Formal Aspects of Computing (2000) 12: 18–40
© 2000 BCS

Formal Aspects
of Computing

Class Refinement as Semantics of Correct Object Substitutability

Ralph-Johan Back, Anna Mikhajlova[1] and Joakim von Wright
Turku Centre for Computer Science, Åbo Akademi University, Finland

Abstract. Subtype polymorphism, based on syntactic conformance of objects' methods and used for substituting subtype objects for supertype objects, is a characteristic feature of the object-oriented programming style. While certainly very useful, typechecking of syntactic conformance of subtype objects to supertype objects is insufficient to guarantee correctness of object substitutability. In addition, the behaviour of subtype objects must be constrained to achieve correctness. In class-based systems classes specify the behaviour of the objects they instantiate. In this paper we define the class refinement relation which captures the semantic constraints that must be imposed on classes to guarantee correctness of substitutability in all clients of the objects these classes instantiate. Clients of class instances are modelled as programs making an iterative choice over invocation of class methods, and we formally prove that when a class C' refines a class C, substituting instances of C' for instances of C is refinement for the clients.

Keywords: Class refinement; Code inheritance; Behavioural compatibility; Object substitutability; Subclassing; Subtyping; Semantics of object-oriented constructs; Correctness; Implicit and explicit invariants; New methods

1. Introduction

The issue of correctness of object-oriented programs deserves close consideration in view of the popularity of this programming style and the necessity to enhance reliability of programs. Not only is correctness a crucial requirement for safety-critical systems, but also it is becoming increasingly important for distributed object-oriented systems and frameworks, which are composed by independent users and characterised by a late integration phase. We consider here a formal basis for ensuring correctness of class-based statically-typed object-oriented systems.

Subtype polymorphism, which is generally recognised as central to object-orientation, is based on syntactic conformance of objects' methods and used for substituting subtype objects for supertype objects. In most object-oriented languages, such as Simula, Eiffel, and C++, *subclassing* or *implementation inheritance* forms a basis for subtype polymorphism, i.e. signatures of subclass methods automatically conform to those of superclass methods, and, syntactically, subclass instances can be substituted for superclass instances. As

Correspondence and offprint requests to: Anna Mikhajlova, Department of Electronics & Computer Science, University of Southampton, Highfield, Southampton SO17 1BJ, UK. Email: aam@ecs.soton.ac.uk
[1] Currently at the Department of Electronics & Computer Science, University of Southampton, UK.

the mechanism of polymorphic substitutability is, to a great extent, independent of the mechanism of implementation reuse, languages like Java and Sather separate the subtyping and subclassing hierarchies.

With both approaches, typechecking can be used to verify syntactic conformance of subtype objects to their supertype objects. However, it has been recognised that, while certainly very useful, typechecking is insufficient to guarantee correctness of object substitutability. An attempt to establish behavioural conformance along with syntactic one has created a research direction known as *behavioural subtyping* [Ame87, LiW94, DhL95, LeW95, DhL96]. The essence of behavioural subtyping is to associate behaviour with type signatures and to identify subtypes that conform to their supertypes not only syntactically but also semantically.

In our view subtyping is a mechanism for substituting objects with certain method signatures for other objects with conforming method signatures and, as such, is a purely syntactic concept. Behaviour of objects, on the other hand, has little to do with their syntactic interfaces and is expressed in the specification of the objects' methods manipulating the objects' attributes. Most importantly, syntactic subtyping is decidable and can be checked by a computer, while behaviour-preserving subtyping is undecidable. Hence, in our approach we separate syntactic subtyping from behavioural conformance of subtype objects to supertype objects. We consider classes to be the carriers of behaviour and compare them for behavioural compatibility. Instances of one class are guaranteed to behave as expected from instances of another, more abstract, class if the more concrete class is a *refinement* of the more abstract. We give a definition of *class refinement*, which we regard as semantics of correct substitutability of subclass instances for superclass instances in clients. We formally prove that when a class C' refines a class C, substituting instances of C' for instances of C is refinement for the clients.

Our definition of class refinement guards against inconsistencies potentially introduced by new methods in the presence of subtype aliasing or in a general computational environment permitting sharing of objects by multiple clients. New methods introduced in class C' may break the strongest invariant of class C, and clients of C relying on this invariant may get invalidated when using instances of C' instead. New methods may take an instance of C' to a state which is perceived as unreachable from the perspective of a client relying of the strongest invariant of C. We formalise the notion of *consistent* new methods and prove that they preserve the strongest invariant of the class being refined, ensuring in this way safe substitutability of the corresponding objects in all clients.

Class refinement is orthogonal to subclassing. A class and its subclass may not be in refinement, and two classes can be in refinement even if one of them is not declared to be a subclass of the other. With separate interface inheritance and implementation inheritance hierarchies, a subclass may not even be intended for behavioural conformance with its superclass, as the substitution mechanism is completely independent of the reuse mechanism. Syntactic conformance of method signatures, however, is a prerequisite for class refinement, as it is meaningless to compare behaviour of classes whose instances are not intended for substitution. For simplicity, we consider subclassing to be the basis for subtyping and, consequently, require that class refinement be established between a class and its declared subclasses. However, the same principles also apply to systems with separate subclassing and interface inheritance hierarchies, as we will explain in the concluding section.

We build a logical framework for reasoning about object-oriented programs as a conservative extension of the *refinement calculus* [Mor90, BvW98], which is used for reasoning about correctness and refinement of imperative programs in a rigorous, mathematically precise manner. The refinement calculus is particularly suited for describing object-oriented programs because it allows us to describe classes at various abstraction levels, using *specification statements* along with ordinary executable statements. The notion of an *abstract class*, specifying behaviour common to its subclasses, can be fully elaborated in this formalisation, since the state of class instances can be given using abstract mathematical constructions, like sets and sequences, and class methods can be described as nondeterministic statements, abstractly but precisely specifying the intended behaviour. The versatility of the specification language that we use permits treating specifications and implementations in a uniform manner considering implementations to be just deterministic specifications.

The expressiveness of higher-order logic, which is the formal basis of the refinement calculus, allows us to define relations between classes, such as class refinement, entirely within logic. Reasoning about these relations can, therefore, be carried out completely formally. The detailed elaboration of our formalisation permits mechanised reasoning and mechanical verification, because, being so precisely defined, every concept can be formalised within a theorem proving environment such as HOL [GoM93] or PVS [ORS92].

2. Refinement Calculus Basics

We formalise objects, classes, and relationships between them in the refinement calculus, which is used for reasoning about correctness and refinement of imperative programs. Let us briefly introduce the main concepts of this formalism.

2.1. Predicates, Relations, and Predicate Transformers

A program state with components is modelled by a tuple of values, and a set of states (type) Σ is a product space, $\Sigma = \Sigma_1 \times \ldots \times \Sigma_n$.

A *predicate* over Σ is a boolean function $p : \Sigma \to Bool$ which assigns a truth value to each state. The set of predicates on Σ is denoted $\mathscr{P}\Sigma$. The *entailment ordering* on predicates is defined by pointwise extension. For $p, q : \mathscr{P}\Sigma$, p entails q, written $p \subseteq q$, is defined as follows:

$$p \subseteq q \;\widehat{=}\; (\forall \sigma : \Sigma \cdot p\ \sigma \Rightarrow q\ \sigma)$$

Conjunction $\cap$ and disjunction $\cup$ of (similarly-typed) predicates are also defined pointwise.

A *relation* from Σ to Γ is a function of type $\Sigma \to \mathscr{P}\Gamma$ that maps each state σ to a predicate on Γ. We write $\Sigma \leftrightarrow \Gamma$ to denote a set of all relations from Σ to Γ. This view of relations is isomorphic to viewing them as predicates on the Cartesian space $\Sigma \times \Gamma$. A function $f : \Sigma \to \Gamma$ can always be lifted to a (deterministic) relation $|\,f\,| : \Sigma \leftrightarrow \Gamma$. Functional and relational compositions are defined in a standard way.

A *predicate transformer* is a function $S : \mathscr{P}\Gamma \to \mathscr{P}\Sigma$ from predicates to predicates. We write

$$\Sigma \mapsto \Gamma \;\widehat{=}\; \mathscr{P}\Gamma \to \mathscr{P}\Sigma$$

to denote a set of all predicate transformers from Σ to Γ. The *refinement ordering* on predicate transformers is defined by pointwise extension from predicates. For $S, T : \Sigma \mapsto \Gamma$, S is refined by T, written $S \sqsubseteq T$, is defined as follows:

$$S \sqsubseteq T \;\widehat{=}\; (\forall q : \mathscr{P}\Gamma \cdot S\ q \subseteq T\ q)$$

Product operators combine predicates, functions, relations, and predicate transformers by forming Cartesian products of their state spaces. For example, a product $P \times Q$ of two relations $P : \Sigma_1 \leftrightarrow \Gamma_1$ and $Q : \Sigma_2 \leftrightarrow \Gamma_2$ is a relation of type $(\Sigma_1 \times \Sigma_2) \leftrightarrow (\Gamma_1 \times \Gamma_2)$ defined by

$$(P \times Q)\,(\sigma_1, \sigma_2)\,(\gamma_1, \gamma_2) \;\widehat{=}\; P\ \sigma_1\ \gamma_1 \wedge Q\ \sigma_2\ \gamma_2$$

For predicate transformers $S_1 : \Sigma_1 \mapsto \Gamma_1$ and $S_2 : \Sigma_2 \mapsto \Gamma_2$, their product $S_1 \times S_2$ is a predicate transformer of type $\Sigma_1 \times \Sigma_2 \mapsto \Gamma_1 \times \Gamma_2$ whose execution has the same effect as simultaneous execution of S_1 and S_2. In addition to many other useful properties, presented, e.g., in [BaB95, BvW97], the product operator preserves refinement:

$$S_1 \sqsubseteq S_1' \wedge S_2 \sqsubseteq S_2' \Rightarrow (S_1 \times S_2) \sqsubseteq (S_1' \times S_2')$$

For modelling subtype polymorphism and dynamic binding we employ *sum types*. The sum or disjoint union of two types Σ and Γ is written $\Sigma + \Gamma$. The types Σ and Γ are called *base types* of the sum in this case. Associated with the sum types, are the *injection functions* which map elements of the base type to elements of the summation

$$\imath_1 : \Sigma \to \Sigma + \Gamma \qquad \imath_2 : \Gamma \to \Sigma + \Gamma$$

and *projection relations* which relate elements of the summation with elements of its base types

$$\pi_1 : \Sigma + \Gamma \leftrightarrow \Sigma \qquad \pi_2 : \Sigma + \Gamma \leftrightarrow \Gamma$$

The projection is the inverse of the injection, so that $\pi_1^{-1} = |\,\imath_1\,|$, where $|\,\imath_1\,|$ is the injection function lifted to a relation. Since any element of $\Sigma + \Gamma$ comes either from Σ or from Γ, but not both, the ranges of the injections *ran* $\imath_1$ and *ran* $\imath_2$ partition $\Sigma + \Gamma$. For $\sigma : \Sigma + \Gamma$, the projection π_1 will relate it to a unique $\sigma' : \Sigma$ only if $\sigma \in$ *ran* $\imath_1$, and similarly for π_2. Sum types, as well as product types, associate to the right, so that $\Sigma_1 + \Sigma_2 + \Sigma_3 = \Sigma_1 + (\Sigma_2 + \Sigma_3)$.

