions and let $x \in X_\sigma$ be some variable. Then so are $\neg(e)$, $(e \Rightarrow e')$ and $\forall x \bullet (e)$ boolean expressions, with the obvious intended meanings. (We note that the other propositional connectives like $\wedge$, $\vee$ and $\Leftrightarrow$, as well as the existential quantifier $\exists$, can be defined in terms of the above mentioned constructs in the usual way.) Furthermore, we have boolean constructs of the form $(e = e')$ and $(e \lhd e')$. The boolean construct $(e = e')$ is used to test whether e, e' are *equal*, and the second construct is used to test whether e is a *specialization* of the e'.
As far as defining the *typing* of the various expressions introduced in this section, we will proceed as follows. First, we will define a typing ":" for these expressions, and then add these definitions as clauses to Definition 3. We then define a minimal typing "::", and add these definitions as clauses to Definition 12. We can then prove that Theorem 17 holds in the extended language (cf. [7]). We shall assume, for reasons of simplicity in presentation, the following rule in the sequel of this paper for minimal typing of boolean expressions: $e : bool \Rightarrow e :: bool$.
As far as defining the *semantics* of the various expressions introduced in this section, we will proceed as follows. First, we will define a pre-semantics $\llbracket\ \rrbracket$ for these expressions, and then add these definitions as clauses to Definition 6. We then define a minimal semantics $\llbracket\ \rrbracket^*$, and add these definitions as clauses to Definition 14. The actual semantics $[\{\ \}]$ is then correctly obtained in accordance with Definition 11 and Theorem 18 (cf. [7]).
We note that the extensions described in this section do not affect the property of static typecheckability that the original language described in section 2 already had ([2,6,7]).

3.1 Typing and semantics of $\neg(e)$, $(e \Rightarrow e')$ and $\forall x \bullet (e)$

The expressions $\neg(e), (e \Rightarrow e')$ and $\forall x \bullet (e)$ have a very simple typing rule: these expressions are correctly typed with type *bool*, whenever the component expressions e and e' are correctly typed as *bool*.
These expressions also have a simple Tarskian-style pre-semantics, as demonstrated below.

Let A be an assignment. Definition 4 is now enhanced with the following three clauses

1. $\llbracket\neg(e)\rrbracket^{bool}_A = \mathtt{true} \iff \llbracket e\rrbracket^{bool}_A = \mathtt{false}$, whenever $e : bool$
2. $\llbracket(e \Rightarrow e')\rrbracket^{bool}_A = \mathtt{true} \iff (\llbracket e\rrbracket^{bool}_A = \mathtt{true}$ implies $\llbracket e'\rrbracket^{bool}_A = \mathtt{true})$, whenever $e, e' : bool$
3. $\llbracket\forall x \bullet (e)\rrbracket^{bool}_A = \mathtt{true} \iff (\llbracket e\rrbracket^{bool}_{A[x\mapsto d]} = \mathtt{true}$, for all $d \in \llbracket\sigma\rrbracket)$, whenever $x \in X_\sigma,\ e : bool$

Minimal semantics for these expressions can be obtained by substituting :: for :, and $*$ for *bool*.

3.2 Typing and semantics of $(e = e')$ and $(e \lhd e')$

Suppose that we adopt the following rule for the typing and semantics of $(e = e')$

$$e : \tau,\ e' : \tau \implies (e = e') : \mathit{bool}, \text{ for some } \tau \in T$$

$$[\![e = e']\!]_A^{bool} = \texttt{true} \iff [\![e]\!]_A^{\tau} = [\![e']\!]_A^{\tau}, \text{ for some } \tau \in T \text{ such that } e, e' : \tau$$

