JFP **17** (3): 353–386, 2007. © 2007 Cambridge University Press
doi:10.1017/S0956796806006174 First published online 4 January 2007 Printed in the United Kingdom

An object-oriented calculus with term constraints

GÁBOR M. SURÁNYI
Budapest University of Technology and Economics, H–1117 Budapest, Hungary
(*e-mail:* gabor.suranyi@db.bme.hu)

Abstract

Safety has become a fundamental requirement in all aspects of computer systems. Object-oriented calculi, such as Castagna's λ&-calculus and its variants (Castagna, 1997) ensure type safety in environments based on the distinguished object-oriented paradigm. Although for safety reasons object invariance and operation specifications are getting widely employed in all stages of the engineering process, they are not supported by these calculi. In this paper, a new calculus is presented which supports term (value) constraints besides the key object-oriented mechanisms (class types, inheritance, overloading with multiple dispatch and late binding). We also show how a type with constraints may realise a role, another useful object-oriented modelling element. The soundness of the type system and the confluence of the notion of reduction of the calculus are considered. The contribution also discusses computability issues partially arising from the use of first-order logic to formalise the constraints.

1 Introduction

1.1 Object-Oriented Constraint Specifications

Safety-aware design and implementation are no longer the privilege of mission-critical computer software. Methodologies ensuring the development of correct programs and methods proving program correctness are generally employed, especially when computer networks are involved.

Although object-orientation had turned out to be powerful to capture real-world scenarios, it was realised that the paradigm needs to be extended to deal with constraints describing object invariants and/or pre- and postconditions of message processing (OCL, 2003). This feature is in turn a great aid in pursuing the avoidance of design flaws and the elimination of implementation errors. Earlier, users in general used not to be particularly aware of hidden attributes of software including defects, but nowadays secureness, whose existence and absence are both inherently hidden properties, is a prerequisite to customer satisfaction. Therefore, computer engineers ought to use any technique which helps to reach this objective.

1.2 State-Based Roles

Roles (sometimes also referred to as dynamic or virtual classes) are nowadays widely-used modelling elements of object-oriented analysis and design (UML, 2003;

Li, 2004). They are especially fundamental in systems with long lifespan, such as databases (Rundensteiner, 1992; Papazoglou & Krämer, 1997).

The fact that an object instance plays a role is often expressed by the existence of a ‘dynamic object’ or a ‘role object’, which is created and destroyed as needed (Gottlob *et al.*, 1996; Li, 2004). This approach, however, has a drawback: whether an object takes on a role or not is often not independent of the object state (field values and links to other objects). As a consequence, when a new role is introduced, all non-observer operations have to be revised. Ultimately, this forbids great dynamism in the number of roles.

Most works dealing with roles and types mention that these notions are close to each other. In fact, the notion of type used in functional programming can cover roles as well (Ghelli, 2002). If a role type is state based, i.e. it describes constraints on the value domain, the typing relation can substitute role objects. This is beneficial as role membership can automatically be maintained by the system and new role types do not affect existing methods.

1.3 Overview of the area of functional calculi

The fundamental purpose of type systems is to prevent the occurrence of errors during the execution of programs (Cardelli, 2004) merely by analysing their code. Hence, these systems can act as a tool to prove partial correctness also w.r.t. security requirements and state-based roles. Formal type systems are designed for various forms of the λ-calculus which nowadays incorporate constructs to handle the object-oriented phenomena such as inheritance, late binding, polymorphism etc. These issues were initially tackled by introducing the subtyping relation and record types (Wand, 1987; Cardelli, 1988). Later, higher-order calculi such as $F_{\leq}$ (Curien & Ghelli, 1992) were invented to describe type-preserving functions and bounded polymorphism. A foundation for objects with explicitly designated roles is given in Ghelli (2002). Consequently, object-oriented calculi provide all apparatus for type checking w.r.t. the essential object-oriented features.

Unfortunately, there is currently no type system which is capable of ensuring error-free operation w.r.t. additional value constraints set forth by object invariance, operation and arbitrary state-based role specifications. In the current paper, we thus propose a new calculus which gives a typed foundation to the basic features of object-oriented programming (i.e. classes, inheritance, overloading with multiple dispatch[1] and late binding) along with value constraints. We have chosen $\lambda\&$-calculus (Castagna, 1997) as the basis of our work, because variants of $\lambda\&$ incorporate all vital object-oriented features including type-preserving functions, bounded polymorphism as well as multiple dispatch, and similar techniques may be applied to our results to gain a full-fledged object-oriented calculus. We call our calculus $\lambda\&\mathfrak{C}$, where $\mathfrak{C}$ stands for constraints.

[1] Multiple dispatch means that method selection is based on taking into account types of all arguments, not only the type of the receiver of the message.

Besides type systems, there are several other approaches to proving correctness of algorithms. Most of them are direct descendants of the method proposed by Hoare in his famous article (Hoare, 1969), and enriched with object-oriented features (Poetzsch-Heffter & Müller, 1998; de Boer, 1999). For this reason they are able to express constraints on objects and operations. However, a significant disadvantage of these techniques is that they do not really distinguish between classes and types (i.e. a collection of required properties) since they concentrate on concrete object-oriented languages with fixed type interpretations. This limits the applicability of the results to certain object-oriented systems. On the contrary, $\lambda\&\mathfrak{C}$ is adaptable to any object-oriented modelling environment including ones with higher-order features. We plan to base the type and role system of our deductive object-oriented database (Kardkovács & Surányi, 2004) on $\lambda\&\mathfrak{C}$.

The rest of the paper is organised as follows. The next section introduces the notational conventions of the contribution and the syntactical elements of the calculus including types, terms, the subtyping relation and the type system. Section 3 deals with various properties of the calculus. Practical aspects of the calculus such as computability, expressive power and specification-independent semantics are considered in Section 4. Although small examples are scattered around the paper to illustrate the new concepts, in a separate section a real-life scenario shows our calculus in action. Section 6 gives outlook and concludes the paper. The proofs of the theorems are presented in Appendix A. Appendix B shows how the notion of role presented in Ghelli (2002) translates into $\lambda\&\mathfrak{C}$.

2 Syntax of the $\lambda\&\mathfrak{C}$-calculus

To begin with, we formally define the calculus. We make use of the following notation throughout the paper:

- Γ is a type environment,
- A, B denote atomic types,
- R, S, T denote sorts (see later),
- U, V, W, X, Y denote (pre-)types,
- Θ, Φ, Ψ denote formula sets,
- M, N denote terms,
- I, J denote sets of indices and
- i, j, k, l, n are indices,
- x, y, z are variables.

These letters may also occur with indices. Moreover, we have the following symbols:

- $\Vdash$ signifies type entailment,
- $\vdash$ is the logical derivability symbol,
- $=$ ($\neq$) is the symbol of (in)equality in our logical language as well as in the meta-language, and
- $\equiv$ ($\not\equiv$) denotes syntactic (in)equality.

In this paper, $\wedge$ is frequently used as a unary symbol which creates a single formula from the elements of the formula set by joining them with the ordinary $\wedge$ connective. However, the symbols $\neg$, $\vee$ and $\Rightarrow$ retain their traditional use: they are respectively the logical negation and the binary logical connectives of disjunction and implication. $\forall$ and $\exists$ are, as usual, quantifiers, but the variables to quantify may be omitted if they apply to all open variables of the formula. At last, $\subseteq$ and $\emptyset$ are the common symbols of the subset relation and the empty set.

2.1 Sorts and pre-types

$\lambda\&$-calculus distinguishes pre-types and types: overloaded function pre-types (and any further pre-type based on them) which do not fulfil special consistency criteria are not considered as types. We also make this distinction (see Section 2.3) to allow the consistent application of reduction rules (that will be introduced in Section 3.1). However, according to our goal we need to introduce constraints. That is **PreTypes** of our calculus are:

$$V ::= {}_xA/\Theta \mid ({}_xV \rightarrow {}_yV)/\Theta \mid \{({}_{x_1}V \rightarrow {}_{y_1}V)/\Theta_1, \ldots, ({}_{x_n}V \rightarrow {}_{y_n}V)/\Theta_n\}/\Theta,$$

where Θ is a set of first-order well-formed formulae, the constraints. The last construction is the notation for types of overloaded functions with n branches. A pre-type without its outermost constraint set is called a *sort*. (This corresponds to the left side of the outermost / symbol in the pre-type.) Note that in this terminology, atomic types are actually not types, only sorts. We have retained their original terminology, however, to be coherent with other calculi.

Well-formed formulae of the constraint sets are built from atomic formulae with the standard logical connectives and quantifiers. Each free variable of a constraint formula must appear as a lower-left index of an atomic type or a function pre-type in the sort to which the formula belongs. As suggested and will be introduced later in the type system, each of these variables refers to the part of the pre-type expression which it marks. As a consequence, these index variables must all be different within each sort. To ease reading, we will omit the lower-left indices if they are not referenced or the constraint set is designated only by a symbol.

The actual set of predicate and function symbols can be freely chosen and are usually determined by the application domain, i.e. by the pre-types. (But for technical reasons, function symbols except constants may be disallowed, see Section 4.1.4.) A few predicate symbols, namely for each atomic type a unary symbol must be defined, however. Their semantics are that the parameter is of that type (i.e. an element of the domain of that type) and they serve the purpose of separating the theories of the atomic types, as explained later.

Two special operations are interpreted on constraint sets:

1. $\dot{\Theta}$ is the set of free variables in Θ.
2. $\hat{\Theta}$ is a set of open, atomic formulae of the form

$$type(x),$$

$$\frac{\langle A, B\rangle \in \mathscr{R}}{A/\Theta \leq B/\Theta} \qquad \text{VM [taut]}$$

$$\frac{U \leq V \quad V \leq W}{U \leq W} \qquad \text{[trans]}$$

$$\frac{\vdash \forall \wedge (\hat{\Theta} \cup \Theta) \Rightarrow \Phi}{S/\Theta \leq S/\Phi} \qquad \text{VM } [\vdash]$$

$$\frac{U_2 \leq U_1 \quad V_1 \leq V_2}{(U_1 \rightarrow V_1)/\Theta \leq (U_2 \rightarrow V_2)/\Theta} \qquad \text{VM } [\rightarrow]$$

$$\frac{I \subseteq J \quad \forall i \in I \ (U_i \rightarrow V_i)/\Phi_i \leq (X_i \rightarrow Y_i)/\Theta_i}{\{(U_j \rightarrow V_j)/\Phi_j\}_{j \in J}/\Theta \leq \{(X_i \rightarrow Y_i)/\Theta_i\}_{i \in I}/\Theta} \qquad \text{VM } [\{\}]$$

Fig. 1. Subtyping rules.

where $x \in \dot{\Theta}$ and *type* is the unary predicate symbol for the atomic type indexed by x in S/Θ.

Example 1
Assuming *int* is the atomic type of integer numbers, ${}_x int/\{x>0\}$ is the type of positive integers with the usual *greater than* relation. Provided that $\Theta = \{x>0\}$, the following equalities hold: $\dot{\Theta} = \{x\}$ and $\hat{\Theta} = \{int(x)\}$.

Example 2
Now let us consider a (curried) function which adds an integer to a *set*. This function then may have the type

$$({}_x set/\emptyset \rightarrow (int/\emptyset \rightarrow {}_z set/\emptyset)/\emptyset)/\{\text{size}(z) > \text{size}(x) \vee \text{size}(z) = \text{size}(x)\},$$

where *size* is a symbol of a function returning the size of a collection. Alternatively,

$$({}_x set/\emptyset \rightarrow (int/\emptyset \rightarrow {}_z set/\emptyset)/\emptyset)/\{\forall x_l \forall z_l \ (\text{size}(x, x_l) \wedge \text{size}(z, z_l)) \Rightarrow (z_l > x_l \vee z_l = x_l)\},$$

or

$$({}_x set/\emptyset \rightarrow (int/\emptyset \rightarrow {}_z set/\emptyset)/\emptyset)/\{\exists x_l \exists z_l \ (\text{size}(x, x_l) \wedge \text{size}(z, z_l)) \wedge (z_l > x_l \vee z_l = x_l)\},$$

may type the function, if *size* is a symbol of a binary predicate which is true if and only if the size of the first argument equals to the second argument.