We define the type Σ to be a subtype of Σ', written $\Sigma <: \Sigma'$, if $\Sigma = \Sigma'$, or $\Sigma <: \Sigma_i'$, where $\Sigma' = \Sigma_1' + \ldots + \Sigma_n'$

and $i \in \{1,\ldots,n\}$. For example, $\Sigma <: \Sigma + \Sigma'$ and, of course, $\Sigma + \Sigma' <: \Sigma + \Sigma'$. The subtype relation is reflexive, transitive, and antisymmetric. For any Σ and Σ' such that $\Sigma <: \Sigma'$, we can construct the corresponding injection function $\iota_\Sigma : \Sigma \to \Sigma'$ and projection relation $\pi_\Sigma : \Sigma' \leftrightarrow \Sigma$ in a straightforward way.

2.2. Specification Language

The language used in the refinement calculus includes executable statements along with (abstract) specification statements. Every statement has a precise mathematical meaning as a monotonic predicate transformer. A statement with initial state in Σ and final state in Γ determines a monotonic predicate transformer $S : \Sigma \mapsto \Gamma$ that maps any postcondition $q : \mathscr{P}\Gamma$ to the weakest precondition $p : \mathscr{P}\Sigma$ such that the statement is guaranteed to terminate in a final state satisfying q whenever the initial state satisfies p. A statement need not have identical initial and final state spaces, though if it does, we write $S : \Xi(\Sigma)$ instead of $S : \Sigma \mapsto \Sigma$ for the corresponding predicate transformer. Following an established tradition, we will from now on identify statements with the monotonic predicate transformers that they determine in this manner.

The total correctness assertion $p \; \{\!|\, S \,|\!\} \; q$ is said to hold if execution of the statement S establishes the postcondition q when started in the set of states p. The pair of state predicates (p, q) is usually referred to as the pre- and post-condition specification of the statement S. Formally, the total correctness assertion $p \; \{\!|\, S \,|\!\} \; q$ is defined to be $p \subseteq S\ q$. The refinement ordering on predicate transformers models the notion of total-correctness preserving program refinement. For statements S and T, the relation $S \sqsubseteq T$ holds if and only if T satisfies any specification satisfied by S.

$$S \sqsubseteq T \;=\; (\forall p\ q \cdot p \; \{\!|\, S \,|\!\} \; q \;\Rightarrow\; p \; \{\!|\, T \,|\!\} \; q)$$

Predicate transformers form a complete lattice under the refinement ordering. The bottom element is the predicate transformer **abort** that maps each postcondition to the identically false predicate *false*, and the top element is the predicate transformer **magic** that maps each postcondition to the identically true predicate *true*. We know nothing about how **abort** is executed and it is never guaranteed to terminate. The **magic** statement is *miraculous* since it is always guaranteed to establish any postcondition; as such, **magic** is the opposite of the abortion and is not considered to be an error. Intuitively, **magic** can be understood as an infinite *wait* statement, and, although not directly implementable, it serves as a convenient abstraction in manipulating program statements.

Conjunction $\sqcap$ and disjunction $\sqcup$ of (similarly-typed) predicate transformers are defined pointwise, e.g.,

$$(\sqcap\, i \in I \cdot S_i)\ q \;\hat{=}\; (\cap\, i \in I \cdot S_i\ q)$$

Both conjunction and disjunction of predicate transformers model *nondeterministic choice* among executing either of S_i. Conjunction models *demonic* nondeterministic choice in the sense that nondeterminism is uncontrollable and each alternative must establish the postcondition. Disjunction, on the other hand, models *angelic* nondeterminism, where the choice between alternatives is free and aimed at establishing the postcondition.

Sequential composition of program statements is modelled by functional composition of predicate transformers and the program statement **skip** is modelled by the identity predicate transformer:

$$\begin{array}{lcl} (S;T)\ q & \hat{=} & S\ (T\ q) \\ \textbf{skip}\ q & \hat{=} & q \end{array}$$

Given a function $f : \Sigma \to \Gamma$ and a relation $P : \Sigma \leftrightarrow \Gamma$, the *functional update* $\langle f \rangle : \Sigma \mapsto \Gamma$, the *angelic update* $\{P\} : \Sigma \mapsto \Gamma$, and the *demonic update* $[P] : \Sigma \mapsto \Gamma$ are defined by

$$\begin{array}{lcl} \langle f \rangle\ q\ \sigma & \hat{=} & q\ (f\ \sigma) \\ \{P\}\ q\ \sigma & \hat{=} & (\exists\, \gamma : \Gamma \cdot P\ \sigma\ \gamma \;\wedge\; q\ \gamma) \\ [P]\ q\ \sigma & \hat{=} & (\forall \gamma : \Gamma \cdot P\ \sigma\ \gamma \;\Rightarrow\; q\ \gamma) \end{array}$$

The functional update applies the function f to the state σ to yield the new state $f\ \sigma$. When started in a state σ, $\{P\}$ angelically chooses a new state γ such that $P\ \sigma\ \gamma$ holds, while $[P]$ demonically chooses a new state γ such that $P\ \sigma\ \gamma$ holds. If no such state exists, then $\{P\}$ aborts, whereas $[P]$ behaves as **magic**. For the identity function id and the identity relation Id, all of $\langle id \rangle$, $\{Id\}$, and $[Id]$ behave as **skip**.

Following [BvW97], we use the notions of assignments, program variables, and variable declarations based on a simple syntactic extension to the typed lambda calculus. For a function $(\lambda u \cdot t)$ which replaces the old state u with the new state t, changing some components $x_1, \ldots, x_m$ of u while leaving the others unchanged, the *functional assignment* describing such a state change is defined by

$$(\lambda u \cdot x_1, \ldots, x_m := t_1, \ldots, t_m) \;\widehat{=}\; (\lambda u \cdot u[x_1, \ldots, x_m := t_1, \ldots, t_m])$$

For a relation $(\lambda u \cdot \lambda u' \cdot b)$, which using the set notation could also be written as $(\lambda u \cdot \{u' \mid b\})$, changing a component x of state u to some x' related to x via a boolean expression b, the *relational assignment* is defined by

$$(\lambda u \cdot x := x' \mid b) \;\widehat{=}\; (\lambda u \cdot \{u[x := x'] \mid b\})$$

As such, the notation for both functional and relational assignments is a convenient syntactic abbreviation for the corresponding lambda term describing a certain state change. Unfortunately, lambda terms do not maintain consistent naming of state components, due to the possibility of α-conversion of bound variables. To enforce naming consistency, we use the program variable notation, writing, e.g., $(\mathbf{var}\ x, y \cdot (x := x+y); (y := 0))$ to express that each function term is to be understood as a lambda abstraction over the bound variables x, y:

$$(\mathbf{var}\ x, y \cdot (x := x+y); (y := 0)) \;=\; (\lambda x, y \cdot x := x+y); (\lambda x, y \cdot y := 0)$$

Ordinary program statements may be modelled using the basic predicate transformers and operators presented above, using the program variable notation. For example, the (multiple) assignment statement may be modelled by the functional update:

$$(\mathbf{var}\ u \cdot x_1, \ldots, x_m := t_1, \ldots, t_m) \;\widehat{=}\; \langle \lambda u \cdot x_1, \ldots, x_m := t_1, \ldots, t_m \rangle$$

Our specification language includes specification statements. The *demonic assignment* and the *angelic assignment* are modelled by the demonic and the angelic updates respectively:

$$\begin{array}{lcl} [\mathbf{var}\ u \cdot x := x' \mid b] & \widehat{=} & [\lambda u \cdot x := x' \mid b] \\ \{\mathbf{var}\ u \cdot x := x' \mid b\} & \widehat{=} & \{\lambda u \cdot x := x' \mid b\} \end{array}$$

Intuitively, the demonic assignment expresses an uncontrollable nondeterministic choice in selecting a new value x' satisfying b, whereas the angelic assignment expresses a free choice. The angelic assignment can, e.g., be understood as a request to the user to supply a new value.

Our specification language also includes the *assertion* and the *assumption* statements, written $\{b\}$ and $[b]$ respectively, where b is a predicate stating a condition on program variables. Both the assertion and the assumption behave as **skip** if b is satisfied; otherwise, the assertion aborts, whereas the assumption behaves as **magic**.