Then we would have as a consequence that <u>all</u> *record* expressions are semantically equal, due to the fact that all record expressions have the empty record type as a type. This leads, of course, to absurd results. We want that only those expressions can be considered equal that have exactly the same typing possibilities. (Stepping from a type to a supertype could be considered, both syntactically and semantically, as a step in abstraction. Equality is thus defined on the lowest level of abstraction; i.e. on the level of minimal typing.) This leads to the following clauses for typing and accompanying semantics

$$(e = e') : \mathit{bool}, \text{ whenever } e : \sigma, \text{ for some type } \sigma, \textit{ and } e : \tau \Leftrightarrow e' : \tau, \text{ for all } \tau \in T$$

$$[\![e = e']\!]_A^{bool} = \texttt{true} \iff [\![e]\!]_A^{\tau} = [\![e']\!]_A^{\tau}, \text{ for all } \tau \in T \text{ such that } e, e' : \tau$$

This has as a consequence that *quantification* over types is introduced in some typing rules. These quantifications pose no serious problems (for example for static typechecking), though, because these statements involving quantifications over types can be replaced by suitable statements concerning minimal types of expressions, which we now explain. Let minimal typing of equality expressions be given by the following clause

$$(e = e') :: \mathit{bool}, \text{ whenever } e, e' :: \tau, \text{ for some } \tau \in T.$$

It follows from Theorem 17 that two expressions with the same minimal type satisfy the property that they have exactly the same typing possibilities (cf. [7], for a proof that Theorem 17 holds in this extended language). This property has as an immediate consequence that the statement

$$(e = e') : \mathit{bool}, \text{ whenever } e : \sigma, \text{ for some type } \sigma, \text{ and } e : \tau \Leftrightarrow e' : \tau, \text{ for all } \tau \in T$$

can be replaced by the statement

$$(e = e') :: \mathit{bool}, \text{ whenever } e, e' :: \tau, \text{ for some } \tau \in T.$$

thus circumventing any problems possibly arising from quantifications over types in the former statement.

Minimal semantics for equality is given by

$$[\![e = e']\!]_A^{*} = \texttt{true} \iff [\![e]\!]_A^{*} = [\![e']\!]_A^{*}, \text{ whenever } e, e' :: \tau, \text{ for some } \tau \in T.$$

We have an analogous situation when confronted with defining a semantics $(e \lhd e')$, concerning specialization. The typing and semantics is given by

$$(e \lhd e') : \mathit{bool}, \text{ whenever } e' : \sigma, \text{ for some type } \sigma, \text{ and } e' : \tau \Rightarrow e : \tau, \text{ for all } \tau \in T$$

$$[\![e \lhd e']\!]_A^{bool} = \texttt{true} \iff [\![e]\!]_A^{\tau} = [\![e']\!]_A^{\tau}, \text{ for all } \tau \in T \text{ such that } e' : \tau$$

Minimal typing and minimal semantics is given in the following two clauses

$$(e \lhd e') :: \mathit{bool}, \text{ whenever } e' :: \tau',\ e :: \tau \text{ and } \tau \leq \tau'$$

$$[\![e \lhd e']\!]^*_A = \mathtt{true} \iff cv_{\tau \leq \tau'}([\![e]\!]^*_A) = [\![e']\!]^*_A, \text{ whenever } e' :: \tau',\ e :: \tau \text{ and } \tau \leq \tau'$$

As an example, $\langle \mathit{age} = 3, \mathit{speed} = 100, \mathit{fuel} = \text{'gas'} \rangle$ (called *mycar* here) is a specialization of the expression $\langle \mathit{age} = 3, \mathit{speed} = 100 \rangle$ (which we will call *myvehicle*). That is to say, the expression $(\mathit{mycar} \lhd \mathit{myvehicle})$ is correctly typed (with type *bool*), and also evaluates in the semantics to `true`.

3.3 Semantical equivalence

We define the following (congruence) relation on the set of expressions E. Let $e, e' :: \tau$, then

$$\begin{aligned} e \equiv e' &\iff [\{ e = e' \}]^{bool}_A = \mathtt{true}, \text{ for all assignments } A \\ &\iff [\{ e \}]^{\sigma}_A = [\{ e' \}]^{\sigma}_A, \text{ for all assignments } A, \text{ and all } \sigma \in T \text{ such that } e : \sigma \\ &\iff [\{ e \}]^*_A = [\{ e' \}]^*_A, \text{ for all assignments } A \end{aligned}$$

If $e \equiv e'$ holds, we say that e and e' are *semantically equivalent*.
A simple straightforward proof yields that the semantical equivalence relation constitutes a congruence relation on the set of expressions E. In the following section we shall use semantical equivalence as a formal tool for determining correct typing rules for object-oriented set constructions.