2.2 *Subtyping*

Assuming there exists a partial order $\mathscr{R}$ on atomic types (i.e. $\langle A, B\rangle \in \mathscr{R}$ means the atomic type A is a subtype of the atomic type B), the subtyping relation ($\leq$) is defined for pre-types by the rules depicted in Fig. 1.

There are three points to comment on regarding these rules. First, in all cases marked with *VM* a special condition must hold between the pre-types compared in the consequent. *VM* is a symmetric binary relation; its formal definition is given below.

Definition 1 (*Variable Match of pre-types*)

- $\mathrm{VM}({}_{x}A/\Theta,\ {}_{x}B/\Phi)$ is always true,
- $\mathrm{VM}((U_1{\rightarrow}V_1)/\Theta,(U_2{\rightarrow}V_2)/\Phi)$ holds if $\mathrm{VM}(U_1,U_2)$ and $\mathrm{VM}(V_1,V_2)$,
- $\mathrm{VM}(\{(U_i{\rightarrow}V_i)/\Phi_i\}_{i\in I}/\Phi,\{(X_i{\rightarrow}Y_i)/\Theta_i\}_{i\in I}/\Theta)$ holds if for all $i{\in}I$

$$\mathrm{VM}((U_i{\rightarrow}V_i)/\Phi_i,(X_i{\rightarrow}Y_i)/\Theta_i).$$

A standard technique of logic called substitution of free variables permits replacing variables with other variables in pre-types as long as the replacing variable does not previously exist in the formula set. This means that the notation *VM* rather imposes syntactic conditions on its arguments and therefore appears as a side-condition.

Second, rule [$\vdash$] is responsible for reflexivity, and expresses the trivial expectation that a (pre-)type enforcing 'stricter' rules is a subtype of another. As first-order logic is sound and complete, everything deducible is true and conversely. To be able to deduce true formulae, however, proper axioms which formalise the properties of predicates and functions used for formula construction are needed. Axioms as well as the formulae to prove make use of the unary predicate symbols to specify the domains from which the variables take their values. In rule [$\vdash$] the formula set with a hat specifies the domains of variables.

Third, the rest of the rules are straightforward extensions of the traditional ones but rule [$\{\}$] is noteworthy. Actually, the similar rule of $\lambda\&$ is defined as

$$\frac{\forall i{\in}I\ \exists j{\in}J\ (U_j{\rightarrow}V_j){\le}(X_i{\rightarrow}Y_i)}{\{U_j{\rightarrow}V_j\}_{j\in J}{\le}\{X_i{\rightarrow}Y_i\}_{i\in I},} \qquad [\{\}_{\lambda\&}]$$

i.e. an overloaded function (pre-)type is subtype of another overloaded function (pre-)type if there is an element in the former which is subtype of the latter. In contrast to this, if we disregard the constraint sets, rule [$\{\}$] requires that the (pre-)types of each branch of the overloaded function (pre-)type on the right side have a separate branch with subtype on the left. It is, however, not a significant restriction in our opinion as only rather sophisticated and rare scenarios involving manipulation of arguments of an overloaded function (pre-)type exploit the flexibility offered by the more general rule. We opted for this less general rule as in this way definitions of *VM* and subtyping can be decoupled, which results in clearer presentation. There is no theoretical hurdle which would prevent combining the two definitions.

Example 3
One intuitively expects that the pre-types ${}_{x}int/\{x{>}0\}$ and ${}_{y}int/\{y{>}0\}$ are equivalent (i.e. subtypes of each other). In fact, this is easily provable with the rule [$\vdash$]. Clearly, similar equivalence holds between any two pre-types which differ only in their variables.

Example 4
Based on the subtyping rules, if all integers are real numbers, i.e. $\langle int, real\rangle \in \mathscr{R}$, it is provable that positive integers are positive real numbers, i.e.

$${}_{x}int/\{x{>}0\}{\le}\ {}_{y}real/\{y{>}0\}$$

holds.

Since $\langle int, real\rangle \in \mathscr{R}$ holds, with rule [taut] one gains

$$_x int/\{x>0\} \leq {}_x real/\{x>0\}.$$

The latter relationship is equivalent to

$$_x int/\{x>0\} \leq {}_y real/\{y>0\}$$

as only x is substituted with y on the right side.

2.3 Types

As mentioned before, consistency criteria are enforced on types. We do not impose any new requirement compared to $\lambda\&$-calculus, i.e. types in $\lambda\&\mathfrak{C}$ are defined by induction on the shape of the pre-type as follows.

Definition 2 (*Types*)

1. $A/\Theta \in$ **Types**,
2. if $V_1, V_2 \in$ **Types** then $(V_1 \rightarrow V_2)/\Theta \in$ **Types**,
3. $\{(U_i \rightarrow V_i)/\Theta_i\}_{i\in I}/\Theta \in$ **Types** if for all $i, j \in I$
   (a) $(U_i \rightarrow V_i)/\Theta_i \in$ **Types** and
   (b) $U_i \leq U_j \Rightarrow V_i \leq V_j$ and
   (c) for every U maximal element in the set of common lower bounds of U_i and U_j, there exists a unique $h \in I$ such that U_h is contained by the equivalence class of U.

Point 3b is the formalisation of the object-oriented notion called function (here: branch) specialisation. Point 3c ensures that an unambiguous most specific branch to select always exists.

Example 5
It may be helpful to illustrate the necessity of the last condition. Let

$$oldsmobile/\emptyset \leq car/\emptyset,$$

$$oldsmobile/\emptyset \leq oldtimer/\emptyset$$

and

$$\begin{array}{rcl} \{car/\emptyset & \rightarrow & {}_{y_1}real/\{y_1 \geqslant 0\}, \\ oldtimer/\emptyset & \rightarrow & {}_{y_2}real/\{y_2 \geqslant 0\}\}/\Theta \end{array}$$

be the pre-type of an overloaded function to calculate the tax payable by the owners of various types of vehicles each year (where $\geqslant$ denotes the usual *greater than or equal to* relation). Then it is not clear which branch applies to an $oldsmobile/\emptyset$. Formally, the above pre-type is not accepted as a type because it does not meet the last condition of types.

$$\Gamma \Vdash x : \Gamma(x) \qquad [\text{taut}_x]$$

$$\Gamma \Vdash c :\ {}_xT/\{x{=}c\} \qquad [\text{taut}_c]$$

$$\Gamma \Vdash \varepsilon : \{\}/\emptyset \qquad [\text{taut}_\varepsilon]$$

$$\frac{\Gamma \Vdash M : U{\leq}V}{\Gamma \Vdash M : V} \qquad [\text{subsumption}]$$

$$\frac{\Gamma, (x : U) \Vdash M : V}{\Gamma \Vdash \lambda x^U.M : (U{\to}V)/\emptyset} \qquad [\to \text{intro}]$$

$$\frac{\Gamma \Vdash M : (U{\to}V)/\Theta \quad \Gamma \Vdash N : U}{\Gamma \Vdash M{\cdot}N : V} \qquad [\to \text{elim}]$$

$$\frac{\Gamma \Vdash M : \{(U_i{\to}V_i)/\Theta_i\}_{i\in I}/\Theta \quad \Gamma \Vdash N : U_j}{\Gamma \Vdash M{\bullet}N : V_j} \qquad [\{\}\ \text{elim}]$$

$$\frac{\Gamma \Vdash M : W{\leq}\{U_i\}_{i\leqslant n-1}/\Theta \quad \Gamma \Vdash N : U_n}{\Gamma \Vdash (M\&^{\{U_i\}_{i\leqslant n}/\Theta\cup\Phi_n}N) : \{U_i\}_{i\leqslant n}/\Theta\cup\Phi_n} \quad \forall j \forall k\ \dot{\Phi}_j \cap \dot{\Phi}_k = \emptyset \quad U_l \equiv (V_l{\to}S_l/\Phi_l)/\Theta_l \qquad [\{\}\ \text{intro}]$$

Fig. 2. Type system rules.

2.4 Terms

$\lambda\&\mathfrak{C}$-calculus has exactly the same terms as $\lambda\&$. This means that we have the following **Terms**:

$$M \ ::= \quad x \mid c \mid \lambda x^V.M \mid M{\cdot}M \mid \varepsilon \mid M\&^V M \mid M{\bullet}M$$

where c denotes generic constants and $V \in$ **Types**. The first part of the terms is well-known from typed λ-calculus. The last three terms are the constant for the empty overloaded function, overloaded function construction and overloaded function application respectively. The symbol & in overloaded functions, as in $\lambda\&$, has to be annotated by a type. Just as in $\lambda\&$, it assures the soundness of the calculus since it is not affected by term reduction; see Section 3.1.

2.5 Type System

In Fig. 2 we precisely define the typing relation, which is denoted by the symbol $:$. The notation $\Gamma \Vdash M : V{\leq}U$ is used as a shorthand for the conjunction $\Gamma \Vdash M : V$ and $V{\leq}U$.

As the type system is a crucial part of the calculus and in consequence of improvements substantially differs from the one of $\lambda\&$, we pay special attention to its explanation. The first important property is that the type system does not specify unique typing (w.r.t. equivalence classes) for terms but rather employs rule [subsumption]. This fact also affects the rules dealing with overloaded functions; apart from the constraint sets they have simpler forms than in $\lambda\&$-calculus.

The rest of the changes are due to the introduction of constraints. Rule [taut$_c$] makes constants fully usable as terms by specifying that they are equal to the corresponding constants in the logical language.

There are modifications in the function introduction rules, too. Obviously, when typing a λ-abstraction the type (not only the sort) of the input variable has to be recorded in the type environment. The type of the abstraction is straightforward up to the constraint set, which is empty. The constraint set could describe the connection between the parameters and the result of the function (see Example 2). By the restriction, this feature is deliberately excluded from the current version of the calculus, as conditions which must hold when functions are invoked and the results are delivered (unrelated pre- and postconditions) can adequately be expressed in this way as well.

Typing an overloaded function works in a similar way. To outermost constraint set could keep track of all constraints among all elements of the overloaded function type, however, currently only the constraints on elements of the return types of the member function types are recorded there as this suffices to represent object invariants (see Section 4.3.1). Obviously, variables have to unambiguously identify parts of the overloaded function. This is ensured by the intersection condition on the dotted formula sets.

3 Semantics of the $\lambda\&\mathfrak{C}$-calculus

3.1 Operational Semantics

We define substitutions on the terms before giving the operational semantics of $\lambda\&\mathfrak{C}$ by the reduction relation.

Definition 3 (*Term substitution*)
The term $M[x{:=}N]$ is defined by induction as follows.

- $x[x{:=}N] = N$
- $y[x{:=}N] = y$ if $y \not\equiv x$
- $\varepsilon[x{:=}N] = \varepsilon$
- $(\lambda x.M')[x{:=}N] = \lambda x.M'$
- $(\lambda y.M')[x{:=}N] = \lambda y.(M'[x{:=}N])$ if y is not free in N
- $(P \&^U Q)[x{:=}N] = ((P[x{:=}N])\&^U(Q[x{:=}N]))$
- $(P \cdot Q)[x{:=}N] = (P[x{:=}N]) \cdot (Q[x{:=}N])$
- $(P \bullet Q)[x{:=}N] = (P[x{:=}N]) \bullet (Q[x{:=}N])$

Definition 4 (*Reduction*)
The one-step reduction relation ($\triangleright$) is defined by the following rules and can be applied to terms in any context (where contexts are defined in the standard way).

(Non-overloaded) function application

$$\lambda x^V.M \cdot N \triangleright M[x{:=}N] \qquad (\beta)$$

Overloaded function application If N is closed and irreducible, let

$$U \in \{U' | \Gamma \Vdash N : U' \ \wedge \ \forall V \Gamma \Vdash N : V \Rightarrow U' \leq V\}$$

and

$$U_j = \min_i \{U_i | U \leq U_i\}$$

then

$$(M_1 \&^{\{(U_i \to V_i)/\Theta_i\}_{i=1,\dots,n}/\Theta} M_2) \bullet N \triangleright \begin{cases} M_1 \bullet N, & \text{for } j<n \\ M_2 \cdot N, & \text{for } j=n \end{cases} \tag{$\beta_\&$}$$

The reduction relation ($\triangleright^*$) is the reflexive and transitive closure of $\triangleright$. Symbol $\triangleright_{\beta_\&}$ denotes a single reduction step performed via rule ($\beta_\&$).