The *conditional* statement is defined by the demonic choice of guarded alternatives or the angelic choice of asserted alternatives:

$$\mathbf{if}\ g\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi} \;\widehat{=}\; [g]; S_1 \sqcap [\neg g]; S_2 \;=\; \{g\}; S_1 \sqcup \{\neg g\}; S_2$$

Iteration is defined as the least fixpoint of a function on predicate transformers with respect to the refinement ordering:

$$\mathbf{while}\ g\ \mathbf{do}\ S\ \mathbf{od} \;\widehat{=}\; (\mu X \cdot \mathbf{if}\ g\ \mathbf{then}\ S; X\ \mathbf{else\ skip\ fi})$$

A variant of iteration, the *iterative choice* [BvW98, BMW99], allows the user to choose repeatedly an alternative that is enabled and have it executed until the user decides to stop:

$$\begin{array}{l} \mathbf{do}\ g_1 :: S_1 \between \ldots \between g_n :: S_n\ \mathbf{od} \;\widehat{=} \\ \quad (\mu X \cdot \{g_1\}; S_1; X \sqcup \ldots \sqcup \{g_n\}; S_n; X \sqcup \mathbf{skip}) \end{array}$$

We will abbreviate $g_1 :: S_1 \between \ldots \between g_n :: S_n$ by $\between_{i=1}^{n} g_i :: S_i$.

Finally, the language supports blocks with *local variables*. Introduction and removal of new variables are modelled by demonic and functional updates respectively:

$$\begin{array}{lcl} \mathbf{enter}\ p & \widehat{=} & [\lambda u \cdot \lambda(x, u') \cdot p\,(x, u') \;\wedge\; u = u'] \\ \mathbf{exit} & \widehat{=} & \langle \lambda(x, u) \cdot u \rangle \end{array}$$

Here p is the predicate initializing the new variables. We can define a block introducing a local variable x initialised according to a boolean expression b as follows:

$$\begin{array}{l} (\mathbf{var}\ u \cdot \mathbf{begin}\ (\mathbf{var}\ x, u \cdot b); S; \mathbf{end}) \;\widehat{=} \\ \quad (\mathbf{var}\ u \cdot \mathbf{enter}\ (\mathbf{var}\ x, u \cdot b); S; \mathbf{exit}) \end{array}$$

When the variable declaration is clear from the context, we will for simplicity write just **begin var** $x \cdot b; S;$ **end**.

The program variable declaration can be propagated outside statements and distributed through sequential composition, so that, e.g.,

$$(\mathbf{var}\ u \cdot [x := x' \mid x' \geqslant 0]; y := x) \ = \\ [\mathbf{var}\ u \cdot x := x' \mid x' \geqslant 0]; (\mathbf{var}\ u \cdot y := x)$$

When the variable declaration is clear from the context, we will omit it.

As suggested in [BvW98], we can define an interactive executable language, where a statement is built by the following syntax:

S	$::=$	**abort**	*(abortion)*
	$\mid$	**skip**	*(skip)*
	$\mid$	$\{b\}$	*(assertion)*
	$\mid$	$x_1, \ldots, x_m := t_1, \ldots, t_m$	*(assignment)*
	$\mid$	$\{x := x' \mid b\}$	*(angelic assignment)*
	$\mid$	$S_1; S_2$	*(sequential composition)*
	$\mid$	**if** g **then** S_1 **else** S_2 **fi**	*(conditional statement)*
	$\mid$	**while** g **do** S **od**	*(iteration)*
	$\mid$	**do** $g_1 :: S_1 \between \ldots \between g_n :: S_n$ **od**	*(iterative choice)*
	$\mid$	**begin var** $x \cdot b; S;$ **end**	*(block with local variables)*

When extended with miraculous statements, this executable language becomes a general specification language:

S	$::=$	$\ldots$	
	$\mid$	**magic**	*(magic)*
	$\mid$	$[b]$	*(assumption)*
	$\mid$	$[x := x' \mid b]$	*(demonic assignment)*

In the next section we explain how to extend this language with object-oriented constructs.

2.3. Data Refinement

Data refinement is a general technique by which one can change the state space in a refinement. For statements $S : \Xi(\Sigma)$ and $S' : \Xi(\Sigma')$, let $R : \Sigma' \leftrightarrow \Sigma$ be a relation between the state spaces Σ and Σ'. According to [Bac89], the statement S is said to be data refined by S' via R, denoted $S \ \sqsubseteq_R \ S'$, if

$$\{R\};\ S \ \sqsubseteq \ S';\ \{R\}$$

This notion of data refinement is the standard one, often referred to as forward data refinement or downward simulation. Alternative and equivalent characterizations of data refinement using the inverse relation R^{-1} are then

$$S; [R^{-1}] \ \sqsubseteq \ [R^{-1}]; S' \qquad S \ \sqsubseteq \ [R^{-1}]; S'; \{R\} \qquad \{R\}; S; [R^{-1}] \ \sqsubseteq \ S'$$

These characterizations follow from the fact that $\{R\}$ and $[R^{-1}]$ form a Galois connection, i.e. $\{R\}; [R^{-1}] \sqsubseteq$ **skip** and **skip** $\sqsubseteq [R^{-1}]; \{R\}$. Further on we will abbreviate $\{R\}; S; [R^{-1}]$ by $S \downarrow R$ and $[R^{-1}]; S'; \{R\}$ by $S' \uparrow R$. The refinement calculus provides rules for transforming more abstract program structures into more concrete ones based on the notion of refinement of predicate transformers presented above. A large collection of algorithmic and data refinement rules is given, for instance, in [BvW98, Mor90]. In the Appendix, we present some of these rules, used in proofs of theorems and lemmas in this paper.

3. Modelling Object-Oriented Constructs

We focus on modelling class-based statically-typed object-oriented languages, which form the mainstream of object-oriented programming. Accordingly, we take a view that *objects* are instances of *classes*. A class describes objects with similar behaviour through specifying their *interface*. The interface represents signatures of object methods, i.e. the method name and the types of value and result parameters. For simplicity, we consider all object attributes as private or hidden, and all methods as public or visible to clients of the object. We consider an *object type* to be the type of object attributes having an additional unique global identifier distinguishing this object type from the others. A class can be given by the following declaration:

$$\begin{array}{l} C \;=\; \textbf{class} \\ \quad \textbf{var}\; attr_1 : \Sigma_1, \ldots, attr_m : \Sigma_m \\ \\ \quad C\,(\textbf{val}\; x_0 : \Gamma_0) \;=\; K, \\ \quad Meth_1\,(\textbf{val}\; x_1 : \Gamma_1, \textbf{res}\; y_1 : \Delta_1) \;=\; M_1, \\ \quad \ldots \\ \quad Meth_n\,(\textbf{val}\; x_n : \Gamma_n, \textbf{res}\; y_n : \Delta_n) \;=\; M_n \\ \textbf{end} \end{array}$$

This class specifies the interface $Meth_1\,(\textbf{val} : \Gamma_1, \textbf{res} : \Delta_1), \ldots, Meth_n\,(\textbf{val} : \Gamma_n, \textbf{res} : \Delta_n)$, where Γ_i and Δ_i are the types of value and result parameters respectively. A method may be parameterless, with both Γ_i and Δ_i being the unit type (), or may have only value or only result parameters.

The class C describes (possibly abstract) attributes, specifies the way the objects are created, and gives a (possibly nondeterministic) specification for each method. Class attributes $(attr_1, \ldots, attr_m)$ have the corresponding types Σ_1 through Σ_m. Apart from the declared attributes, every class has an implicit constant attribute $type : String$ which contains the name of the object type specified by this class. This constant identifier is unique in every class. We will use an identifier *self* for the tuple $(attr_1, \ldots, attr_m, type)$. The type of *self* is then $\Sigma \;=\; \Sigma_1 \times \ldots \times \Sigma_m \times String$. We impose a non-recursiveness restriction on Σ so that none of Σ_i is equal to Σ. This restriction allows us to stay within the simple-typed lambda calculus, and is not a major limitation, as pointers introduced in Sec. 3.3 allow (indirectly) recursive types to be modelled.

A *class constructor* is used to instantiate objects and has the same name as the class. Due to the fact that the constructor concerns object creation rather than object functionality, it is associated with the class rather than with the specified interface. Semantically, the constructor is equivalent to a stand-alone global procedure which is associated with the class for encapsulation reasons. The statement $K : \Gamma_0 \mapsto \Sigma \times \Gamma_0$, representing the body of the constructor, introduces the attributes into the state space and initialises them using the value parameter(s) $x_0 : \Gamma_0$. Methods $Meth_1$ through $Meth_n$, specified by bodies $M_1, \ldots, M_n$, operate on the attributes and realise the object functionality. Every statement M_i is, in general, of type $\Xi(\Sigma \times \Gamma_i \times \Delta_i)$. The identifier *self* acts in this model as an implicit result parameter of the constructor and an implicit variable parameter of the methods.

Being declared as such, the class C is a tuple $(K, M_1, \ldots, M_n)$. Further on we will refer to K as the constructor and to $M_1, \ldots, M_n$ as the methods, unless stated otherwise. The object type specified by a class can always be extracted from the class and we do not need to declare it explicitly. We use $\tau(C)$ to denote the type of objects generated by the class C; as such, $\tau(C)$ is just another name for Σ.

3.1. Object Instantiation and Method Invocations

Initialization of a new variable c of object type $\tau(C)$ involves invoking the corresponding class constructor:

$$\begin{array}{rcl} \textbf{create var}\; c.C(e) & \widehat{=} & \textbf{enter}\;(\textbf{var}\; x_0, u \cdot x_0 = e); K \times \textbf{skip}; \\ & & \textbf{enter}\;(\textbf{var}\; c, (self, x_0), u \cdot c = self); Swap; \textbf{exit} \end{array}$$

where $Swap = \langle \lambda x, y, z \cdot y, x, z \rangle$. A variable $x_0 : \Gamma_0$ is first entered into the state space and initialised with the value of e. Then the constructor K is "injected" into the global state space, skipping on the global state component u. The next statement enters a variable c and initialises it to the value of the state component *self*. The state rearranging *Swap* makes the pair $(self, x_0)$ the first state component before exiting it from the block. Naturally, a variable of an object type initialised in this way can be local to a block:

$$\textbf{create var}\; c.C(e); S; \textbf{end} \;\widehat{=}\; \textbf{create var}\; c.C(e); S; \textbf{exit}$$

Invocation of a method $Meth_i(\textbf{val}\; x_i : \Gamma_i, \textbf{res}\; y_i : \Delta_i)$ on an object c instantiated by class C is modelled by

$$\begin{array}{rcl} (\textbf{var}\; c, u \cdot c.Meth_i\,(g_i, d_i)) & \widehat{=} & \textbf{begin}\;(\textbf{var}\;(self, x_i, y_i), c, u \cdot \\ & & \quad self = c \wedge x_i = g_i); \\ & & \quad M_i \times \textbf{skip}; c, d_i := self, y_i; \\ & & \textbf{end} \end{array}$$

where $u : \Phi$ are global variables including $d_i : \Delta_i$, and $g_i : \Gamma_i$ is some expression.