4 Adding Powertypes and Sets

In this section we will offer our definition of the language LPT, an extension of the Cardelli language with powertypes and sets. In defining typing and semantics of the various expressions introduced in this section, we will deviate slightly from the procedure given in section 3. In stead of explicitly offering a typing ":" for new expressions, we will directly state clauses related to typing in terms of minimal types. We have proceeded in this manner in order to circumvent possible quantifications over types in the typing rules. (The reader will have no problem, though, to reconstruct these typing rules from the rules given for minimal types.) Proceeding in this manner, we define a minimal typing "::", and add these definitions as clauses to Definition 12. We can then prove that Theorem 17 holds in the extended language (cf. [7]). For the semantics, we will define a pre-semantics $[\![\ ]\!]$ for these expressions, and then add these definitions as clauses to Definition 6. The actual semantics $[\{\ \}]$ is then obtained, via minimal semantics $[\![\ ]\!]^*$, in accordance with Definitions 11 and 14, as well as Theorem 18 (cf. [7]). We note that the extensions described in this section, as in section 3, also do not affect the static typecheckability of the original language. In other words, the *full* language LPT, with logical formalism and general set constructs, is statically typecheckable.

4.1 Set membership and subtyping

If σ is a type then $\mathbb{P}\sigma$ denotes the powertype of σ. Intuitively, a powertype $\mathbb{P}\sigma$ denotes the collection of all sets of expressions e such that e satisfies $e : \sigma$. Note that the semantics of a powertype as well as elements thereof can be infinite, depending on the particular type. The powertype constructor resembles the construction of the powerset $\mathcal{P}(V)$ of a set V in ordinary set theory. An expression e in our language is called a *set* if it has a powertype as its type; i.e. $e : \mathbb{P}\sigma$, for some type σ. We stress here that a set in our theory is an *expression* and not a type; i.e. we add to the set of types special types called powertypes, and, in addition, we add to the set of expressions special expressions called sets. Powertypes obey the following rule regarding subtyping: if $\sigma \leq \tau$ then $\mathbb{P}\sigma \leq \mathbb{P}\tau$.

The pre-semantics of powertypes is given by

$$[\![\mathbb{P}\tau]\!] = \mathcal{P}([\![\tau]\!]) = \{S \mid S \subseteq [\![\tau]\!]\}$$

Semantics of powertypes follows the general definition given in Definition 9.

Set membership is denoted by $\in$ and indicates that an element occurs in some set. We have the following clause for minimal typing of set membership

$$(e \in e') :: \texttt{bool}, \text{ whenever } e' :: \mathbb{P}\sigma \text{ and } e :: \sigma, \text{ for some } \sigma \in T.$$

The pre-semantics of set membership is given by

$$[\![e \in e']\!]_A^{bool} = \texttt{true} \iff [\![e]\!]_A^{\sigma} \in [\![e']\!]_A^{\mathbb{P}\sigma}, \text{where } e' :: \mathbb{P}\sigma \text{ and } e :: \sigma$$

The minimal semantics of set membership is given by

$$[\![e \in e']\!]_A^{*} = \texttt{true} \iff [\![e]\!]_A^{*} \in [\![e']\!]_A^{*}, \text{whenever } e' :: \mathbb{P}\sigma \text{ and } e :: \sigma, \text{ for some } \sigma \in T.$$

We also have a *specialized* version of set membership, denoted by ε, indicating that an element occurs, modulo the $\triangleleft$-relation, in some set. We have the following clause for minimal typing of specialized set membership

$$(e \,\varepsilon\, e') :: \texttt{bool}, \text{ whenever } e' :: \mathbb{P}\sigma', e :: \sigma \text{ and } \sigma \leq \sigma'.$$

The minimal semantics of specialized set membership is given by

$$[\![e \,\varepsilon\, e']\!]_A^{*} = \texttt{true} \iff cv_{\sigma \leq \sigma'}([\![e]\!]_A^{*}) \in [\![e']\!]_A^{*}, \text{whenever } e' :: \mathbb{P}\sigma',\ e :: \sigma, \text{ and } \sigma \leq \sigma'$$

Intuitively, $e \,\varepsilon\, e'$ holds if for some e'' we have $e \triangleleft e''$ and $e'' \in e'$.