The intuitive, operational meaning of rule ($\beta_\&$) is easily understood when looking at the simple case, i.e. when there are as many branches as arrows in the overloaded type. In this case, under the assumptions in the rule:

$$(\varepsilon \&^{\{(U_1 \to V_1)/\Theta_1\}/\Phi_1} M_1 \&^{\{(U_i \to V_i)/\Theta_i\}_{i=1,2}/\Phi_2} \dots \&^{\{(U_i \to V_i)/\Theta_i\}_{i=1,\dots,n}/\Theta} M_n) \bullet N \triangleright^*$$
$$\triangleright^* M_j \cdot N.$$

However, in general, the number of branches of the overloaded function may differ from the number of arrows in the overloaded type. This is because an overloaded function could begin with an application or with a variable, accounting for an initial segment of the overloaded type, and because of the subtyping relation used in rule [{} intro].

Clearly, the argument of an overloaded function must not be open or reducible since the system would not be confluent then (the type of an open or reducible argument can be different in different phases of the computation). In contrast to $\lambda\&$, the reason for the sophistication of rule ($\beta_\&$) is that the most specific branch (existence of which is guaranteed by type condition 3c, see Definition 2) is to be selected.

Example 6 (*cont.*)
After defining the reduction relation, we can demonstrate with the following scenario what consequences it would have to omit type annotations at the & symbols. To this end, let us assume that they would not be enforced. This implies that rule ($\beta_\&$) could only be based on the current deducible types of the overloaded function members.

In order to avoid clogging the formulae, now *car*, *oldtimer* and *oldsmobile* denote types not just sorts and *nnreal* is the type of non-negative real numbers. Let the symbols *ctax* and *ttax* denote functions that have been used to calculate the annual tax for simple *car*s and any *oldtimer* respectively. After a reform in taxation the term

$$\begin{array}{rl} \varepsilon & \& \\ ctax & \& \\ \lambda x^{(oldtimer \to nnreal)/\emptyset}.0 & \& \\ (\lambda x^{(oldsmobile \to nnreal)/\emptyset}.x) & \cdot ttax \end{array}$$

could be the overloaded function to calculate the tax for any vehicle. It expresses that now oldtimers are tax free. The term would be well-typed and have the type

$$\begin{array}{l}\{\quad (car{\rightarrow}nnreal)/\emptyset,\\ \quad (oldtimer{\rightarrow}nnreal)/\emptyset,\\ \quad (oldsmobile{\rightarrow}nnreal)/\emptyset \quad \}/\Theta.\end{array}$$

The actual outermost constraint set has no influence on the outcome of this reasoning, therefore it is only designated by a symbol throughout this example.

By β-reducing the last member of the overloaded function the above term would become

$$\varepsilon \;\&\; ctax \;\&\; \lambda x^{(oldtimer\rightarrow nnreal)/\emptyset}.0 \;\&\; ttax.$$

This would have the pre-type

$$\begin{array}{l}\{\quad (car{\rightarrow}real)/\emptyset,\\ \quad (oldtimer{\rightarrow}nnreal)/\emptyset,\\ \quad (oldtimer{\rightarrow}nnreal)/\emptyset \quad \}/\Theta'\end{array}$$

which is not a valid type because of condition 3c of Definition 2. Realising this, one would look for another pre-type which

- is in fact a type,
- is a subtype of the former pre-type so that rule [subsumption] entails for the term the same type as before the reduction,
- has 3 branches so that rule ($\beta_{\&}$) then results in the selection of the same non-overloaded function as before the β-reduction.

There is no such pre-type according to condition 3c of Definition 2 and rules [{}] and [→] in this case, however, as there is no greater type than *car* or *oldtimer*. That is, a well-typed overloaded function without type annotations at the & symbols could be reduced to an term which is ill typed.

Note that even a more liberal subtyping rule for overloaded function types, which is similar to $[\{\}_{\lambda\&}]$, would not eliminate the problem. Then an overloaded function type would have to be found which

- consists of three function types for rule ($\beta_{\&}$) for the same reason as above,
- contains function types which
  - — are subtypes of the branches of the above pre-type (according to the subtyping rule for overloaded function types),
  - — are supertypes of the types of the corresponding branches of the reduced term (so that non-overloaded function application would select them with appropriate arguments).

That is, either

$$\begin{array}{l}\{\quad (car{\rightarrow}real)/\emptyset,\\ \quad (oldtimer{\rightarrow}nnreal)/\emptyset,\\ \quad (oldsmobile{\rightarrow}nnreal)/\emptyset \quad \}/\Theta''\end{array}$$

or

$$\{\quad (car \to real)/\emptyset,\ (oldsmobile \to nnreal)/\emptyset,\ (oldtimer \to nnreal)/\emptyset \quad \}/\Theta'''$$

could type the reduced term, but if the overloaded function were applied to an oldtimer, for example, they would be reduced to different terms (0 and *ttax*, respectively). That is, the calculus would not be confluent.

3.2 Properties of the reduction relation

A calculus must possess both soundness and confluence properties to be useful.[2] In this section the relevant theorems of $\lambda\&\mathfrak{C}$-calculus are enumerated; the proofs can be found in Sections A.1 and A.2.

Theorem 1 (*Soundness aka subject reduction*)
If $\Gamma \Vdash M : V$ and $M \rhd^* N$ then $\Gamma \Vdash N : V$.

Theorem 2 (*Confluence of the notion of reduction*)
The notion of reduction of $\lambda\&\mathfrak{C}$-calculus, which consists of rules (β) and ($\beta_\&$), is confluent. That is, for all M, M_1, M_2 whenever $M \rhd^* M_1$ and $M \rhd^* M_2$, there exists M_3 such that $M_1 \rhd^* M_3$ and $M_2 \rhd^* M_3$.

4 Practical aspects of $\lambda\&\mathfrak{C}$-calculus

Now we have a new formalism and the question naturally arises how it can be used to design and build more robust object-oriented algorithms and systems. This section is devoted to answer this.

4.1 Computability issues

4.1.1 Concerning the subtyping relation

Subtyping plays a central role in the object-oriented paradigm. It is therefore important to be able to decide for each pair of types whether one is a subtype of the other. Unfortunately, the subtyping relation in its current form does not really support this as because of rules [trans], [$\vdash$] there can be multiple options how to proceed with the decision. As usual, it can be proved, however, that a proper equivalent subtyping relation exists.

[2] One may consider *strong normalisation* an important property, too. However, plain $\lambda\&\mathfrak{C}$ is not strongly normalising for the same reason as plain $\lambda\&$ (Castagna, 1997). Simple solutions based on a term ranking technique which can be applied to both calculi to eliminate this flaw are also presented in Castagna (1997).

Theorem 3
$U \leq V$ if and only if $U \preceq V$ where $\preceq$ is defined by the rules of $\leq$ but replacing [trans] with

$$\frac{S/\Theta \leq S/\Phi \quad S/\Phi \leq T/\Phi}{S/\Theta \leq T/\Phi} \qquad S \not\equiv T \text{ and } \Theta \neq \Phi \quad [\text{trans}']$$

and [$\vdash$] with

$$\frac{\vdash \forall \wedge (\hat{\Theta} \cup \Theta) \Rightarrow \Phi}{S/\Theta \leq S/\Phi} \qquad \Theta \neq \Phi \quad [\vdash']$$

Section A.3 contains the corresponding proof.

4.1.2 About pre-types and types

Definition 2 is based on a maximal element of the common lower bounds of two (already known) types, denoted by U_i and U_j in part 3c of the definition. The computability of such an element is not straightforward in the presence of constraints. However, provided that $U_i \equiv R_i/\Theta_i$, $U_j \equiv R_j/\Theta_j$ and $\text{VM}(U_i, U_j)$, after computing the sort part of the desired element, S, one has just to unify Θ_i and Θ_j, and find *the* element whose type is equivalent to $S/\Theta_i \cup \Theta_j$.

Types are not required to be consistent concerning their constraint set, i.e. the set may be unsatisfiable. This means, there can be types which cannot type any term. This is impractical as such types indicate modelling error and consume system resources without any advantage. So do functions which take terms of inconsistent types as input. Lastly, condition 3c of Definition 2 may need an inconsistent branch (a function which is never invoked because no argument can satisfy the constraints specified for the input of the function) in an overloaded function pre-type.

Example 7
The pre-type

$$\begin{aligned} \{ \quad & ({}_{x_1}real/\{x_1>0\} \rightarrow real/\emptyset)/\emptyset, \\ & ({}_{x_2}real/\{0>x_2\} \rightarrow real/\emptyset)/\emptyset \quad \}/\emptyset \end{aligned}$$

is not considered as a type. The reason is that because of condition 3c of Definition 2, there must be a branch which accept values of type ${}_{x}real/\{x>0\} \cup \{0>x\}$ but no branch typed

$$({}_{x_3}real/\{x_3>0, 0>x_3\} \rightarrow V)/\Theta$$

is present in the overloaded function pre-type. Note that according to the usual interpretation of relation $>$, no real number can be greater than 0 and less than 0 at the same time, i.e. the required branch is inconsistent.

To avoid the aforementioned situations, consistency may be enforced on all types, and condition 3c in the definition of types can be weakened so that it does not require an inconsistent branch. The modified definition is presented next.

Definition 5 (*Types with consistency*)
1. if $\exists \wedge (\hat{\Theta} \cup \Theta)$ then $A/\Theta \in$ **Types**,

$$\Gamma \Vdash_* x : \Gamma(x) \qquad [\mathrm{taut}_x]$$

$$\Gamma \Vdash_* c : {}_xT/\{x{=}c\} \qquad [\mathrm{taut}_c]$$

$$\Gamma \Vdash_* \varepsilon : \{\}/\emptyset \qquad [\mathrm{taut}_\varepsilon]$$

$$\frac{\Gamma \Vdash_* M : W {\leq} \{U_i\}_{i \leqslant n-1}/\Theta \quad \Gamma \Vdash_* N : W' {\leq} U_n}{\Gamma \Vdash_* (M \&^{\{U_i\}_{i\leqslant n}/\Theta\cup\Phi_n} N) : \{U_i\}_{i \leqslant n}/\Theta\cup\Phi_n}$$
$$\forall j \forall k\ \dot{\Phi}_j \cap \dot{\Phi}_k = \emptyset \quad U_l \equiv (V_l {\to} S_l/\Phi_l)/\Theta_l \qquad [\{\}\ \mathrm{intro}*]$$

$$\frac{\Gamma, (x : S/\Theta) \Vdash_* M : V}{\Gamma \Vdash_* \lambda x^{S/\Theta}.M : (S/\Theta {\to} V)/\emptyset} \qquad [\to \mathrm{intro}]$$

$$\frac{\Gamma \Vdash_* M : (U {\to} V)/\Theta \quad \Gamma \Vdash_* N : W {\leq} U}{\Gamma \Vdash_* M{\cdot}N : V} \qquad [\to \mathrm{elim}*]$$

$$\frac{\Gamma \Vdash_* M : \{(U_i {\to} V_i)/\Theta_i\}_{i\in I}/\Theta \quad \Gamma \Vdash_* N : U}{\Gamma \Vdash_* M \bullet N : V_j} \qquad U_j = \min_{i\in I}\{U_i | U {\leq} U_i\} \qquad [\{\}\ \mathrm{elim}*]$$

Fig. 3. Rules of the type algorithm.

2. if $V_1 {\equiv} T/\Theta, V_2 \in$ **Types** and $\exists\wedge(\hat{\Theta} \cup \Theta)$ and $\exists\wedge(\hat{\Theta}' \cup \Theta)$ then $(V_1 {\to} V_2)/\Theta' \in$ **Types**,
3. $\{(U_i {\to} V_i)/\Theta_i\}_{i\in I}/\Theta \in$ **Types** if $\exists\wedge(\hat{\Theta} \cup \Theta)$ and for all $i, j \in I$
   (a) $(U_i {\to} V_i)/\Theta_i \in$ **Types** and
   (b) $U_i {\leq} U_j \Rightarrow V_i {\leq} V_j$ and
   (c) for every U maximal element in the set of common lower bounds of $U_i \equiv T_i/\Phi_i$ and $U_j \equiv T_j/\Phi_j$, where $\mathrm{VM}(U_i, U_j)$,
   $$\neg\exists\wedge(\hat{\Phi}_i \cup \Phi_i \cup \hat{\Phi}_j \cup \Phi_j)$$
   or there exists a unique $h \in I$ such that U_h is contained by the equivalence class of U.