3.2. Modelling Object Clients

A client program using an object $c : \tau(C)$ does so by invoking its methods. Every time a client has a choice of which method to choose for execution. In general, each option is preceded with an assertion which determines whether the option is enabled in a particular state. While at least one of the assertions holds, the client may repeatedly choose a particular option which is enabled and have it executed. The client decides on its own when it is willing to stop choosing options. Such an iterative choice of method invocations, followed by arbitrary statements not affecting the object directly, describes all the actions the client program might undertake:

$$(\textbf{var}\ c, u \cdot \textbf{begin var}\ l \cdot b; \textbf{do}\ \lozenge_{i=1}^{n} q_i :: c.Meth_i(g_i, d_i); L_i\ \textbf{od}; \textbf{end})$$

Here $u : \Phi$ are global variables including d_i, $l : \Lambda$ are some local variables initialised according to b, predicates $q_1 \dots q_n$ are the asserted conditions on the state, and statements L_1 through L_n are arbitrary. The initialization b, the assertions $q_1 \dots q_n$, and the statements $L_1, \dots, L_n$ do not refer to c, which is justified by the assumption that the object state is encapsulated. Therefore, c is not free in b, every q_i is of the form $q_i' \times true \times q_i''$, with $q_i' : \mathscr{P}\Lambda$ and $q_i'' : \mathscr{P}\Phi$, and every L_i is of the form $L_i' \times \textbf{skip} \times L_i''$, with $L_i' : \Xi(\Lambda)$ and $L_i'' : \Xi(\Phi)$.

Note that we do not allow clients to enquire objects about their types, classes, attribute and method names. Although sometimes useful, such introspection facilities are generally regarded as unsafe. Also, a client cannot copy an object directly, but only through invoking a *Copy* method if one is supplied in the interface of the corresponding object.

Objects can, of course, be their own clients, and any method of class C can invoke other methods of C, including itself. The most general behaviour of a method $Meth_j(\textbf{val}\ x_j : \Gamma_j, \textbf{res}\ y_j : \Delta_j)$ can therefore be described by

$$\begin{aligned}&(\textbf{var}\ self, x_j, y_j \cdot \\ &\quad \textbf{begin var}\ l \cdot b; \textbf{do}\ \lozenge_{i=1}^{n} q_i :: self.Meth_i(g_i, d_i); L_i\ \textbf{od}; \textbf{end})\end{aligned}$$

where *self* is free in b, all q_i can directly access *self*, and all L_i can directly access and modify it. For example, a method self-calling itself is an instance of this general definition. The meaning of such a recursive method is given by the least fixpoint of the corresponding function with respect to the refinement ordering.

3.3. Modelling Dynamic Objects

Following [BvW98], we model pointers to class instances as indices of an array of these class instances. Natural numbers can be used as the index set of such an array, and we can declare a program variable *heap* to contain the whole dynamic data structure:

$$\textbf{var}\ heap : \textbf{array}\ Nat\ \textbf{of}\ \tau(C)$$

The type of pointers to instances of class C can then simply be defined as the type of natural numbers. The index value 0 can be used as a *nil* pointer. New pointer (index) values can be generated dynamically, on demand, by keeping a separate counter *new* for the next unused index:

$$\begin{aligned}&\textbf{type pointer to}\ \tau(C)\ \widehat{=}\ Nat; \\ &\textbf{const}\ nil : \textbf{pointer to}\ \tau(C) = 0; \\ &\textbf{var}\ new : \textbf{pointer to}\ \tau(C) := 1;\end{aligned}$$

There is a separate heap for each object type. Dynamic creation of an object of type $\tau(C)$ and association of a pointer p : **pointer to** $\tau(C)$ with this object are modelled as follows:

$$\begin{aligned}p := \textbf{new}\ C(e)\ \widehat{=}\ &p, new := new, new + 1; \\ &\textbf{create var}\ c.C(e); heap[p] := c; \textbf{end}\end{aligned}$$

To keep the array of class instances implicit, the notation $p\uparrow$ is used for the access operation $heap[p]$, so that $p\uparrow := e$ stands for the update operation $heap[p] := e$, and $p\uparrow.Meth_i(g_i, d_i)$ stands for the method invocation $heap[p].Meth_i(g_i, d_i)$.

```
$TextDoc$ = **class**
  **var** $text : String$,
      $views$ : **set of pointer to** $\tau(View)$

  $TextDoc$ (**val** $t : String$) =
    **enter var** $text, views \cdot$
      $text = t \wedge views = \{\}$,

  $AddView$ (**val** $v$ : **pointer to** $\tau(View)$) =
    $\{v \neq nil\}; views := views \cup \{v\}$,

  $AddText$ (**val** $t : String$) =
    $text := text \frown t; self.Notify()$,

  $GetText$ (**res** $t : String$) = $t := text$,

  $Notify$ () =
    **begin var** $(vs, v) \cdot vs = views$;
      **while** $vs \neq \{\}$ **do**
        $[v := v' \mid v' \in vs]$;
        $vs := vs \setminus v; v\uparrow.Update()$
      **od**;
    **end**
**end**
```

```
$View$ = **class**
  **var** $txt : String$,
      $doc$ : **pointer to** $\tau(TextDoc)$

  $View$ (**val** $d$ : **pointer to**
    $\tau(TextDoc)$) =
    $\{d \neq nil\}$;
    **enter var** $txt, doc \cdot doc = d$;
      $doc\uparrow.GetText(txt)$,

  $AddText$ (**val** $t : String$) =
    $doc\uparrow.AddText(t)$,

  $Update$ () = $doc\uparrow.GetText(txt)$
**end**
```

Fig. 1. Example of class specification.

3.4. Example

As an example of class specifications consider a text-editing application in which a text document may be viewed and possibly changed in several different windows. Whenever the text is changed in any of the windows in response to, e.g., user actions, all the other windows displaying the same text are notified of this change and updated to achieve consistency in presenting the data. We specify these interactions in Fig. 1. Views are responsible for presenting the textual data in various windows and providing operations for changing it. A text editor can open a new text document and display it in several different windows. For this, it needs to invoke the constructor *TextDoc*, create *View* instances using the corresponding constructor, with the pointer to the newly-created text document passed as an argument, and finally attach these *View* instances to the text document by invoking the method *AddView*. When one of the views is asked to add some text to the existing one, the method *AddText* is invoked. This method forwards the request to the method *AddText* of the current view's *doc* attribute. After the new text is concatenated to the old one, all views on the document are notified of the change and asked to update their state.

A point to notice here is that such a specification, although being rather abstract, precisely documents the behaviour of the involved parties without resorting to verbal descriptions. The necessity for a precise documentation was pointed out in [GHJV95] when discussing the Observer pattern which our example follows. In particular, it was advised to document which *Subject* (in our case *TextDoc*) methods trigger modifications. Also, the place of the *Notify* method invocation can be fixed in the specification. We chose to call it from the state-modifying *AddText* method of *TextDoc* after the change. Alternatively, this method could be called from *AddText* of *View* after invoking the corresponding method on the *doc* attribute. The advantages and disadvantages of both approaches are discussed in [GHJV95], we only would like to note that fixing the invocation of this method in the specification helps avoiding the problem of calling this method at inappropriate times or, even worse, not calling it at all from the overridden methods in subclasses of *TextDoc* and *View*.

3.5. Subclassing

New classes can be constructed from existing ones by *inheriting* some or all of their attributes and methods, possibly *overriding* some attributes and methods, and adding extra methods. This mechanism is known as *subclassing*.[2]

[2] We prefer the term *subclassing* to *implementation inheritance* because the latter literally means reuse of existing methods and does not, as such, suggest the possibility of method overriding.

A class constructed from C by subclassing is declared as follows:

$$\begin{array}{l}
C' \;=\; \textbf{subclass of } C \\
\quad \textbf{var } attr_1 : \Sigma_1, \ldots, attr_i : \Sigma_i, attr'_1 : \Sigma'_1, \ldots, attr'_p : \Sigma'_p \\
\quad C'(\textbf{val } x'_0 : \Gamma'_0) \;=\; K', \\
\quad Meth_1(\textbf{val } x_1 : \Gamma_1, \textbf{res } y_1 : \Delta_1) \;=\; M'_1, \\
\quad \ldots \\
\quad Meth_k(\textbf{val } x_k : \Gamma_k, \textbf{res } y_k : \Delta_k) \;=\; M'_k, \\
\quad NMeth_1(\textbf{val } u_1 : \Phi_1, \textbf{res } v_1 : \Psi_1) \;=\; N_1, \\
\quad \ldots \\
\quad NMeth_p(\textbf{val } u_p : \Phi_p, \textbf{res } v_p : \Psi_p) \;=\; N_p
\end{array}$$

A subclass may have attributes different from those of its superclass, inheriting $attr_1, \ldots, attr_i$ and overriding $attr_{i+1}, \ldots, attr_m$ by $attr'_1, \ldots, attr'_p$. The class constructor is not inherited from the superclass, but rather redefined in every subclass. The statements $M'_1, \ldots, M'_k$ override the corresponding definitions of $Meth_1, \ldots, Meth_k$ given in C. The methods $NMeth_1, \ldots, NMeth_p$ with bodies given by $N_1, \ldots, N_p$ are new.

When a subclass C' inherits all attributes of its superclass C without overriding them, methods defined in the superclass can be invoked from methods $M'_1, \ldots, M'_k, N_1, \ldots, N_p$ using a special identifier *super*. For example, a method $Meth_i(\textbf{val } x_i : \Gamma_i, \textbf{res } y_i : \Delta_i)$ defined in C by M_i can be super-called inside any of $M'_1, \ldots, M'_k, N_1, \ldots, N_p$ by writing $super.Meth_i(g_i, d_i)$, where g_i and d_i are some value and result arguments respectively. Such a super-call corresponds to executing statement $M_i \times \textbf{skip}$, with **skip** operating on the additional attributes of C'. Methods of the superclass can also be inherited as a whole. In this case their redefinition in the subclass corresponds to super-calling them, passing value and result parameters as arguments. Following the standard convention, we omit such inherited methods from the subclass declaration.