Consider, as an example, the constants `1.0` and `2.0` of type *real* and the integer constant 2, and postulate that *int* $\leq$ *real*. We then have that $\texttt{2.0} \in \{\texttt{1.0, 2.0}\}$ is correctly typed and evaluates to true, whereas $2 \in \{\texttt{1.0, 2.0}\}$ cannot even be evaluated for the simple reason that it is wrongly typed. It does hold, however, that $2 \,\varepsilon\, \{\texttt{1.0, 2.0}\}$, since $2 \triangleleft \texttt{2.0}$.

4.2 Enumerative and predicative sets

The first set construct that we have is *set enumeration.* In order to get a clean typing of enumerated set terms, all of the form $\{e_1, \ldots, e_m\}$, we will only allow expressions $e_1, \ldots, e_m$ that have exactly the same typing possibilities; i.e. the expressions $e_1, \ldots, e_m$ must have the same minimal type. The reason for this, at first sight, rather severe typing rule is explained by means of semantical equivalence. This explanation will be given after we have discussed predicatively described sets. A predicative set is of the form $\{x \mid e\}$, where e is some boolean expression possibly containing the variable x, where $x \in X_\sigma$ for some $\sigma \in T$. Intuitively, such a set denotes the collection of all those expressions of type σ satisfying the predicate e. For example, let $x \in X_{real}$, then $\{x \mid (x = \texttt{1.0}) \vee (x = \texttt{2.0})\}$ is the set corresponding to $\{\texttt{1.0, 2.0}\}$. We have the following clause for the minimal typing of predicative sets:

$$\{x \mid e\} :: \mathbb{P}\sigma, \text{ whenever } x \in X_\sigma,\ e :: bool$$

The pre-semantics of predicatively described sets is given by

$$[\![\{x \mid e\}]\!]_A^{\mathbb{P}\sigma} = \{d \in [\![\sigma]\!] \mid [\![e]\!]_{A[x \to d]}^{bool} = \texttt{true}\}, \text{ whenever } x \in X_\sigma,\ e :: bool$$

Minimal semantics of predicatively defined sets is given by

$$[\![\{x \mid e\}]\!]^*_A = \{d \in [\![\sigma]\!] \mid [\![e]\!]^*_{A[x\to d]} = \texttt{true}\}, \text{ whenever } x \in X_\sigma,\ e :: \mathit{bool}$$

We now return to the question of how to type enumerated sets. What we would like is that the following semantical equivalence holds

$$\{e_1, \ldots, e_m\} \equiv \{x \mid (x = e_1) \vee \ldots \vee (x = e_m)\}$$

Note that we have not said yet to which set X_σ the variable x belongs in the predicatively described set above, on the right-hand side of the equivalence. From section 3.2 we can conclude that each of the disjuncts in the set on the right-hand side above is only properly typed if each of the expressions $e_1, \ldots, e_m$ has exactly the same minimal type as the minimal type chosen for x! This brings us to the following clause for typing of enumerated sets

$$\{e_1, \ldots, e_m\} :: \mathbb{P}\sigma, \text{ whenever } e_k :: \sigma, \text{ for all } k\ (1 \leq k \leq m), \text{ for some } \sigma \in T.$$

The pre-semantics of enumerative sets is given by

$$[\![\{e_1, \ldots, e_m\}]\!]^{\mathbb{P}\sigma}_A = \{[\![e_i]\!]^\sigma_A \mid 1 \leq i \leq m\}$$