Although the use of Definition 5 instead of Definition 2 may seem straightforward, it contributes to the computability issue discussed in Section 4.1.4.

4.1.3 Type algorithm

Our type system does not directly specify a type (checking) algorithm as besides the other rules, rule [subsumption] can always be applied. To overcome this, a slightly modified version of the type system can be used. It is depicted in Fig. 3, where the modified rules are marked with asterisks. The following theorem declares the relationship of the type system and the type algorithm. The proof is presented in Section A.4.

Theorem 4 (*Minimum typing*)
If $\Gamma \Vdash_* M : V$ then

$$V \in \{U | \Gamma \Vdash M : U \text{ and } \forall W\ \Gamma \Vdash M : W \Rightarrow U {\leq} W\}.$$

4.1.4 Decidable Formulae and expressive power

It is well-known that in general several problems are undecidable in first-order logic (Börger *et al.*, 1997). This fact affects our results as well since we rely on derivability, which is generally undecidable, too. Therefore we have to select a decidable subclass of the problem.

There are several ways to restrict the constraint language to gain a decidable system. For instance, linear arithmetic logic and Presburger arithmetic logic are known to be decidable. However, a general-purpose application such as a deductive object-oriented data model (Kardkovács & Surányi, 2004) is not restricted to arithmetic constraints and may need further function and predicate symbols. The book (Börger *et al.*, 1997) presents a classification of first-order formulae based on quantifier prefixes and the cardinality of predicate and function symbols. It also exhaustively enumerates the maximal decidable and minimal undecidable cases w.r.t. this classification.

Since the domain to model can be arbitrary, the sum of the number of function and predicate symbols may not be limited, must be infinite. Furthermore, taking into consideration that rule [taut$_c$] requires equality, only three of the decidable classes may be suitable for us.

Bernays-Schönfinkel-Ramsey class: In the prefix form, existential quantifiers must precede universal ones and no function symbols except constants are allowed in the language.

Gurevich class: The prefix form of the formulae contains only existential quantifiers. Function and predicate symbols of any arity may occur.

Shelah class: The prefix form of the formulae contains a single universal quantifier and at most one unary function symbol. The number of existential quantifiers in the prefix and the number of predicate symbols are not limited.

Condition 3c of Definition 5 rules out the two latter classes in general because the prefix form of that formula may contain more than one universal quantifier.

The Bernays-Schönfinkel-Ramsey class limits the expressive power of our calculus in terms of constraints, however: in the prefix form of the constraints, no existential quantifier is allowed within universally quantified subformulae. In comparison with OMG's OCL (OCL, 2003) this is still less restrictive, as in OCL 'quantifiers' iterate over elements of collections only.

Moreover, if type consistency is important, no function symbols except constants may be used in $\lambda\&\mathfrak{C}$. This does not affect the expressive power of the calculus, since each n-ary function symbol can be represented by an $n{+}1$-ary predicate symbol with appropriate semantics provided that the extra parameter is a placeholder for the return value of the function (see Example 2).

If the original definition of types (Definition 2) is considered, i.e. type consistency is not enforced, formulae of the Gurevich and Shelah classes can be used in constraints, too. Then taking into consideration the aforementioned alternative representation of functions as predicates, at most a single universal quantifier may occur in the prefix form of the formulae, but the universal quantifier may be surrounded (i.e. preceded and succeeded) by any number of existential quantifiers. Conversely, if such formulae are needed, type consistency cannot be enforced in general.

4.2 Separating specifications from the semantics

Definition 4 requires that constraints be checked when evaluating a term by means of rule ($\beta_{\&}$). This means that program behaviour depends on the preconditions declared in the constraint set of a function. This may be a problem if constraints are regarded as specifications; however, it is a feature if constraints describe state-based roles. $\lambda\&\mathfrak{C}$-calculus as it is so far defined prefers the latter. Here we show how the calculus can be altered to fully decouple specifications from the semantics.

Definition 6 (*Types for specifications*)
In addition to the ones enumerated earlier (Definition 2 or 5), two extra conditions must be satisfied by overloaded function types:

3. $\{(S_i/\Phi_i \to V_i)/\Theta_i\}_{i\in I}/\Theta \in$ **Types** if for all $i, j \in I$
   (d) $i = j$ or $S_i \neq S_j$,
   (e) $S_i/\emptyset \leq S_j/\emptyset \Rightarrow S_i/\Phi_i \leq S_j/\Phi_j$.

Definition 7 (*Reduction for specifications*)
In the reduction and one-step reduction relations overloaded function application is replaced with the following.

If N is closed and irreducible, let

$$U \in \{U' \equiv R'/\emptyset \mid \Gamma \Vdash N : U' \ \wedge \ \forall V \Gamma \Vdash N : T/\emptyset \Rightarrow U' \leq T/\emptyset\}$$

and

$$U_j = \min_i \{U_i \equiv R_i/\Phi_i \mid U \leq R_i/\emptyset\}$$

then

$$(M_1 \&^{\{(U_i \to V_i)/\Theta_i\}_{i=1,\dots,n}/\Theta} M_2) \bullet N \triangleright' \begin{cases} M_1 \bullet N, & \text{for } j<n \\ M_2 \cdot N, & \text{for } j=n \end{cases} \qquad (\beta'_{\&})$$

The new one-step reduction relation and its reflexive and transitive closure are denoted by $\blacktriangleright$ and $\blacktriangleright^*$, respectively. Symbols $\blacktriangleright_{\beta}$ and $\blacktriangleright_{\beta'_{\&}}$ stand for the one-step reduction performed by rules (β) and ($\beta'_{\&}$), respectively.

When constraints are used only as specifications, firstly, branches of an overloaded function type expect arguments of different sorts. This is the key difference between uses of constraints for specifications and for roles. Secondly, branch selection in overloaded function application is based only on the sort of the argument and the expected sorts of the arguments of each branch. The actually selected branch is still unambiguous (even if only the sorts are considered) because in the overloaded function type sorts of the input arguments are all different in accordance with point 3d of Definition 6. Thirdly, so that sort-based branch selection suffices, condition 3e of the same definition ensures constraint set subsumption of the input arguments if the sorts are comparable.

Example 8

Let *treatment* denote the function which calculates the needed medicine to cure a particular illness. The dose usually varies with the state of the person, but the medicine recommended most likely depends upon whether the patient is pregnant.

On these assumptions *treatment* may have the type

$$\{(person/\emptyset \to medicine)/\emptyset, (\,{}_{x}person/\{pregnant(x)\} \to medicine)/\emptyset\}/\emptyset$$

where *person* is the sort of persons, *medicine* is the type for medicines and *pregnant* is a unary predicate symbol, the semantics of which is whether the person is pregnant. In this case, branches of *treatment* have their own specifications. Indeed, the second requires that the argument be a pregnant person. However, we cannot treat the whole overloaded function type as a specification. The prescribed medicine should depend on the person's pregnancy. In this case, rule $(\beta'_{\&})$ should not be able to be used for branch selection. It cannot be because the above pre-type is not a type in accordance with Definition 6. From the modelling point of view, here the constraint describes a role, the role of persons' pregnancy.

The situation is similar if *woman* is an additional sort for women, if and $\mathcal{R}(woman, person)$ holds, i.e. all women are persons, and *treatment* has the pre-type

$$\{(person/\emptyset \to medicine)/\emptyset, (\,{}_{x}woman/\{pregnant(x)\} \to medicine)/\emptyset\}/\emptyset.$$

Because of point 3e of Definition 6 this pre-type cannot be a type either. The rationale behind it is that if a non-pregnant woman is the argument, the wrong branch would be selected by rule $(\beta'_{\&})$.

Then the user can try to correct the model and use the constraint only as specifications. By replacing *person* with the sort *man* (of course *man* and *woman* are incomparable), they gain the type

$$\{(man/\emptyset \to medicine)/\emptyset, (\,{}_{x}woman/\{pregnant(x)\} \to medicine)/\emptyset\}/\emptyset.$$

Here sort-based branch selection, i.e. rule $(\beta'_{\&})$ works. This syntactic step delivers semantically incorrect result only if the argument is a non-pregnant woman. However, that case is tackled by the typing rules (which remained the same!) since such a term cannot be typed (rule [$\{\}$ elim] cannot be applied) before the $\beta'_{\&}$-reduction is executed.

The modified calculus too bears sound and confluence properties; the proofs of the following theorems are presented in Sections A.5 and A.6.

Theorem 5 (*Soundness for specifications*)

Defining types by Definition 6, if $\Gamma \Vdash M : V$ and $M \blacktriangleright^* N$ then $\Gamma \Vdash N : V$.

Theorem 6 (*Confluence of the notion of reduction for specifications*)

The notion of reduction of the modified $\lambda\&\mathfrak{C}$-calculus, which consists of rules (β) and $(\beta'_{\&})$, is confluent. That is, for all M, M_1, M_2 whenever $M \blacktriangleright^* M_1$ and $M \blacktriangleright^* M_2$, there exists M_3 such that $M_1 \blacktriangleright^* M_3$ and $M_2 \blacktriangleright^* M_3$.

$$\langle\!\langle \ell_1 : T_1/\Theta_1; \ldots ; \ell_n : T_n/\Theta_n ; \Theta \rangle\!\rangle = \{(L_1 \rightarrow T_1/\Theta_1)/\emptyset, \ldots, (L_n \rightarrow T_n/\Theta_n)/\emptyset\}/ \bigcup_{1 \leqslant i \leqslant n} \Theta_i \cup \Theta$$

only lower-left indices of atomic types in $T_1, \ldots, T_n$ may be the free variables in Θ

$$\langle \ell_1 = M_1; \ldots ; \ell_n = M_n \rangle = (\varepsilon \;\&\; \lambda x^{L_1}.M_1 \;\&\; \ldots \;\&\; \lambda x^{L_n}.M_n)$$

x does not occur or is not free in $M_1, \ldots M_n$

$$M.\ell_i = M \bullet \ell_i$$

Fig. 4. Encoding record types, records and field selection in λ&ʗ.

4.3 Object-Oriented Features

λ&ʗ is claimed to be an object-oriented calculus, however, there has been no mention of object constructs in the paper. Now we show how the rudimentary object-oriented notions are represented in λ&ʗ.

4.3.1 Objects and Classes

To begin with, let $\{K_i\}$ $(i \in I)$ be a set of atomic types and let us assume that its elements are isolated, i.e. no subtype relationship holds between each other and between them and other atomic types. Moreover, $L_i \equiv K_i/\emptyset$ and let us introduce for each L_i a constant $\ell_i : L_i$. It is now possible to encode record types, records and field selection, respectively, as depicted in Fig. 4.[3]

In words, a record is mapped to an overloaded function and labels (ℓ_i's) are used to select the branches (here: fields). So that these constructs do not interfere with ordinary overloaded functions, K_i's must be isolated. Labels are discarded after selection since x does not occur or is not free in M_i's. Record types also represent constraints among elements of the record via the constraint set Θ, which is mapped to the appropriate constraint set of the overloaded function type.

The state of an *object* (also called instance of a class) is stored in a record. A *class type* is an atomic type (A) that is associated with a unique representation type (V_A), which is a record type, the type of the object state. Whenever a class type, A is a *subtype* of another, B, its representation type, V_A must be a subtype of the representation type of the other, V_B. *Operations* correspond to overloaded functions of λ&ʗ. If the name of the operation is unique, the overloaded function has a sole branch. Otherwise all methods with the same name are collected in an overloaded function.

Although this way of object representation resembles the one of λ&, it also deals with object invariants. Their semantically correct use is achieved via the specially constructed typing rule [{} intro].

[3] Because they are all clear from the context, from this point on we deliberately omit the type annotations at the & symbols in order to keep the expressions simple.