We view subclassing as a syntactic relation on classes, since subclasses are distinguished by an appropriate declaration. Subclassing implies conformance of interfaces, meaning that the interface specified by a subclass is an extension of the interface specified by the superclass, having at least all the method signatures of the latter and possibly introducing new ones. In an extended interface the inherited method signatures can be modified to allow more flexibility in polymorphic object substitutability. In the next section we explain how this can be achieved.

3.6. Modelling Subtype Polymorphism and Dynamic Binding

To model subtype polymorphism, we allow object types to be sum types. The idea is to group together an object type of a certain class and object types of all its subclasses, to form a polymorphic object type. A variable of such a sum type can be instantiated to any base type of the summation, in other words, to any object instantiated by a class whose object type is the base type of the summation.

A sum of object types, denoted by $\tau(C)^+$ is defined to be such that its base types are $\tau(C)$ and all the object types of subclasses of C. For example, if D is the only subclass of C with the object type $\tau(D)$, then $\tau(C)^+ \;=\; \tau(C) \;+\; \tau(D)$, and we have that

$$\tau(C) <: \tau(C)^+ \text{ and } \tau(D) <: \tau(C)^+$$

The diagram in Fig. 2 illustrates the relationship between subclassing and subtyping hierarchies. The subclassing hierarchy on the left-hand side corresponds to the subtyping hierarchy on the right-hand side, with the arrows meaning "is the type of instances of".

Suppose a method $Meth_i(\textbf{val } x_i : \Gamma_i, \textbf{res } y_i : \Delta_i)$ is specified in both C and D. An invocation of this method on an object p of type $\tau(C)^+$ is modelled as a choice between two alternatives each calling $Meth_i$, but one assuming that p is instantiated by class C and the other assuming instantiation by class D:

$$p.Meth_i(g_i, d_i) \;\widehat{=}\;
\left(\begin{array}{l}
\{p \in ran\ \iota_{\tau(C)}\}; \\
\textbf{begin var } c \cdot \pi_{\tau(C)}\ p\ c; \\
\quad c.Meth_i(g_i, d_i); \\
\quad p := \iota_{\tau(C)}\ c; \\
\textbf{end}
\end{array}\right)
\sqcup
\left(\begin{array}{l}
\{p \in ran\ \iota_{\tau(D)}\}; \\
\textbf{begin var } d \cdot \pi_{\tau(D)}\ p\ d; \\
\quad d.Meth_i(g_i, d_i); \\
\quad p := \iota_{\tau(D)}\ d; \\
\textbf{end}
\end{array}\right)$$

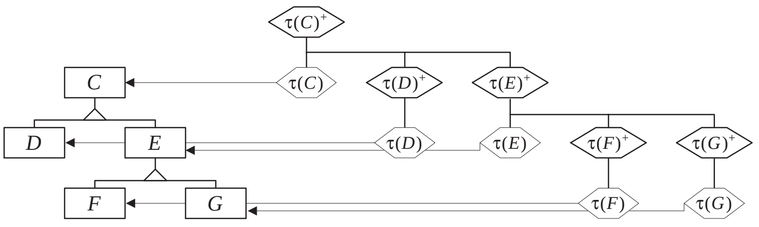

Fig. 2. The relationship between subclassing and subtyping hierarchies.

When p is an instance of C, the assertion $\{p \in ran\ \iota_{\tau(C)}\}$ skips, and the method $Meth_i$ is invoked on the object c corresponding to the projection $\pi_{\tau(C)}$ of p. Afterwards, the value of c is injected to be of type $\tau(C)^+$ and used to update p. The invocation $c.Meth_i(g_i, d_i)$ is modelled as in Sec. 3.1. The assertion that p is an instance of D is false, aborting the second alternative of the angelic choice, since for all predicates q, $\{q\}$ = **if** q **then skip else abort fi**, and for all statements S, **abort**;S = **abort**. The angelic choice between the two statements is then equal to the first alternative, since $S \sqcup$ **abort** $= S$. Similarly, when p is an instance of D, the first alternative aborts, and the choice is equal to the second alternative. As such, the choice between the alternatives is deterministic.

A polymorphic variable $p : \tau(C)^+$ can be instantiated by either class C or its subclass. In practice we occasionally would like to underspecify which particular class instantiates p. We can express this by using a demonic choice of possible instantiations. We will write $p.C^+(e)$, where e is any expression of type Γ_0, for this kind of polymorphic instantiation:

$$\textbf{create var } p.C^+(e) \ \widehat{=}$$
$$\left(\begin{array}{l}\textbf{create var } c.C(e);\\ \textbf{begin var } p \cdot \pi_{\tau(C)}\ p\ c;\\ \quad Swap;\\ \textbf{end}\end{array}\right) \sqcap \left(\begin{array}{l}\textbf{create var } d.D(e);\\ \textbf{begin var } p \cdot \pi_{\tau(D)}\ p\ d;\\ \quad Swap;\\ \textbf{end}\end{array}\right)$$

Intuitively, the demonic choice can be interpreted as underspecification, which would eventually be eliminated in a refinement. Note that since the demonic choice is refined by either alternative, we have that any concrete instantiation refines the polymorphic instantiation. Modelling of both the invocation of a method on a polymorphic variable and the instantiation of such a variable generalises to class hierarchies with several classes in a straightforward way, recursively.

Being equipped with subtype polymorphism, we can allow overriding methods in a subclass to be *generalised* on the type of value parameters or *specialised* on the type of result parameters. In the first case this type redefinition is *contravariant* and in the second *covariant*.[3] When one interface is the same as the other, except that it can redefine contravariantly value parameter types and covariantly result parameter types, this interface *conforms* to the original one. For example, $Meth_i$ (**val** $x_i : \Gamma_i$, **res** $y_i : \Delta_i$) specified in class C could be redefined in its subclass D so that the value parameters are of type Γ'_i, such that $\Gamma_i <: \Gamma'_i$, and the result parameters are of type Δ'_i, such that $\Delta'_i <: \Delta_i$. An invocation of such a method would then need to adjust the input arguments and the result using the corresponding projections and injections. For example, invocation of $Meth_i$ (**val** $x'_i : \Gamma'_i$, **res** $y'_i : \Delta'_i$) specified by M'_i in class D on an object $d : \tau(D)$ with input argument $g_i : \Gamma_i$ and result argument $d_i : \Delta_i$ is modelled by

$$\begin{array}{lll} d.Meth_i\,(g_i, d_i) & \widehat{=} & \textbf{begin var } self, x'_i, y'_i \cdot self = d \wedge x'_i = \iota_{\Gamma_i}\ g_i;\\ & & \quad M'_i \times \textbf{skip}; d, d_i := self, \iota_{\Delta'_i}\ y'_i;\\ & & \textbf{end}\end{array}$$

Here the value parameter x'_i is initialised with the value of the input argument injected into the type Γ'_i. Similarly, the value of the method result y'_i, being of type Δ'_i, cannot be directly assigned to the variable $d_i : \Delta_i$ and is injected into the type Δ'_i using the corresponding injection function.

Subtype polymorphism extends in a natural way to pointer types. A sum of pointer types **pointer to** $\tau(C)^+$ is defined to be such that its base types are **pointer to** $\tau(C)$ and all the pointer types to subclasses of C. A variable of such a polymorphic pointer type cannot be instantiated using **new**, because the latter is defined to generate

[3] For a more extensive explanation of covariance and contravariance see, e.g., [AbC96].

```
$Bag$ = class
  var $b$ : bag of $Char$

  $Bag()$ = enter var $b \cdot b = \llbracket\rrbracket$,

  $Add$(val $c : Char$) = $b := b \cup \llbracket c \rrbracket$,

  $AddAll$(val $nb$ : bag of $Char$) =
    while $nb \neq \llbracket\rrbracket$ do
      begin var $c \cdot c \in nb$;
        $self.Add(c); nb := nb - \llbracket c \rrbracket$;
      end
    od
end
```

```
$CountingBag$ = subclass of $Bag$
  var $b$ : bag of $Char, n : Nat$

  $CountingBag()$ =
    enter var $b, n \cdot b = \llbracket\rrbracket \wedge n = 0$,

  $Add$(val $c : Char$) =
    $n := n + 1; super.Add(c)$
end
```

Fig. 3. Example of subclassing with dynamic binding of self-referential methods.

a new index in some array of class instances associated with a base pointer type. A polymorphic pointer variable can, however, be assigned a value of an existing index to one of the arrays $heap_C, heap_{C_1}, \ldots, heap_{C_n}$, which keep instances of C and instances of its subclasses $C_1, \ldots, C_n$. Before assignment, this pointer value should be injected into the corresponding sum type.

Dynamic binding of self-referential methods can occur only when a subclass inherits all attributes of its superclass without overriding them. Essentially, a super-call to a method self-calling other methods of the same class resolves the latter with the definitions of the self-called methods in the class which originated the super-call.

Suppose that class C' inherits all attributes of its superclass C and has some new attributes, so that the first projection of $self : \Sigma \times \Sigma'$ in C' is equal to $self : \Sigma$ in C. The general behaviour of a self-referential method $Meth_j(\mathbf{val}\ x_j : \Gamma_j, \mathbf{res}\ y_j : \Delta_j)$ in C can be described by

$$(\mathbf{var}\ self, x_j, y_j \cdot \\ \quad \mathbf{begin\ var}\ l \cdot b; \mathbf{do}\ \Diamond_{i=1}^{n} q_i :: self.Meth_i(g_i, d_i); L_i\ \mathbf{od}; \mathbf{end})$$

The input and result arguments g_i, d_i are among the variables $self, x_j, y_j$ and l.