Minimal semantics of enumerative sets is given by

$$[\![\{e_1, \ldots, e_m\}]\!]^*_A = \{[\![e_i]\!]^*_A \mid 1 \leq i \leq m\}, \quad \text{whenever } e_k :: \sigma, \text{ for all } k\ (1 \leq k \leq m), \text{ for some } \sigma \in T.$$

4.3 Union, intersection and set difference

Given two sets e and e', we can also form the union, intersection and difference of these two sets, denoted by $(e \cup e')$, $(e \cap e')$, $(e - e')$. Again, the concept of semantical equivalence will give us the correct typing rules for these set constructs.
In general, we would like the following semantical equivalence to hold

$$(\{x \mid e\} \cup \{y \mid e'\}) \equiv \{z \mid \exists x\ (e \wedge (x = z))\ \vee\ \exists y\ (e' \wedge (y = z))\}$$

Again, we have refrained from saying what the minimal types of the variables x, y and z are in the three sets given above. However, analogous to the arguments pertaining to the typing of enumerated sets, we immediately see that the only way that the set on the right-hand side of the equivalence can be correctly typed, is when the minimal types of the variables x, y and z are exactly the same. This means that the only way in which the union on the left-hand side of the equivalence can be correctly typed, is when the both sets involved in the union have the same minimal type.
This brings us to the following clause for minimal typing of union, intersection, and difference of sets.

$$(e \cup e'),\ (e \cap e'),\ (e - e')\ ::\ \mathbb{P}\sigma, \text{ whenever } e, e' :: \mathbb{P}\sigma.$$

Pre-semantics for union, intersection, and difference of sets can now be described in a straightforward manner.

Assume that $e, e' :: \mathbb{P}\sigma$.

1. $[\![(e \cup e')]\!]^{\mathbb{P}\sigma}_A \;=\; [\![e]\!]^{\mathbb{P}\sigma}_A \bigcup [\![e']\!]^{\mathbb{P}\sigma}_A$
2. $[\![(e \cap e')]\!]^{\mathbb{P}\sigma}_A \;=\; [\![e]\!]^{\mathbb{P}\sigma}_A \bigcap [\![e']\!]^{\mathbb{P}\sigma}_A$
3. $[\![(e - e')]\!]^{\mathbb{P}\sigma}_A \;=\; [\![e]\!]^{\mathbb{P}\sigma}_A - [\![e']\!]^{\mathbb{P}\sigma}_A$

Minimal semantics for union, intersection, and difference of sets is defined by replacing the superscript $\mathbb{P}\sigma$ in the three above listed clauses by $*$.

We mention, finally, that we also have a subset relation between sets, denoted by $(e \subseteq e')$, with the following clause for minimal typing

$$(e \subseteq e') :: \mathit{bool}, \text{ whenever } e, e' :: \mathbb{P}\sigma \text{ (for some type } \sigma).$$

The definition of the pre-semantics for the subset relation is straightforward and is given below.

Assume that $e, e' :: \mathbb{P}\sigma$. Then

$$[\![(e \subseteq e')]\!]_A^{\mathbb{P}\sigma} = \mathtt{true} \iff [\![e]\!]_A^{\mathbb{P}\sigma} \subseteq [\![e']\!]_A^{\mathbb{P}\sigma}.$$

Minimal semantics for the subset relation is defined by replacing the superscript $\mathbb{P}\sigma$ by $*$.

We also note that we can define a specialized form $\sqsubseteq$ of the subset relation $\subseteq$, analogous to $\in$ and ε: $e \sqsubseteq e'$ holds if, for some e'', it holds that $e \lhd e''$ and $e'' \subseteq e'$.