4.3.2 Messages with constraints

Messages are, as in $\lambda\&$, the overloaded functions themselves with appropriate arguments. But here, unrelated pre- and post-conditions of message processing (function calls) can be enforced by means of constraints on the parameters and the return value. Indeed, if constraints on multiple parameters are present, the participating parameters have to be passed as a single record for this purpose. $\lambda\&\mathfrak{C}$, currently, does not support constraints on multiple input parameters of curried functions. This is a straight consequence of our definition of **PreTypes** (see Section 2.1), which requires that all open variables of a constraint set be indices in their corresponding sorts. A more sophisticated definition, however, raises problems concerning reasoning over constraints, soundness, etc. as demonstrated in the following example. We are going to address these issues in the future.

Example 9
Let us consider the curried function

$$f \equiv \lambda x^{int/\emptyset}.\lambda y^{\,{}_{y}int/\{0>y>x\}}.y$$

where the type of y uses as well the (open) variable x. Here the second parameter of f must be negative but greater than the first one. The proper type of f would be

$$({}_{x}int/\{0>x\}\rightarrow({}_{y}int/\{0>y\}\rightarrow int/\Theta)/\Theta')/\{0>y>x\}$$

where some of the constraint sets are denoted by symbols (Θ and Θ') only. Note that here x should implicitly get a constraint set derived from the constraints on y. The calculation of such a set is not a straightforward task.

However, verifying the constraints of the first argument of f is inevitable, otherwise e.g. $(f \cdot 2) \cdot 3$ could have a type but after a β-reduction, it would become

$$(\lambda y^{\,{}_{y}int/\{0>y>2\}}.y) \cdot 3,$$

which cannot be typed any longer as rule $[\rightarrow \text{elim}]$ can not be applied. That is, the calculus would not be sound. Moreover, it is in general non-trivial to calculate the new constraint set of y in f after the first β-reduction, since the argument (which is in this case part of the constraint as well!) could be an arbitrary expression.

4.3.3 Inheritance, overloading, overriding and late binding

The other object-oriented features of $\lambda\&$-calculus are left intact by our extension, i.e. they work in exactly the same way in $\lambda\&\mathfrak{C}$-calculus. These features include

- overloading with multiple dispatch, which is a core element of the $\lambda\&$-system,
- inheritance, which is given by subtyping and the branch selection rule for overloaded functions,
- overriding, which is ensured by condition 3b in the definition of overloaded function types,
- late binding, which is provided by reduction rule $(\beta_{\&})$ or $(\beta'_{\&})$.

This section showed that our extension of $\lambda\&$ with term constraints integrates smoothly into the framework set up by Castagna to model object-oriented languages

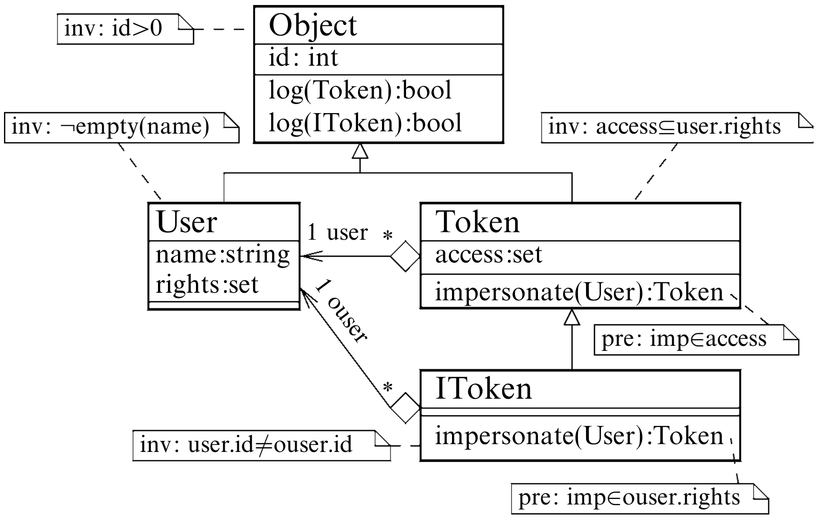

Fig. 5. Class diagram of an access control subsystem.

and systems. The reader interested in such higher-level constructs is referred to Castagna (1997).

5 An illustrative example

The application of calculi tends to be indirect, i.e. they are usually deeply hidden in compilers of programming languages, various checkers etc. It is therefore difficult to present a simple but practical example of how a calculus can be used. The solution we opted for is that we take a real-life scenario, model it in an object-oriented way but operate on low-level functional primitives, which can be derived from the model in the way introduced in Section 4.3.

Let us consider the object model of an access control subsystem. We conceive the access control as a two-step process: first users of the system authenticate themselves to gain certain access rights to any object and receive a token, then with appropriate tokens they are authorised to carry out actions on objects. One of the benefits of token usage is that it inherently supports impersonation.

Figure 5 depicts an excerpt of the class diagram of this scenario. The diagram uses the notations of OMG's UML (UML, 2003) and OCL (OCL, 2003) and presumes that the classifiers *bool*, *int*, *set* and *string* are predefined. To enable uniform object management, we define (à la Java) the class *Object* as the root of all classes. Although the name of the attributes, operations and association roles should be self-describing, we give a brief explanation of them:

access: rights the user obtained in a particular token;
id: object identifier;
impersonate: creates a new token on behalf of another user;
log: audits object access, returns true if successful;
name: user name;
ouser: user who impersonates another;

Table 1. *Representation types*

Class	**Representation Type**
Object	$\langle\!\langle id:\ {}_{i}int/\{i{>}0\} \quad ;\emptyset\rangle\!\rangle$
User	$\langle\!\langle id:\ {}_{i}int/\{i{>}0\};name:\ {}_{n}string/\{\neg empty(n)\};rights: set/\emptyset \quad ;\emptyset\rangle\!\rangle$
Token	$\langle\!\langle id:\ {}_{i}int/\{i{>}0\};access:\ {}_{a}set/\emptyset$ $;user:\langle\!\langle id:\ {}_{u}int/\{u{>}0\};name:\ {}_{n}string/\{\neg empty(n)\};rights: \quad {}_{r}set/\emptyset;\emptyset\rangle\!\rangle$ $;\{a{\subseteq}r\}\rangle\!\rangle$
IToken	$\langle\!\langle id:\ {}_{i}int/\{i{>}0\};access:\ {}_{a}set/\emptyset$ $;user:\langle\!\langle id:\ {}_{u}int/\{u{>}0\};name:\ {}_{n}string/\{\neg empty(n)\};rights: \quad {}_{r}set/\emptyset;\emptyset\rangle\!\rangle$ $;ouser:\langle\!\langle id:\ {}_{o}int/\{o{>}0\};name:\ {}_{m}string/\{\neg empty(m)\};rights: \quad set/\emptyset;\emptyset\rangle\!\rangle$ $;\{u{\neq}o \wedge a{\subseteq}r\}\rangle\!\rangle$

Table 2. *Representation types encoded in* $\lambda\&\complement$

Class	**Representation Type in** $\lambda\&\complement$
Object	$\{(id{\rightarrow}\ {}_{i}int/\{i{>}0\})/\emptyset\} \quad /\{i{>}0\}$
User	$\{(id{\rightarrow}\ {}_{i}int/\{i{>}0\})/\emptyset,(name{\rightarrow}\ {}_{n}string/\{\neg empty(n)\})/\emptyset$ $,(rights{\rightarrow}set/\emptyset)/\emptyset\} \quad /\{i{>}0,\neg empty(n)\}$
Token	$\{(id{\rightarrow}\ {}_{i}int/\{i{>}0\})/\emptyset,(access{\rightarrow}\ {}_{a}set/\emptyset)/\emptyset, \quad (user{\rightarrow}$ $\{(id{\rightarrow}\ {}_{u}int/\{u{>}0\})/\emptyset,(name{\rightarrow}\ {}_{n}string/\{\neg empty(n)\})/\emptyset,(rights{\rightarrow}\ {}_{r}set/\emptyset)/\emptyset\}$ $/\{u{>}0,\neg empty(n)\})/\emptyset\} \quad /\{i{>}0,u{>}0,\neg empty(n),a{\subseteq}r\}$
IToken	$\{(id{\rightarrow}\ {}_{i}int/\{i{>}0\})/\emptyset,(access{\rightarrow}\ {}_{a}set/\emptyset)/\emptyset, \quad (user{\rightarrow}$ $\{(id{\rightarrow}\ {}_{u}int/\{u{>}0\})/\emptyset,(name{\rightarrow}\ {}_{n}string/\{\neg empty(n)\})/\emptyset,(rights{\rightarrow}\ {}_{r}set/\emptyset)/\emptyset\}$ $/\{u{>}0,\neg empty(n)\})/\emptyset, \quad (ouser{\rightarrow}$ $\{(id{\rightarrow}\ {}_{o}int/\{o{>}0\})/\emptyset,(name{\rightarrow}\ {}_{m}string/\{\neg empty(m)\})/\emptyset,(rights{\rightarrow}set/\emptyset)/\emptyset\}$ $/\{o{>}0,\neg empty(m)\})/\emptyset\} \quad /\{i{>}0,u{>}0,o{>}0,u{\neq}o,\neg empty(n),\neg empty(m),a{\subseteq}r\}$

rights: access rights a user can have in tokens;
user: user whose name is logged whenever an object is accessed with the token.

According to Section 4.3.1, attributes of the classes are collected in record types, which are the representation types of the classes. The representation types used in the access control subsystem are enumerated in Table 1. They are mapped to the $\lambda\&\complement$-types listed in Table 2 by the encoding rule described in the same section. In the tables a handy notation is applied for the first time: the special terms ℓ_i and the types L_i are denoted by the same strings since ℓ_i and L_i may not occur at the same place in $\lambda\&\complement$-expressions. Furthermore, we sacrifice our notation for readability as new letters denote variables. We continue with this practice in this chapter. The extra spaces in the rows serve the same purpose.

It is easy to recognise the salient feature of the class representation in $\lambda\&\complement$, i.e. the specification and the enforcement of object invariants. The encoded invariants were earlier indicated in the class diagram: identifiers are positive, user names are not empty, a token's access rights are a subset of the user's access rights and self-impersonation is not allowed.

We now turn to the operations of the classes. The operation *log* is declared to be overloaded because when impersonation is in effect, the real user has to be recorded as well. Since recursive impersonation must preserve the original user's identity, *impersonate* demonstrates another object-oriented concept, overriding. In accordance with Section 4.3.1, both operations are modelled as overloaded functions in $\lambda\&\mathfrak{C}$, and late binding is realised by the reduction rule $(\beta_{\&})$ as it can be executed with an irreducible argument only. Then the types of operations *log* and *impersonate* straightforwardly follow from the class diagram and are respectively:

$$\{(\mathbf{Token} \to \mathit{bool}/\emptyset)/\emptyset, (\mathbf{IToken} \to \mathit{bool}/\emptyset)/\emptyset\}/\emptyset$$

and

$$\begin{array}{lcr}
\{ & (& \\
\{ & (\mathit{id}\!\to {}_{i_1}\mathit{int}/\{i_1{>}0\})/\emptyset, (\mathit{access}\!\to {}_{a}\mathit{set}/\underline{\{\mathit{imp}{\in}a\}})/\emptyset, (\mathit{user}\!\to\!\mathbf{User})/\emptyset & \} \\
/ & \{i_1{>}0, \ldots, \underline{\mathit{imp}{\in}a}\} & \\
\to & (\mathbf{User} \to \mathbf{Token})/\emptyset)/\emptyset & , \\
 & (& \\
\{ & (\mathit{id}\!\to {}_{i_2}\mathit{int}/\{i_2{>}0\})/\emptyset, (\mathit{access}\!\to\!\mathit{set}/\emptyset)/\emptyset, (\mathit{user}\!\to\!\mathbf{User})/\emptyset, & \\
 & (\mathit{ouser}\!\to & \\
\{ & (\mathit{id}\!\to {}_{o}\mathit{int}/\ldots)/\emptyset, (\mathit{name}\!\to {}_{m}\mathit{string}/\ldots)/\emptyset, (\mathit{rights}\!\to {}_{c}\mathit{set}/\underline{\{\mathit{imp}{\in}c\}})/\emptyset & \} \\
 & /\{o{>}0, \neg\mathit{empty}(m), \mathit{imp}{\in}c\})/\emptyset & \} \\
/ & \{i_2{>}0, o{>}0, \neg\mathit{empty}(m), \ldots, \underline{\mathit{imp}{\in}c}\} & \\
\to & (\mathbf{User} \to \mathbf{Token})/\emptyset)/\emptyset & \} \\
 & /\emptyset &
\end{array}$$

where bold class names stand for the representation types of the designated classes (with appropriately substituted variables) and trivial parts of the type expressions are replaced with ellipses. Note that we modelled *impersonate* as a curried function and that the operations' return type is not an *IToken* because whenever self-impersonation is attempted, a primary (i.e. not an impersonation) token is returned.