Let the behaviour of this method be given in C' by $super.Meth_j(x_j, y_j)$. Then the super-call is defined to invoke the self-called methods on the current $self$ object:

$$\begin{aligned}(\mathbf{var}\ self, x_j, y_j \cdot super.Meth_j(x_j, y_j)) \ \widehat{=}\ & \\ (\mathbf{var}\ self, x_j, y_j \cdot \mathbf{begin}\ \langle\rho\rangle\ &(\mathbf{var}\ l, self, x_j, y_j \cdot b); \\ &\mathbf{do}\ \Diamond_{i=1}^{n} \langle\rho\rangle\ q_i :: self.Meth_i(g_i, d_i); L_i \downarrow |\rho|\ \mathbf{od}; \\ \mathbf{end})&\end{aligned}$$

where $\rho = (\lambda x, (y, y'), z \cdot x, y, z)$ is the projection function removing the extra attribute of $self$. Applying the functional update $\langle\rho\rangle$ to the predicates $(\mathbf{var}\ l, self, x_j, y_j \cdot b)$ and $q_1, \ldots, q_n$, and wrapping the statements $L_1, \ldots, L_n$ in the relation $|\rho|$, coerces them to operate on the extended state space $\Lambda \times (\Sigma \times \Sigma') \times \Gamma_j \times \Delta_j$. As such, this is a technicality not changing the meaning of the corresponding statements. Self-calls to $Meth_1, \ldots, Meth_n$ are resolved with the definitions of these methods given in C'.

As an example consider specifications of Bag and $CountingBag$ presented in Fig. 3. The subclass $CountingBag$ inherits the only attribute of its superclass Bag, representing a bag of characters, and adds a counter of bag elements. The method Add overrides the corresponding method of the superclass by incrementing the counter and then super-calling Add of Bag.

The method $AddAll$ joins two bags by self-calling Add. The self-call in the definition of $AddAll$ in Bag is resolved by substituting the body of Add as defined in Bag:

```
$Bag :: AddAll$(val $nb$ : bag of $Char$) = while $nb \neq \llbracket\rrbracket$ do
                                              begin var $c \cdot c \in nb$;
                                                $b := b \cup \llbracket c \rrbracket$;
                                                $nb := nb - \llbracket c \rrbracket$;
                                              end
                                            od
```

The definition of the method *AddAll* in *CountingBag* exemplifies dynamic binding of self-referential methods. First of all, inheriting this method from *Bag* corresponds to super-calling it:

$$\textit{CountingBag} :: \textit{AddAll}(\textbf{val } nb : \textbf{bag of } \textit{Char}) \;=\; \textit{super.AddAll}(nb)$$

According to the definition of a super-called method involving self-calls, we then have that for $\textit{self} = (b, n)$, $\textit{super.AddAll}(nb)$ is equal to

```
(var $(b,n), nb \cdot$ while $\langle\rho\rangle$ (var $b, nb \cdot nb \neq [\![\,]\!]$) do
                begin $\langle\rho\rangle$ (var $c, b, nb \cdot c \in nb$);
                    $\textit{self.Add}(c)$; (var $c, b, nb \cdot nb := nb - [\![c]\!]) \downarrow |\,\rho\,|$;
                end
            od)
```

which, using the definitions of $\rho, \downarrow$, and functional update, is equal to

```
(var $(b,n), nb \cdot$ while (var $(b,n), nb \cdot nb \neq [\![\,]\!]$) do
                begin (var $c, (b,n), nb \cdot c \in nb$);
                    $\textit{self.Add}(c)$; (var $c, (b,n), nb \cdot nb := nb - [\![c]\!]$);
                end
            od)
```

The self-call, being on $\textit{self} = (b, n)$, is resolved with the definition of *Add* in *CountingBag*.

4. Class Refinement

When a subclass overrides some methods of its superclass, there are no guarantees that its instances will deliver the same or refined behaviour as the instances of the superclass. Unrestricted method overriding in a subclass can lead to arbitrary behaviour of its instances. When used in a superclass context, such subclass instances can invalidate their clients. To avoid such problems, we would like to ensure that whenever C' is subclassed from C, clients using objects instantiated by C can safely use objects instantiated by C' instead. First we consider class refinement between two classes having the same number of methods and then extend the definition to account for additional methods defined in a subclass.

4.1. Class Refinement Without New Methods

Suppose classes C and C' specify interfaces

$$\begin{array}{l} \textit{Meth}_1(\textbf{val} \;: \Gamma_1, \textbf{res} \;: \Delta_1), \ldots, \textit{Meth}_n(\textbf{val} \;: \Gamma_n, \textbf{res} \;: \Delta_n) \text{ and} \\ \textit{Meth}_1(\textbf{val} \;: \Gamma'_1, \textbf{res} \;: \Delta'_1), \ldots, \textit{Meth}_n(\textbf{val} \;: \Gamma'_n, \textbf{res} \;: \Delta'_n) \end{array}$$

respectively. Let C and C' be modelled by tuples $(K, M_1, \ldots, M_n)$ and $(K', M'_1, \ldots, M'_n)$, where $K : \Gamma_0 \mapsto \Sigma \times \Gamma_0$ and $K' : \Gamma'_0 \mapsto \Sigma' \times \Gamma'_0$ are the class constructors, and all $M_i : \Xi(\Sigma \times \Gamma_i \times \Delta_i)$ and $M'_i : \Xi(\Sigma' \times \Gamma'_i \times \Delta'_i)$ are the corresponding methods. The value parameter types of the constructors and the methods in C' are either the same or contravariant, so that $\Gamma_0 <: \Gamma'_0$ and $\Gamma_i <: \Gamma'_i$, and the result parameter types of its methods are either the same or covariant, $\Delta'_i <: \Delta_i$.

Let $R : \Sigma' \leftrightarrow \Sigma$ be a relation coercing attribute types of C' to those of C, so that R is of the form $(\lambda c \cdot \{a \,|\, R\; c\; a\})$. The refinement of class constructors K and K' with respect to R is defined as follows:

$$K \sqsubseteq_R K' \;\;\widehat{=}\;\; \{\pi_{\Gamma_0}\}; K \;\sqsubseteq\; K'; \{R \times \pi_{\Gamma_0}\} \qquad (\textit{constructor refinement})$$

where π_{Γ_0} is the projection relation coercing Γ'_0 to Γ_0. The commuting diagram in Fig. 4 (a) illustrates constructor refinement.

The refinement of all corresponding methods M_i and M'_i with respect to the relation R is defined by

$$M_i \sqsubseteq_R M'_i \;\;\widehat{=}\;\; M_i \downarrow (R \times \pi_{\Gamma_i} \times |\iota_{\Delta'_i}|) \;\sqsubseteq\; M'_i \qquad (\textit{method refinement})$$

where $\pi_{\Gamma_i} : \Gamma'_i \leftrightarrow \Gamma_i$ projects the corresponding value parameters, and $|\iota_{\Delta'_i}| : \Delta'_i \leftrightarrow \Delta_i$ injects the corresponding result parameters. Obviously, when $\Gamma_i = \Gamma'_i$, the projection relation π_{Γ_i} is the identity relation *Id*. The same holds when $\Delta_i = \Delta'_i$, namely, $|\iota_{\Delta'_i}| = \textit{Id}$. The commuting diagram in Fig. 4 (b) illustrates method refinement.

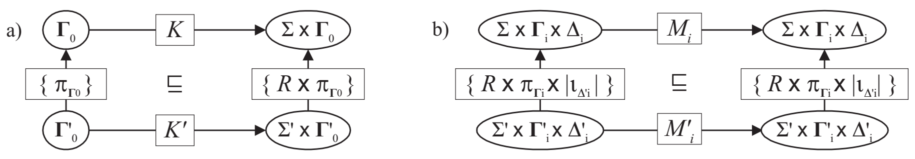
a)
Γ_0
K
$\Sigma \times \Gamma_0$
$\{\pi_{\Gamma_0}\}$
$\sqsubseteq$
$\{R \times \pi_{\Gamma_0}\}$
Γ'_0
K'
$\Sigma' \times \Gamma'_0$
b)
$\Sigma \times \Gamma_i \times \Delta_i$
M_i
$\Sigma \times \Gamma_i \times \Delta_i$
$\{R \times \pi_{\Gamma_i} \times |\iota_{\Delta'_i}|\}$
$\sqsubseteq$
$\{R \times \pi_{\Gamma_i} \times |\iota_{\Delta'_i}|\}$
$\Sigma' \times \Gamma'_i \times \Delta'_i$
M'_i
$\Sigma' \times \Gamma'_i \times \Delta'_i$

Fig. 4. Constructor refinement a) and method refinement b).

Definition 1. For classes $C = (K, M_1, \ldots, M_n)$ and $C' = (K', M'_1, \ldots, M'_n)$, class refinement $C \sqsubseteq C'$ is defined as follows:

$$C \sqsubseteq C' \;\widehat{=}\; (\exists R \cdot K \sqsubseteq_R K' \wedge (\forall i \mid 1 \leqslant i \leqslant n \cdot M_i \sqsubseteq_R M'_i))$$

The class refinement relation is reflexive and transitive. This definition of class refinement is also a proof rule allowing us to check for any two given classes whether they are in refinement. However, from this definition alone we cannot make any conclusions about the behaviour of clients using instances of classes that are in refinement. Before presenting a theorem which relates class refinement to object substitutability in clients, let us introduce two useful lemmas.