5 Conclusions and Directions for Further Research

We have extended the Cardelli type system with logical formalism and set constructs. This extended system, called LPT, is statically typecheckable and is provided with a set-theoretic semantics. The LPT system offers general set constructs as enumeration, comprehension, union, intersection, and set difference. A concept of "semantical equivalence" is introduced, and it is shown how semantical equivalence can be used to give typing rules concerning set constructs an intuitive and formal basis. As far as further research is concerned, we wish to incorporate polymorphism into the language. As LPT stands now, it can be used as a mathematical foundation for an object-oriented data model employing sets and constraints, thus as a basis for a database *specification* language. In order to use LPT as a foundation for a database *programming* language, such as *Machiavelli* [4], there very naturally arises the need to introduce polymorphism in the language. To this end, we are studying possible extensions of LPT with bounded quantification over types. We are also studying an extension of LPT with so-called *schemes*, both on the level of expressions and types, involving type parameters [12]. Both of these extensions are used to develop a theory, based on static typecheckability and a set-theoretic semantics, of inheritable methods.

Acknowledgements We would like to thank Maarten Fokkinga, Paris Kanellakis, and Val Breazu-Tannen for their comments improving an earlier version of this paper.

6 References

[1] Luca Cardelli, "Types for data-oriented languages," in *Advances in Database Technology—EDBT '88*, J. W. Schmidt, S. Ceri & M. Missikoff, eds., Springer-Verlag, New York–Heidelberg–Berlin, 1988, 1–15, Lecture Notes in Computer Science 303.

[2] Luca Cardelli, "A semantics of multiple inheritance," in *Semantics of Data Types*, G. Kahn, D. B. MacQueen & G. Plotkin, eds., Lecture Notes in Computer Science #173, Springer-Verlag, New York–Heidelberg–Berlin, 1984, 51–67.

[3] Francois Bancilhon, "Object-Oriented Database Systems," GIP Altaïr–INRIA, Rapport Technique *Altaïr* 16-88, Le Chesnay, FR, 1988, *Invited lecture Seventh ACM SIGART-SIGMOD-SIGACT Symposium on Principles of Database Systems, Austin, TX, March 1988.*

[4] A. Ohori, P. Buneman & V. Breazu-Tannen, "Database programming in Machiavelli—a polymorphic language with static type inference," in *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data, Portland, OR, May 31–June 2, 1989*, James Clifford, Bruce Lindsay & David Maier, eds., ACM Press, New York, NY, 1989, 46–57, (also appeared as ACM SIGMOD Record, 18, 2, June, 1989).

[5] S. Abiteboul & P.C. Kanellakis, "Object identity as a query language primitive," in *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data, Portland, OR, May 31–June 2, 1989*, James Clifford, Bruce Lindsay & David Maier, eds., ACM Press, New York, NY, 1989, 159–173, (also appeared as ACM SIGMOD Record, 18, 2, June, 1989).

[6] Herman Balsters & Maarten M. Fokkinga, "Subtyping can have a Simple Semantics," *Proceedings Computing Science in the Netherlands, Utrecht, The Netherlands, Nov. 3–4, 1988* (1988), *Revised version to appear in:* Theoretical Computer Science 87, 10, October 1991.

[7] Chris C. de Vreeze, "Extending the Semantics of Subtyping, accommodating Database Maintenance Operations," Universiteit Twente, Enschede, The Netherlands, Aug., 1989, Doctoraal verslag.

[8] John C. Reynolds, "Three Approaches to Type Structure," in *Mathematical Foundations of Software Development*, H. Ehrig et al., ed., Lecture Notes in Computer Science #185, Springer-Verlag, New York–Heidelberg–Berlin, 1985, 97–138.

[9] John C. Reynolds, "Polymorphism is not set-theoretic," in *Semantics of Data Types*, G. Kahn, D. B. Macqueen & G. Plotkin, eds., Lecture Notes in Computer Science #173, Springer-Verlag, New York–Heidelberg–Berlin, 1984, 145–156.

[10] D. B. MacQueen, R. Seti & G. D. Plotkin, "An ideal model for recursive polymorphic types," in *Proceedings Eleventh Annual ACM Symposium on Principles of Programming Languages*, 1984, 165–175.

[11] Kim Bruce & Peter Wegner, "An algebraic model of subtype and inheritance," in *Proceedings of the Workshop on Database Programming Languages*, 1987, 107–132.

[12] Chris C. de Vreeze & Rolf A. de By, "A formal approach to the inheritance of methods," 1991, submitted for publication.