Most importantly, the underlined parts of the above type expression impose special preconditions, i.e. additional constraints on the input parameters besides the regular object invariants. In accordance with the class diagram, they require that the caller have impersonate rights in its token. Indeed, this goal can alternatively be reached by defining atomic types for each required set of rights and a partial order on them. However, our approach is scalable as new access rights can be introduced and used without further effort. As the types of operations *log* and *impersonate* fulfil the requirements of Definition 6, they are really specifications and Reduction for Specifications (Definition 7) is usable in this model.

Last but not least, let us consider a simple implementation of the operations *impersonation* to see the type system at work. Let the following function be given, and we want to verify that this can be an implementation of *impersonation*:

$$
\begin{array}{llcl}
\varepsilon\ \& & & \lambda x^{\{(id\rightarrow\ {}_{i_1}int/\{i_1>0\})/\emptyset,(access\rightarrow\ {}_{a}set/\{imp\in a\})/\emptyset,(user\rightarrow\mathbf{User})/\emptyset\}}. & (1)\\
 & & \lambda y^{\mathbf{User}}.\quad \mathrm{ite}\cdot(x\bullet user\bullet id =_f y\bullet id)\qquad\cdot & (2)\\
(& & \varepsilon\ \&\ \lambda z^{id}.\mathrm{unique}\ \&\ \lambda z^{access}.\emptyset\ \&\ \lambda z^{user}.y\qquad)\,\cdot & (3)\\
(& & \varepsilon\ \&\ \lambda z^{id}.\mathrm{unique}\ \&\ \lambda z^{access}.\emptyset\ \&\ \lambda z^{user}.y\ \&\ \lambda z^{ouser}.x\bullet user\quad) & (4)\\
\& & & impersonate\_for\_IToken & (5)
\end{array}
$$

Here the symbol $=_f$ denotes the binary function which tests if its arguments are the same integers. Let us assume it has a type of

$$(\{(left\rightarrow int/\emptyset)/\emptyset,(right\rightarrow int/\emptyset)/\emptyset\}/\emptyset\rightarrow bool)/\emptyset,$$

that is pairs are records with labels *left* and *right*. The term denoted by unique represents an expression which generates (unique) positive integers to be used as identifiers. It has a type of ${}_{i}int/\{i>0\}$. The symbol ite stands for if-then-else, a function which returns its second argument if the first argument is true and the third argument otherwise. In this case, let it have the type

$$(bool\rightarrow(\mathbf{Token}\rightarrow(\mathbf{Token}\rightarrow\mathbf{Token})/\emptyset)/\emptyset)/\emptyset.$$

The string *impersonate_for_IToken* is another expression to realise the impersonation from an *IToken*. As this branch is very similar to the other and if it were expanded, it would not demonstrate any new idea, it is abbreviated to this form.

We start the type derivation from the innermost terms, i.e. in lines 3 and 4. Let the type of x from line 1 be denoted by U. Moreover, via rule [$\vdash$] we obtain $U\leq\mathbf{Token}$. Line 3 is an overloaded function, its branches are easy to type. However, the branch $\lambda z^{access}.\emptyset$ is noteworthy. Here by rule [taut$_c$],

$$(x:U),(y:\mathbf{User}),(z:access)\Vdash\emptyset:\ {}_{a}set/\{a=\emptyset\}.$$

Then using rule [$\rightarrow$ intro]

$$(x:U),(y:\mathbf{User})\Vdash(\lambda z^{access}.\emptyset):(access\rightarrow\ {}_{a}set/\{a=\emptyset\})/\emptyset.$$

All this implies via rule [{} intro] that line 3 has the type
$\{(id\rightarrow\ {}_{i}int/\{i>0\})/\emptyset,(access\rightarrow\ {}_{a}set/\{a=\emptyset\})/\emptyset,\quad(user\rightarrow$
$\{(id\rightarrow\ {}_{u}int/\{u>0\})/\emptyset,(name\rightarrow\ {}_{n}string/\{\neg empty(n)\})/\emptyset,(rights\rightarrow\ {}_{r}set/\emptyset)/\emptyset\}$
$/\{u>0,\neg empty(n)\})/\emptyset\}\quad/\{i>0,u>0,\neg empty(n),a=\emptyset\}.$
It can in fact be typed as **Token** as well because the empty set is a subset of any set and because of subtyping rules [$\vdash$], [{}] and type system rule [subsumption].

Analogously, the term in line 4 is of type **Token**, too, although it has an additional branch $\lambda z^{ouser}.x\bullet user$. From rules [subsumption] and [{} elim] it follows that

$$(x:U),(y:\mathbf{User}),(z:ouser)\Vdash(x\bullet user):\mathbf{User}.$$

Lastly, rule [$\rightarrow$ intro] entails

$$(x:U),(y:\mathbf{User})\Vdash(\lambda z^{ouser}.x\bullet user):(ouser\rightarrow\mathbf{User})/\emptyset.$$

Now we turn to the terms $x\bullet user\bullet id$ and $y\bullet id$. Their type is ${}_{i}int/\{i>0\}$, its calculation involves no new method. Again rules [$\vdash$] and [subsumption] help so that in turn rule [$\rightarrow$ elim] can be applied to the call of $=_f$. The result is of *bool* then. Arguments of *bool* and **Token** (twice) are accepted for the call of ite and we obtain by using rule [$\rightarrow$ elim] twice consecutively that ite with these arguments has the type **Token**.

After all this the remaining steps are easy: we use rule [$\rightarrow$ intro] twice for the λ-abstractions and then rule [$\{\}$ intro] twice for the overloaded function. The final result is what we expected, i.e. the term may be an implementation of *impersonation*.

6 Conclusion and outlook

The goal of the paper was to reduce the gap emerging between latest object-oriented modelling techniques and (partial) correctness proving systems. Because of the ubiquitous demand for secure computing, one major tendency is the extensive use of constraints in specifications. We tackled the problem by extending the $\lambda\&$-calculus with term constraints and gained a calculus supporting the basic object-oriented features (class types, inheritance, overloading with multiple dispatch and late binding) along with the precise specification of object invariants and unrelated pre- and postconditions of messages. In addition, state-based role types, which are useful object-oriented modelling elements, too, can be mapped to types with term constraints as well. As $\lambda\&\mathfrak{C}$ inherently does not presuppose any particular primitive type, operation or relation, it can form the basis for a flexible specification language with higher-order features and decidable fragments.

Clearly, our system lacks several mechanisms known in object-oriented design and implementation such as recursive types, abstract classes, information hiding, upcast, imperative features, type constraints (type-preserving methods and bounded polymorphism). Some of these can be encoded into $\lambda\&\mathfrak{C}$-calculus without problems and some of them require further effort, e.g. inventing higher-order variants of $\lambda\&\mathfrak{C}$. These are subjects of our future research. In this process the techniques applied to $\lambda\&$ and described in Castagna (1997) can be very useful.

Some applications need a behaviour-based subtyping relation. We partially addressed this problem as $\lambda\&\mathfrak{C}$ can represent and deal with invariant properties only. Considering history and liveness properties based on temporal logic is another possible way of extension.

In imperative programming languages ensuring the proper use of resources is not a straightforward task. Object-oriented design makes it easier but still not trivial. Both the sort and the constraint part of our calculus can support this verification process by incorporating the ideas of Boudol (1997) and Abadi & Leino (1998).

The work most similar to ours is, on one hand, Hofmann et al's verification method (Hofmann *et al.*, 1996; Hofmann *et al.*, 1998). It treats objects as abstract data types and is able to prove if the functional implementation matches the specification part. Because of its perspective, however, object invariants cannot be specified in the system. Furthermore, as the theoretical background is quite involved and the way of description substantially differs from the ones of object-oriented models, Hofmann *et al.*'s method is difficult to apply in practice. On the other hand, Ghelli too gives a foundation for object roles Ghelli (2002). However, as already pointed out in the Introduction, he deals only with roles explicitly indicated in the objects. $\lambda\&\mathfrak{C}$ is more flexible as role membership can depend on any part of the object state, not only on a role tag. (See Appendix B on how Ghelli's roles can be translated into $\lambda\&\mathfrak{C}$.)

Nowadays only a small percent of computer systems does not involve distributed computation. While ad-hoc solutions are usually adequate to prove that a non-distributed algorithm also terminates implying total correctness, proving that a distributed object-oriented system possesses correct behaviour is a challenging task. This includes the proper treatment of shared mutable objects. Fortunately, Krishnaswami & Aldrich (2005) present a solution for standard type theory. Therefore it can later be merged into higher-order variants of $\lambda\&\mathfrak{C}$. Our future endeavour is to examine other interactions of this field (Kobayashi & Yonezawa, 1994; Boudol, 1997; Igarashi & Kobayashi, 2004) with the $\lambda\&\mathfrak{C}$-calculus.

Acknowledgements

I thank the anonymous referees for their helpful comments both on the contents and the presentation of the paper.

A Proofs

We prove six theorems. We proceed by following the techniques for $\lambda\&$-calculus (Castagna, 1997) whenever it is possible.

A.1 Soundness of $\lambda\&\mathfrak{C}$

The proof of soundness requires a lemma stating that by term substitution the type may only decrease.

Lemma 1 (*Substitution lemma*)
Let $\Gamma, (x : U) \Vdash M : V$, $\Gamma \Vdash N : U'$ and $U' \leq U$. Then $\Gamma \Vdash M[x{:=}N] : V'$, where $V' \leq V$.

Proof
The proposition is proven in the same way as in $\lambda\&$, by induction on the structure of M. The following cases differ from the ones in the original proof:

$M \equiv M_1 \cdot M_2$ Rule [subsumption] must be applied to $M_2[x{:=}N]$ before rule [$\rightarrow$ elim] to obtain the desired result.

$M \equiv (M_1 \&^V M_2)$ Rule [subsumption] must be applied to $M_2[x{:=}N]$ before rule [$\{\}$ intro], but $\Gamma \Vdash M[x{:=}N] : V$ holds.

$M \equiv M_1 \bullet M_2$ where

$$\Gamma, (x : U) \Vdash M_1 : \{(U_i \rightarrow V_i)/\Theta_i\}_{i \in I}/\Theta$$

With the chosen

$$U_h = \min_{i \in I}\{U_i | U \leq U_i\},$$

$$V_i \leq V \tag{A 1}$$

because of type condition 3b (see Definitions 2 and 5). The original method proves that

$$\Gamma \Vdash M[x{:=}N] : V'_k {\leq} V_i \tag{A 2}$$

with

$$V'_k = \min_{j \in J}\{U'_j | U' {\leq} U'_j\}$$

where

$$\Gamma \Vdash M_1[x{:=}N] : \{(U'_j {\rightarrow} V'_j)/\Theta'_j\}_{j \in J}/\Theta'$$

and by the induction hypothesis

$$\{(U'_j {\rightarrow} V'_j)/\Theta'_j\}_{j \in J}/\Theta' {\leq} \{(U_i {\rightarrow} V_i)/\Theta_i\}_{i \in I}/\Theta$$

with $I \subseteq J$. Equations (A 1) and (A 2) imply the proposition in this case. □

Theorem 7 (*Strict soundness*)
If $\Gamma \Vdash M : V$ and $M \rhd^* N$ then $\Gamma \Vdash N : V'$ where $V' {\leq} V$.

Proof
The proposition is inferred for $\rhd^*$ if the same holds for $\rhd$. The statement for $\rhd$ is proven as in $\lambda\&$, by induction on the structure of M. Two subcases differ from the ones in the original proof, both are part of the case $M \equiv M_1 \bullet M_2$.