Lemma 1. Let classes C and C' have constructors $K : \Gamma_0 \mapsto \Sigma \times \Gamma_0$ and $K' : \Gamma'_0 \mapsto \Sigma' \times \Gamma'_0$ with $\Gamma_0 <: \Gamma'_0$. In a global state $u : \Phi$, for a relation $R : \Sigma' \leftrightarrow \Sigma$, a statement $S : \Xi(\Sigma \times \Phi)$, and a constructor input argument $e : \Gamma_0$,

$$K \sqsubseteq_R K' \;\Rightarrow$$
$$\quad \textbf{create var}\; c.C(e); S; \textbf{end} \;\sqsubseteq\; \textbf{create var}\; c'.C'(e); S\downarrow(R \times Id); \textbf{end}$$

Lemma 2. Let classes C and C' have methods $M_i : \Xi(\Sigma \times \Gamma_i \times \Delta_i)$ and $M'_i : \Xi(\Sigma' \times \Gamma'_i \times \Delta'_i)$ with $\Gamma_i <: \Gamma'_i$ and $\Delta'_i <: \Delta_i$. In a global state $u : \Phi$ including a variable $d_i : \Delta_i$, for a relation $R : \Sigma' \leftrightarrow \Sigma$ and an input argument $g_i : \Gamma_i$,

$$M_i \sqsubseteq_R M'_i \;\Rightarrow$$
$$\quad (\textbf{var}\; c, u \cdot c.Meth_i(g_i, d_i))\downarrow(R \times Id) \;\sqsubseteq\; (\textbf{var}\; c', u \cdot c'.Meth_i(g_i, d_i))$$

The following theorem proves that clients using objects instantiated by some class are refined when using objects instantiated by its refinement.

Theorem 1. For classes C and C', a program $\mathscr{K}$ expressible as an iterative choice of invocations of C methods, and a constructor input argument $e : \Gamma_0$,

$$C \sqsubseteq C' \;\Rightarrow$$
$$\quad \textbf{create var}\; c.C(e); \mathscr{K}\,[c]; \textbf{end} \sqsubseteq \textbf{create var}\; c'.C'(e); \mathscr{K}\,[c']; \textbf{end}$$

Proofs of Lemma 1, Lemma 2, and Theorem 1 can be found in the full version of this paper [BMW00]. Declaring one class as a subclass of another raises the proof obligation that the class refinement relation holds between these classes. This is, in a way, a semantic constraint that we impose on subclassing to ensure that behaviour of subclasses conforms to the behaviour of their superclasses and that subclass instances can be substituted for superclass instances in all clients.

For lack of space, we cannot present here an example of proving class refinement in practice. The interested reader can refer to [Mik00] presenting a proof of class refinement for specifications of *Collection* and *List* interfaces of the Java Collections Framework which is a part of the standard JDK2.0.

4.2. The Problem Introduced by New Methods

As was pointed out in [LiW94] the effects of new methods become visible in the presence of subsumption (subtype aliasing) as well as in the general computational environment that allows sharing of objects by multiple users. For example, when a client is working with an object c' of a subclass C' of C, it may freely call new methods defined in C'. Other clients of c' considering it as an instance of the polymorphic type $\tau(C)^+$ can only anticipate changes to c' specified by the methods of the class C. New methods specifying some "unexpected behaviour" could take c' to (what for C is) an unreachable state, and clients of this object considering it from the superclass perspective would be damaged.

Let us consider an example illustrating this problem. Suppose that a class *Counter* introduces methods *Val* and *Inc2* which, respectively, return the value of the counter and increment the counter by two. A subclass *Counter'* inherits these methods and, in addition, defines a method *Inc1* incrementing the counter by one:

```
Counter = class                                  Counter' = subclass of Counter
    var n : Nat                                      var n : Nat

    Counter () = enter var n · n = 0,                Counter' () = enter var n · n = 0,
    Inc2 () = n := n + 2,                            Inc1 () = n := n + 1
    Val (res r : Nat) = r := n                   end
end
```

The implicit (or the strongest) invariant established by the class constructor and preserved by the methods of *Counter* states that the predicate *Even* holds of all states reachable by objects instantiated by *Counter*. The new method *Inc1* defined in the subclass breaks this invariant. A client $\mathcal{K}$ of a polymorphic object $c : \tau(Counter)^{+}$ might assume that this invariant holds of all states reachable by c and execute some statement S relying on the invariant:

$$\mathcal{K}[c] \;=\; \textbf{if } (Even\ c.Val()) \textbf{ then } S \textbf{ else abort fi}$$

When c is instantiated by *Counter*, other clients in the environment of $\mathcal{K}$ will only be able to call the methods defined in the class *Counter* which preserve the strongest invariant. When operating in such an environment, $\mathcal{K}$ will always execute S. However, when c is instantiated by *Counter'*, $\mathcal{K}$ may also work in the environment where other clients of c know about its origin and may call the method *Inc1* in addition to the methods *Inc2* and *Val*. Suppose, for example, that there is a client $\mathcal{K}'$ which sees the class *Counter'*, with the new method *Inc1* (), and tries to increase the counter by as little as possible:

$$\mathcal{K}'[c] \;=\; \textbf{if } (c \textbf{ is } \tau(Counter')) \textbf{ then } c.Inc1() \textbf{ else } c.Inc2() \textbf{ fi}$$

If objects c and c' are now instantiated by *Counter* and *Counter'* respectively and initialised to zero, executing $\mathcal{K}'[c];\mathcal{K}[c]$ equals executing S, whereas $\mathcal{K}'[c'];\mathcal{K}[c']$ aborts because the strongest invariant is broken by $\mathcal{K}'$.

To avoid this and similar problems, we want to ensure that invocation of a new method does not result in any unexpected behaviour or, in other words, that the new method *preserves the strongest invariant of its superclass.* Let us formally analyse this consistency property and the requirements that can be imposed on new methods to enforce this property.

4.3. Ensuring New Method Consistency

Let us first define the notion of the strongest class invariant. As suggested by its name, the strongest class invariant is the least state predicate established by the class constructor and preserved by all its methods.

Definition 2. For a class $C = (K, M_1, \ldots, M_n)$, a state predicate I is the strongest class invariant if it is the invariant of C and of all invariants of C it is the least one:

$$\begin{array}{l} Inv\,(C,I) \;\;\widehat{=} \\ \quad true\ \{\!|\, K\, |\!\}\ I\ \wedge\ (\forall i \mid 1 \leqslant i \leqslant n \cdot I\ \{\!|\, M_i\, |\!\}\ I) \wedge \\ \quad (\forall J \cdot true\ \{\!|\, K\, |\!\}\ J\ \wedge\ (\forall i \mid 1 \leqslant i \leqslant n \cdot J\ \{\!|\, M_i\, |\!\}\ J) \Rightarrow I \subseteq J) \end{array}$$

Suppose now that a class $C = (K, M_1, \ldots, M_n)$ specifies the interface

$$Meth_1\,(\textbf{val}\ :\Gamma_1, \textbf{res}\ :\Delta_1), \ldots, Meth_n\,(\textbf{val}\ :\Gamma_n, \textbf{res}\ :\Delta_n)$$

and a class $C' = (K', M'_1, \ldots, M'_n, N_1, \ldots, N_p)$ specifies the interface

$$\begin{array}{l} Meth_1\,(\textbf{val}\ :\Gamma_1, \textbf{res}\ :\Delta_1), \ldots, Meth_n\,(\textbf{val}\ :\Gamma_n, \textbf{res}\ :\Delta_n), \\ NMeth_1\,(\textbf{val}\ :\Phi_1, \textbf{res}\ :\Psi_1), \ldots, NMeth_p\,(\textbf{val}\ :\Phi_p, \textbf{res}\ :\Psi_p) \end{array}$$

For simplicity, we assume that methods $Meth_1, \ldots, Meth_n$ in C' have the same types of value and result parameters as the corresponding methods in C. The case when the value parameter types are contravariant and the result parameter types are covariant is treated similarly. We can express the meaning of an invariant I of C on the attributes of C' as $\{R\}\ I$, where R is a relation coercing the attributes of C' to those of C. To guarantee that a new method N_j of C' preserves the strongest class invariant of C, we then need to prove the correctness assertion $\{R\}\ I\ \{\!|\ N_j\ |\!\}\ \{R\}\ I$ for I such that $Inv\ (C,I)$. By satisfying this correctness assertion, the new method of C' preserves the set of reachable states of C. In general, preserving a coerced invariant $\{R\}\ I$ by a statement $S' : \Xi(\Sigma')$ is the same as preserving the invariant I by the statement S' coerced to operate on the state space Σ, as expressed in the following lemma.

Lemma 3. For a statement $S' : \Xi(\Sigma')$, a relation $R : \Sigma' \leftrightarrow \Sigma$, and a state predicate $I : \mathscr{P}\Sigma$, we have

$$\{R\}\ I\ \{\!|\ S'\ |\!\}\ \{R\}\ I \ = \ I\ \{\!|\ S' \uparrow R\ |\!\}\ I$$

A proof of this lemma can be found in [BMW00].

Class refinement between a class C and a class C' introducing new methods is given as an extension of Def. 1 requiring that every new method of C' preserves the strongest class invariant of C.

Definition 3. For a class $C = (K, M_1, \ldots, M_n)$ and a class $C' = (K', M'_1, \ldots, M'_n, N_1, \ldots, N_p)$, class refinement $C \sqsubseteq C'$ is defined as follows:

$$\begin{aligned} C \sqsubseteq C' \ \widehat{=}\ & (\exists R \cdot K \sqsubseteq_R K' \wedge (\forall i \mid 1 \leqslant i \leqslant n \cdot M_i \sqsubseteq_R M'_i) \wedge \\ & (\forall I \cdot Inv\ (C,I) \Rightarrow (\forall j \mid 1 \leqslant j \leqslant p \cdot \{R\}\ I\ \{\!|\ N_j\ |\!\}\ \{R\}\ I))) \end{aligned}$$

As one can expect, Theorem 1, relating class refinement to object substitutability in clients, holds for the extended definition of class refinement as well. Unfortunately, verifying correctness assertions for new methods can be difficult in practice, because the strongest invariant of a superclass cannot always be easily calculated from its specification, e.g., in the case of recursive method invocations. When such verification is infeasible, we could instead verify that new methods satisfy certain restrictions such that the correctness assertions hold automatically. Intuitively, a new method preserves the strongest invariant of the superclass if it does not modify attributes at all, or if it modifies them as the old methods could have done. More precisely, a new method preserves the strongest invariant of the superclass in the following cases:

- the new method is an observer, i.e. a non-modifying method
- the subclass adds new attributes without overriding the original attributes of the superclass, overridden methods modify the original attributes only via supercalls, and the new method modifies only the new attributes
- the new method is composed of calls to old methods
- the new method is a refinement of (a combination of) old methods

Note that in the last case the new method can either data refine the old method definitions as given in the superclass, or refine the old method definitions as given in the subclass, or be a refinement of any combination of these.