$M_1 \rhd M'_1$ For starting point the induction hypothesis gives the same subtyping rule as branch $M \equiv M_1 \bullet M_2$ of Lemma 1 needs, therefore formally the same proof steps can be carried out.

$M \rhd_{\beta_\&} M'$ If the selected function from the overloaded function is the last one, rule [subsumption] is to be applied before rule [$\rightarrow$ elim], and the result is $V' \equiv V$. Otherwise rule [{} elim] with the selected branch delivers the same result. □

The soundness of $\lambda\&\complement$ simply follows from the preceding theorem via the rule [subsumption].

A.2 Confluence of $\lambda\&\complement$

Theorem 2 (*Confluence*)
For all M, M_1, M_2 whenever $M \rhd^* M_1$ and $M \rhd^* M_2$, there exists M_3 such that $M_1 \rhd^* M_3$ and $M_2 \rhd^* M_3$.

Proof
Since $\lambda\&\complement$ uses exactly the same notion of reduction ($\beta \cup \beta_\&$) over the same terms as $\lambda\&$, the confluence of $\lambda\&$ implies this theorem. □

A.3 Decidable subtyping

To prove that $\preceq$ is equivalent to $\leq$ is not difficult, but a bit tedious. To reduce work, we provide the proof of a few common situations in a generalised way in advance.

Lemma 2 (*Constraint set independence*)
Let Π be a proof for $S/\Theta{\leq}T/\Theta$ in $\preceq$. Then there is a proof in $\preceq$ for $S/\Phi{\leq}T/\Phi$ with arbitrary constraint set Φ.

Proof
Π does not end with rules [trans′], [$\vdash'$] since they require different constraint sets. That is, the last rule is one of [taut], [$\rightarrow$] and [$\{\}$]. Then the same rule can be applied with the constraint set Φ. □

Lemma 3 (*Constraint set derivation*)
Let Π be a proof for $S/\Theta{\leq}S/\Phi$ with $\Theta \neq \Phi$ in $\preceq$. Then it consists of rule [$\vdash'$] only.

Proof
The last step cannot be rule [trans′] as it requires different sorts. Similarly, it is not rule [taut], [$\rightarrow$] or [$\{\}$] because the constraint sets are not the same. □

Lemma 4 (*Normal form of proofs*)
Let Π be a proof for $S/\Theta{\leq}T/\Phi$ with $S \not\equiv T$ and $\Theta \neq \Phi$ in $\preceq$. Then rule [$\vdash'$] with constraint sets Θ and Φ can be applied on sort S. Furthermore, there is a proof in $\preceq$ for $S/\Psi{\leq}T/\Psi$ with any constraint set Ψ.

Proof
Since the sorts as well as the constraint sets differ in Π the last rule is [trans′]:

$$\frac{S/\Theta{\leq}S/\Phi \quad S/\Phi{\leq}T/\Phi}{S/\Theta{\leq}T/\Phi.}$$

According to Lemma 3, $S/\Theta{\leq}S/\Phi$ implies the first proposition. The second proposition directly follows from Lemma 2 by taking into consideration the subproof for $S/\Phi{\leq}T/\Phi$. □

Lemma 5 (*Constraint set in a subtype*)
If there are proofs for $S/\Theta{\leq}T/\Theta$ and $T/\Theta{\leq}T/\Phi$ in $\preceq$, $S/\Theta{\leq}S/\Phi$ is also provable in $\preceq$.

Proof
Because of Lemma 3, the proof of $T/\Theta{\leq}T/\Phi$ is a rule [$\vdash'$]. By the definition of the formula set with hat, the truth of $\hat{\Theta}$ for the sort S always implies the truth of $\hat{\Theta}$ for the sort T. That is, the same rule [$\vdash'$] holds with sort S, too. □

Theorem 3
$U{\leq}V$ if and only if $U \preceq V$.

Proof
The *if* part is trivial as at $\lambda\&$. For the *only if* part we first note that the application of [$\vdash$] is the application of either rule [$\vdash'$] or the rule

$$V{\leq}V \qquad \text{[refl]}$$

That is, in this regard it is to show that rule [refl] is superfluous. That a type is always subtype of itself is provable without any transitivity rule by induction on the type structure. For

atomic types it follows from the partial order property of $\mathscr{R}$ and rule [taut],
function types it follows from the induction hypothesis via rule [$\rightarrow$],
overloaded function types it is implied by the induction hypothesis and by rule [$\{\}$] with $I{=}J$.

We now prove that [trans′] suffices as a transitivity rule. We actually show that for all other forms of the transitivity rule there is a proof without them. To this end, let us assume that Π is a smallest proof in which another transitivity rule is needed. There are several cases based on which sorts are the same in this rule.

1.
$$\frac{S/\Theta{\le}S/\Phi \quad S/\Phi{\le}S/\Psi}{S/\Theta{\le}S/\Psi}$$
If $\Theta = \Phi$ or $\Phi = \Psi$, the rule can be omitted as one of the premises is the conclusion. Otherwise, because of Lemma 3 both subproofs consist of rule [$\vdash'$]. But rule [$\vdash'$] is transitive because so is logical implication used in its definition. That is, a rule [$\vdash'$] can replace this transitivity rule.

2.
$$\frac{S/\Theta{\le}T/\Theta \quad T/\Theta{\le}T/\Phi}{S/\Theta{\le}T/\Phi} \qquad S \neq T$$
Because of Lemma 2, there is a proof for $S/\Phi{\le}T/\Phi$ in $\le$. Moreover, Lemma 5 implies that there is a proof for $S/\Theta{\le}S/\Phi$. At last, rule [trans′] can connect the proofs for $S/\Phi{\le}T/\Phi$ and $S/\Theta{\le}S/\Phi$ to obtain the same subtyping within $\le$.

3.
$$\frac{S/\Theta{\le}T/\Phi \quad T/\Phi{\le}S/\Psi}{S/\Theta{\le}S/\Psi} \qquad S \neq T$$
There are several subcases.

$\Theta = \Psi$: The rule to replace is just a reflexivity rule.
$\Theta = \Phi$, $\Theta \neq \Psi$: The subproof on the left side is a proof for $S/\Theta{\le}T/\Theta$. From the subproof on the right side by Lemma 4 there is a proof for $T/\Theta{\le}T/\Psi$. Then Lemma 5 entails a proof for $S/\Theta{\le}S/\Psi$.
$\Phi = \Psi$, $\Theta \neq \Psi$: In a single step, Lemma 4 proves the existence of a proof for the conclusion based on the subproof on the left side.
$\Theta \neq \Phi \neq \Psi$, $\Theta \neq \Psi$: Lemma 4 entails that there are proofs for $S/\Theta{\le}S/\Phi$ and $S/\Phi{\le}T/\Phi$ (from the subproof on the left side) and for $T/\Phi{\le}T/\Psi$ (from the subproof on the right side). The last two imply by Lemma 5 that there is a proof for $S/\Phi{\le}S/\Psi$. This along with $S/\Theta{\le}S/\Phi$ reduces to case 1.

4.
$$\frac{R/\Theta{\le}S/\Phi \quad S/\Phi{\le}T/\Psi}{R/\Theta{\le}T/\Psi} \qquad R \neq S \quad S \neq T \quad R \neq T$$
We have again subcases based on which constraint sets match.

$\Theta = \Phi = \Psi$: In this branch the proof of transitivity elimination in $\lambda\&$ can be re-used.

$\Theta = \Phi \neq \Psi$: By Lemma 4 it follows from the subproof on the right side that there are proofs for:

$$S/\Theta \leq S/\Psi, \tag{A 3}$$

$$S/\Psi \leq T/\Psi. \tag{A 4}$$

Lemma 5 implies from the proof of equation (A 3) and the subproof on the left side that there is a proof for:

$$R/\Theta \leq R/\Psi. \tag{A 5}$$

Again from the subproof on the left side, it is known via Lemma 2 that

$$R/\Psi \leq S/\Psi \tag{A 6}$$

is provable. From the proofs of equations (A 4) and (A 6) one obtains a proof for

$$R/\Psi \leq T/\Psi$$

by the previous subcase. This and the proof of equation (A 5) are the input for a replacement rule [trans′].

$\Theta \neq \Phi = \Psi$: According to Lemma 4, from the subproof on the left side there are proofs for

$$R/\Phi \leq S/\Phi, \tag{A 7}$$

$$R/\Theta \leq R/\Phi. \tag{A 8}$$

The first subcase of the case being considered ensures from the subproof on the right side and the proof of equation (A 7) that a proof for

$$R/\Phi \leq T/\Phi$$

exists. This can be combined with the proof of equation (A 8) via rule [trans′] to give the proof sought.

$\Theta = \Psi \neq \Phi$: This one is easy. Lemma 4 assures the existence of proofs for

$$R/\Theta \leq S/\Theta,$$

$$S/\Theta \leq T/\Theta.$$

Now we can turn to the transitivity elimination with identical constraint sets.

$\Theta \neq \Phi \neq \Psi, \Theta \neq \Psi$: As usual, because of Lemma 4 used for the subproofs

$$R/\Theta \leq R/\Phi, \tag{A 9}$$

$$R/\Phi \leq S/\Phi, \tag{A 10}$$

$$R/\Psi \leq S/\Psi, \tag{A 11}$$

$$S/\Phi \leq S/\Psi, \tag{A 12}$$

$$S/\Psi \leq T/\Psi \tag{A 13}$$

are all provable. By applying Lemma 5 to the proofs of equations (A 10) and (A 12) a proof for

$$R/\Phi \leq R/\Psi \tag{A 14}$$

is obtained. Case 1 delivers a proof for

$$R/\Theta \leq R/\Psi \tag{A 15}$$

using the proofs for equations (A 9) and (A 14). Based on the proofs of equations (A 11) and (A 13) the subcase of transitivity elimination for $\lambda\&$ ensures that a proof for

$$R/\Psi \leq T/\Psi$$

exists. This and the proof for equation (A 15) are the input for a replacement rule [trans′]. □

A.4 Type algorithm

The proof of the similar theorem for $\lambda\&$ is adaptable in a straightforward manner as follows.

Lemma 6 (*Algorithmic soundness*)
If $\Gamma \Vdash_* M : V$ then $\Gamma \Vdash M : V$.

Proof
As in $\lambda\&$, rules [$\rightarrow$ elim$*$], [$\{\}$ intro$*$] and [$\{\}$ elim$*$] are to be preceded by rule [subsumption] and substituted by their counterpart labelled [$\rightarrow$ elim], [$\{\}$ intro] and [$\{\}$ elim] respectively. □

Lemma 7
If $\Gamma \Vdash M : V$ then there exists a $\Vdash$-derivation of $M : V$ from Γ where rule [subsumption] is not used twice consecutively.

Proof
Since subtyping is transitive, such steps can be reduced to a single application of rule [subsumption]. □

Lemma 8 (*Algorithmic completeness*)
Let Π be a proof for $\Gamma \Vdash M : V$. Then there exists U and Π' such that $U \leq V$, $depth(\Pi') \leq depth(\Pi)$ and Π' is a proof for $\Gamma \Vdash_* M : U$.

Proof
Thanks to Lemma 7, a simple induction on the depth of Π with appropriate branches for different term constructions suffices just as in $\lambda\&$. □

Theorem 4 (*Minimum typing*)
If $\Gamma \Vdash_* M : V$ then

$$V \in \{U | \Gamma \Vdash M : U \text{ and } \forall W \; \Gamma \Vdash M : W \Rightarrow U \leq W\}.$$

Proof
This proposition is a simple corollary of the preceding lemmas, as in $\lambda\&$. □

A.5 Soundness for specifications

The proof for the original $\lambda\&\mathfrak{C}$-calculus can be carried over because of the following lemma.

Lemma 9
Whenever $M_1 \bullet M_2 \triangleright_{\beta_\&} N_a$ and $M_1 \bullet M_2 \blacktriangleright_{\beta'_\&} N_b$ so that both M_1 and M_2 are considered in the reductions, then $\Gamma \Vdash N_a : V_a, N_b : V_b$ and $V_b \leq V_a$.

Proof
From the definition of types and reductions (branches j and k are selected by rules $(\beta_\&)$ and $(\beta'_\&)$, respectively):

$$\Gamma \Vdash M_1 : \{(U_i \to V_i)/\Theta_i\}_{i=1,\dots,n}/\Theta \quad \text{where } U_i \equiv R_i/\Phi_i \text{ and } R_i\text{'s are all different}$$

$$\Gamma \Vdash M_2 : U \equiv S/\Phi \quad \text{where } U \text{ is minimal}$$

$$U_j = \min_i\{U_i | U \leq U_i\} \tag{A 16}$$

$$U_k = \min_i\{U_i | S/\emptyset \leq R_i/\emptyset\} \tag{A 17}$$

Theorem 3 ensures that Lemma 4 is valid for the relation $\leq$ as well. By that lemma it follows then from equation (A 16) that $S/\emptyset \leq R_i/\emptyset$ for all i's which are selected into the set in equation (A 16). That is, all these indices are present in the set constructed in equation (A 17). Thus $R_k \leq R_j$ holds. By condition 3e of Definition 6, this implies $U_k \leq U_j$. From this and from type condition 3b via rule [{} elim] or rules [subsumption] and [$\to$ elim] one can infer that $V_b \equiv V_k \leq V_a \equiv V_j$. □

A.6 Confluence for specifications

Here we cannot cite the proof of confluence of $\lambda\&$, but formally the same proof steps can be carried out. Therefore we only list the supporting propositions.

Theorem 8 (*The notion of reduction $\beta'_\&$ satisfies the diamond property*)
For all M, M_1 and M_2, $M \blacktriangleright_{\beta'_\&} M_1$ and $M \blacktriangleright_{\beta'_\&} M_2$ imply that there exists M_3 such that $M_1 \blacktriangleright_{\beta'_\&} M_3$ and $M_2 \blacktriangleright_{\beta'_\&} M_3$. As a corollary $\beta'_\&$ is Church-Rosser.

Lemma 10
If $N \blacktriangleright^*_{\beta'_\&} N'$ then $M[x:=N] \blacktriangleright^*_{\beta'_\&} M[x:=N']$.

Lemma 11
If $M \blacktriangleright_{\beta'_\&} M'$ then $M[x:=N] \blacktriangleright_{\beta'_\&} M'[x:=N]$.

Theorem 9 (*Weak commutativity*)
If $M \blacktriangleright_\beta N_1$ and $M \blacktriangleright_{\beta'_\&} N_2$ then there exists N_3 such that $N_1 \blacktriangleright^*_{\beta'_\&} N_3$ and $N_2 \blacktriangleright^*_\beta N_3$. As a corollary $\blacktriangleright^*_{\beta'_\&}$ commutes with $\blacktriangleright^*_\beta$.

B Mapping Ghelli's Roles into $\lambda\&\mathfrak{C}$

In this section we show that the feature of roles of *Ghelli's kernel role calculus* (Ghelli, 2002) has its equivalent in $\lambda\&\mathfrak{C}$. More precisely, we present a transcription of the role construct of Ghelli's kernel calculus into $\lambda\&\mathfrak{C}$.

To begin with, we recapitulate Ghelli's object representation and member selection formalism. An object is a construct of the form

$$\langle r, [(r_i, l_i) = \varsigma(x_i : A)b_i]_{i \in I} \rangle,$$

where $r \in \{r_i\}_{i \in I}$ is the current role, r_i is a role-tag, l_i is the name of a member, x_i is a self variable, A is its type and b_i is the member term. As the member term can make use of the self-variable, members (which are called methods in the calculus) include both attribute values and operations. Method selection has the form of $a.l$, where a is an object and l is a method name. The selected method is (r_i, l_i), where $r_i \equiv r$ and $l_i \equiv l$.

The translation begins with the insight that as $\lambda\&\mathfrak{C}$ models attribute values and operations differently, they must be distinguished. It can be achieved, however, in a pure syntactic way, by checking whether a b_i refers to x_i. If yes, the method is an operation, if not, it is an attribute value. Furthermore, in accordance with the most common object-oriented modelling techniques (see e.g. (UML, 2003)) $\lambda\&\mathfrak{C}$ does not deal with operations assigned to objects directly but operations are parts of classes. **Precondition:** We therefore assume instances of a class have exactly the same operations.

Translating attribute values: Each attribute value becomes an overloaded function which has a branch for each role. Moreover, the current role itself is represented as an attribute value. That is

$$\langle r, [(r_i, l_i) = \varsigma(x_i : A)b_i]_{i \in I_{\text{attributes}}} \rangle$$

becomes here

$$\varepsilon \quad \& \quad \lambda x^{role}.r \quad \& \quad \lambda x^{l_{i1}}.(\varepsilon \; \& \; \lambda y^{\;{}_{y}role/\{y=r_{j11}\}}.b_{j11} \; \& \; \ldots) \quad \& \quad \ldots,$$

where $r_{j11}, r_{j12}, \ldots$ are role-tags which occur in the object with the method name l_{i1}. Then the attribute field selection

$$a.l \qquad \text{corresponds to} \qquad (a \bullet l) \bullet (a \bullet role) \qquad \text{in } \lambda\&\mathfrak{C}.$$

Translating operations: All operations with the same name form an overloaded function in $\lambda\&\mathfrak{C}$. This applies to operations of different roles as well. Thus

$$\langle r, [(r_i, l_i) = \varsigma(x_i : A)b_i]_{i \in I_{\text{operations}}} \rangle$$

becomes here

$$\varepsilon \quad \& \quad \lambda x_i^{A/\{role=r_{ij}\}}.b_{ij} \quad \& \quad \ldots \qquad \text{for each different } l_i$$

where as introduced above, A is the representation type of the class of the object and $role$ is one of its fields. Then the selection of a real method

$$a.l \qquad \text{corresponds to} \qquad l \bullet a \qquad \text{in } \lambda\&\mathfrak{C}.$$

Example 10

Ghelli's role construct is illustrated by the following value (*johnAsEmployee*) in his paper.

$$\begin{array}{l} \langle Emp, [\quad (Pers, Name) = \varsigma(x:A)\text{"John"}; \\ \quad (Stud, Name) = \varsigma(x:A)\text{"John"}; (Stud, IdCode) = \varsigma(x:A)100 \quad ; \\ \quad (Emp, Name) = \varsigma(x:A)\text{"John"}; (Emp, IdCode) = \varsigma(x:A)\text{"I1"} \quad]\rangle \end{array}$$

Since *johnAsEmployee* consists of attribute values only, its representation in $\lambda\&\mathbf{C}$ is the term

$$\begin{array}{l} \varepsilon \quad \& \quad \lambda x^{role}.Emp \quad \& \quad \lambda x^{Name}.(\varepsilon \ \& \\ \lambda y^{{}_{y}role/\{role=Pers\}}.\text{"John"} \ \& \ \lambda y^{{}_{y}role/\{role=Stud\}}.\text{"John"} \ \& \ \lambda y^{{}_{y}role/\{role=Emp\}}.\text{"John"} \\ \qquad) \quad \& \quad \lambda x^{IdCode}.(\varepsilon \ \& \\ \qquad \lambda y^{{}_{y}role/\{role=Stud\}}.100 \ \& \ \lambda y^{{}_{y}role/\{role=Emp\}}.\text{"I1"} \\ \qquad). \end{array}$$

It is easy to show in general that the subtyping relation and definition of type are compatible in Ghelli's and our calculus and that the term transformation presented above preserves term typability.

Besides the paradigmatic difference between the calculi (attribute value and operation similarity vs. distinction), which implies that a single term of Ghelli's calculus may be equivalent to a tuple of $\lambda\&\mathbf{C}$-terms, there is another factor which makes it difficult to *formally* and *fully* map Ghelli's kernel role calculus into $\lambda\&\mathbf{C}$: the member update facility. It is not present either in $\lambda\&$ or in $\lambda\&\mathbf{C}$ but Castagna deals with this phenomena in a more sophisticated variant of $\lambda\&$ (Castagna, 1997). His result may be applied to $\lambda\&\mathbf{C}$ as well. Nevertheless, the *feature* of role-tags can clearly be represented in $\lambda\&\mathbf{C}$-calculus.

References

Abadi, M. & Leino, K. R. M. (1998) *A logic of object-oriented programs.* Tech. rept. 161. Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301.

Börger, E., Grädel, E. & Gurevich, Y. (1997) *The classical decision problem.* 1st edn. Springer-Verlag Telos.

Boudol, G. (1997) Typing the use of resources in a concurrent calculus. *Pages 239–253 of: ASIAN'97.* Lecture Notes in Computer Science, vol. 1345. London, United Kingdom: Springer-Verlag.

Cardelli, L. (1988) A semantics of multiple inheritance. *Information and Computation*, **76**(2/3), 138–164.

Cardelli, L. (2004) Type systems. *Chap. 97 of:* Tucker, Jr., Allen B. (ed), *Computer science handbook*, 2nd edn. CRC Press.

Castagna, G. (1997) *Object-oriented programming: A unified foundation.* Progress in Theoretical Computer Science. Boston: Birkhäuser.

Curien, P.-L. & Ghelli, G. (1992) Coherence of subsumption, minimum typing and type checking in $F_{\leq}$. *Mathematical structures in computer science*, **2**(1), 55–91.

de Boer, F. S. (1999) A WP-calculus for OO. *Pages 135–149 of:* Thomas, W. (ed), *FOSSACS'99 (ETAPS'99)*. Lecture Notes in Computer Science, vol. 1578. London, United Kingdom: Springer-Verlag.

Ghelli, G. (2002) Foundations for extensible objects with roles. *Information and Computation*, **175**(1), 50–75.

Gottlob, G., Schrefl, M. & Röck, B. (1996) Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, **14**(3), 268–296.

Hoare, C. A. R. (1969) An axiomatic basis for computer programming. *Communications of the ACM*, **12**(10), 576–580, 583.

Hofmann, M., Naraschewski, W., Steffen, M., & Stroup, T. (1996) *Inheritance of proofs*. Tech. rept. IMMDVII-5/96. Universität Erlangen-Nürnberg.

Hofmann, M., Naraschewski, W., Steffen, M. & Stroup, T. (1998) Inheritance of proofs. *Theory and Practice of Object Systems*, **4**(1), 51–69.

Igarashi, A. & Kobayashi, N. (2004) A generic type system for the pi-calculus. *Theoretical Computer Science*, **311**(1–3), 121–163.

Kardkovács, Zs. T. & Surányi, G. M. (2004) An axiomatic model for deductive object-oriented databases. *Pages 325–336 of: Proceedings of the 5th international symposium of Hungarian researchers on computational intelligence*. Budapest Tech and Hungarian Fuzzy Association.

Kobayashi, N. & Yonezawa, A. (1994) Type-theoretic foundations for concurrent object-oriented programing. *Pages 31–45 of: OOPSLA'94*. ACM Press.

Krishnaswami, N. & Aldrich, J. (2005) Permission-based ownership: Encapsulating state in higher-order typed languages. *Pages 96–106 of: PLDI'05*. New York, NY, USA: ACM Press.

Li, L. (2004) Extending the Java language with dynamic classification. *Journal of object technology*, **3**(7), 101–120.

OCL (2003) *OCL2.0 — OMG final adopted specification*. Object Management Group, Inc.

Papazoglou, M. P. & Krämer, B. J. (1997) A database model for object dynamics. *The VLDB Journal*, **6**(2), 73–96.

Poetzsch-Heffter, A. & Müller, P. (1998) Logical foundations for typed object-oriented languages. *Pages 404–423 of:* Gries, David, & de Roever, Willem P. (eds), *PROCOMET'98*. IFIP Conference Proceedings, vol. 125. Chapman & Hall, Ltd.

Rundensteiner, E. A. (1992) *MultiView*: A methodology for supporting multiple views in object-oriented databases. *Pages 187–198 of:* Yuan, L.-Y. (ed.), *Proceedings of the 18th international conference on Very Large Data Bases*. Morgan Kaufmann.

UML (2003) *OMG unified modeling language specification, version 1.5*. Object Management Group, Inc.

Wand, M. (1987) Complete type inference for simple objects. *Pages 37–44 of: Proceedings of symposium on logic in computer science*. The Computer Society of IEEE.