Formally, weak iteration of a demonic choice of statements $S_1, \ldots, S_n$, namely $(\sqcap_{i=1}^{n} S_i)^*$, describes all possible combinations of these statements. Any combination of statements refines this statement, e.g., $(\sqcap_{i=1}^{3} S_i)^* \sqsubseteq S_1; S_3; S_2; S_1$. To be consistent, a new method should data refine an arbitrary combination of old methods prefixed by enabledness guards and intermixed with arbitrary statements. We require that the arbitrary statements do not update the attributes and necessarily terminate. A statement S is guaranteed to terminate if it can establish any postcondition from any initial state, i.e. $true \ = \ S\ true$.

As old methods and new methods operate on different state spaces, we first have to adjust them to operate on the common state space. Recall that methods M_i of C operate on $\Sigma \times \Gamma_i \times \Delta_i$, while methods M'_i of C' operate on $\Sigma' \times \Gamma_i \times \Delta_i$ and new methods N_j on $\Sigma' \times \Phi_j \times \Psi_j$. We can construct a common state space Π including all value and result parameter types of all methods in C' so that

$$\Pi = \Gamma_1 \times \Delta_1 \times \ldots \times \Gamma_n \times \Delta_n \times \Phi_1 \times \Psi_1 \times \ldots \times \Phi_p \times \Psi_p$$

Then a projection function $\xi_i : \Pi \rightarrow \Gamma_i \times \Delta_i$, for $i = 1..n$, will give us the types of value and result parameters of method M'_i. Similarly, a projection function $\xi_{n+j} : \Pi \rightarrow \Phi_j \times \Psi_j$, for $j = 1..p$, will give us the types of value and result parameters of method N_j. We can always coerce M'_i to operate on the state space $\Sigma' \times \Pi$ using the corresponding projection function.

As methods M_i of C have to operate on the attributes of C' rather than C, they have to be appropriately coerced using the abstraction relation $R : \Sigma' \leftrightarrow \Sigma$. The resulting statement $M_i \downarrow R$, being of type $\Xi(\Sigma' \times \Gamma_i \times \Delta_i)$, still has to be coerced to operate on the common state space $\Sigma' \times \Pi$, using the corresponding projection function.

Putting everything together, we can now define consistency of new methods as follows.

Definition 4. For classes $C = (K, M_1, \ldots, M_n)$ and $C' = (K', M'_1, \ldots, M'_n, N_1, \ldots, N_p)$, some guards q_i, and some terminating statements K_i skipping on the attributes of C', consistency of a new method N_j, for $j = 1..p$, with respect to C and an abstraction relation $R : \Sigma' \leftrightarrow \Sigma$ is defined as follows:

$$\begin{array}{l} \mathit{Consistent}\ (N_j, C, R)\ \mathrel{\widehat{=}} \\ \quad \textbf{begin var}\ l \cdot b; (\sqcap_{i=1}^{n}\ [q_i]; (\textbf{skip} \times M_i \downarrow R) \downarrow |\rho_i|; K_i)^*; \textbf{end}\ \sqsubseteq\ N_j \end{array}$$

Here the local block variables l introduce the value and result parameters of all methods $M'_1, \ldots, M'_n$ and all new methods except N_j, whose value and result parameters are already present in the state. Effectively, the state space inside the block is $\Pi' \times \Sigma' \times \Phi_j \times \Psi_j$, where Π' is the same as Π with $\Phi_j \times \Psi_j$ projected away. The statement **skip** $\times\, M_i \downarrow R$ operates on the state space $\Pi'' \times \Sigma' \times \Gamma_i \times \Delta_i$, where Π'' is the same as Π with $\Gamma_i \times \Delta_i$ projected away. To coerce this statement to operate on the state space of the block, which has the same state components but in a slightly different order (unless M_i and N_j happen to have value and result parameters of the same types), we wrap it in the function $\rho_i : \Pi' \times \Sigma' \times \Phi_j \times \Psi_j \to \Pi'' \times \Sigma' \times \Gamma_i \times \Delta_i$. Note that wrappings in the state-reassociating functions ρ_i are just technicalities not changing the meaning of the corresponding statements.

Definition 4 allows a new method to be an arbitrary non-modifying method refining **skip**, since $(\sqcap_{i=1}^{n}\ [q_i]; (\textbf{skip} \times M_i \downarrow R) \downarrow |\rho_i|; K_i)^* \sqsubseteq \textbf{skip}$. No less important, it follows from Def. 4 that a new method N_j is also consistent if it is composed of calls to overriding methods intermixed with arbitrary statements or refines an arbitrary composition of such calls:

Corollary 1. For classes $C = (K, M_1, \ldots, M_n)$ and $C' = (K', M'_1, \ldots, M'_n, N_1, \ldots, N_p)$, some guards q_i, and some terminating statements K_i skipping on the attributes of C',

$$\begin{array}{l} (\forall i \mid 1 \leqslant i \leqslant n \cdot M_i \sqsubseteq_R M'_i)\ \wedge \\ \textbf{begin var}\ l \cdot b; (\sqcap_{i=1}^{n}\ [q_i]; (\textbf{skip} \times M'_i) \downarrow |\rho_i|; K_i)^*; \textbf{end}\ \sqsubseteq\ N_j\ \Rightarrow \\ \quad \mathit{Consistent}\ (N_j, C, R) \end{array}$$

If all new methods in a class C' are consistent, the constructor of C is refined by the constructor of C' and all old methods of C are refined by the corresponding methods of C', then class refinement between C and C' is guaranteed to hold, as proved by the following theorem.

Theorem 2. For classes $C = (K, M_1, \ldots, M_n)$ and $C' = (K', M'_1, \ldots, M'_n, N_1, \ldots, N_p)$,

$$\begin{array}{l} (\exists R \cdot K \sqsubseteq_R K'\ \wedge\ (\forall i \mid 1 \leqslant i \leqslant n \cdot M_i \sqsubseteq_R M'_i) \wedge \\ \quad (\forall j \mid 1 \leqslant j \leqslant p \cdot \mathit{Consistent}\ (N_j, C, R)))\ \Rightarrow\ C \sqsubseteq C' \end{array}$$

A proof of this theorem can be found in [BMW00].

5. Conclusions and Related Work

This work is based on [MiS97], but concentrates on class refinement and its relation to object substitutability. One of the main contributions of the present paper is in modelling clients of class instances by an iterative choice of method invocations. In our opinion, polymorphic substitutability of objects in clients is central to the object-oriented programming style, and, in this respect, the ability to reason about the behaviour of object clients, and not only objects, is very important. Our model allows us to reason formally about the relationship between refinement on classes and substitutability of class instances in clients. We prove that substituting instances of a refined class for instances of the original class is refinement for the clients.

5.1. Related Work in Formalisation of Object-Oriented Concepts

Related work in formalisation of object-oriented concepts includes [CoP89, Nau94, Nau99, Sek96, AbL97]. William Cook and Jens Palsberg in [CoP89] give a denotational semantics of inheritance and prove its correctness with respect to an operational "method lookup" semantics. They model dynamic binding of self-referential methods by representing classes as functions of self-called methods and constructing subclasses using modifying wrappers. There are only functional methods in their model, whereas we consider the methods modifying object state as well.

Martín Abadi and Rustan Leino in [AbL97] develop a logic of object-oriented programs in the style of Hoare [Hoa69], prove its soundness and discuss completeness issues. Rather than building a new logic, we extend a logic for reasoning about imperative programs (the refinement calculus) with definitions of classes, subclassing, subtyping, and class refinement. Our extension is conservative in the sense that it does not extend the set of theorems over the original constants in the underlying logic. Being itself a conservative extension of higher-order logic, the refinement calculus has the syntax of higher-order logic, with some syntactic sugaring, and the simple set-theoretic semantics of higher-order logic. As the refinement calculus identifies program statements with the monotonic predicate transformers that they determine, it does not emphasise the distinction between syntax, semantics, and proof theory that is traditional in programming logics [BvW98].

Semantics of an imperative Oberon-like programming language with similar specification constructs as here, also based on predicate transformers, is defined by David Naumann in [Nau94] and recently extended to include more interesting and useful features in [Nau99]. The language does not include classes or visibility controls, references or aliasing, but goes beyond our work in supporting procedure variables. Also, procedures (methods) may have global variables in [Nau99]. We feel that permitting class methods access and modify global variables is discordant with the object-oriented paradigm, for methodological reasons, and don't model this feature.

Emil Sekerinski [Sek96] defines a rich object-oriented programming and specification notation by using a type system with subtyping and type parameters, and also using predicate transformers. In both approaches, subtyping is based on extensions of record types. Here we use sum types instead, as suggested by Ralph Back and Michael Butler in [BaB95] and further elaborated in [MiS97]. One motivation for moving to sum types is to avoid complications in the typing and the logic when reasoning about record types: the simply typed lambda calculus as the formal basis is sufficient for our purposes. Also, to allow objects of a subclass to have different (private) attributes from those of the superclass, hiding by existential types was used in [Sek96]. It turned out that, when reasoning about method calls, this leads to complications which are not present when using the model of sum types. Leonid Mikhajlov and Emil Sekerinski in [MiS98] give semantics to object-oriented constructs in the refinement calculus, modelling dynamic binding of self-referential methods following [CoP89] but permitting state-modifying methods as we do here. As their formalisation is tailored for studying a particular problem, namely the fragile base class problem, they consider a limited set of object-oriented constructs and mechanisms.

The detailed elaboration of our formalisation, especially the fact that we define all object-oriented constructs and mechanisms on the semantic level, within the logic, rather than by syntactic definitions, opens the possibility of mechanised reasoning and mechanical verification. An interesting recent work by Bart Jacobs et al. in [JBH98] reports a work in progress on building a front-end tool for translating Java classes to higher-order logic in PVS [ORS92]. The authors state that "current work involves incorporation of Hoare logic [Hoa69], via appropriate definitions and rules in PVS", and present in [JBH98] a description of the tool "directly based on definitions". We develop a theoretic foundation for reasoning about object-oriented programs based on the logical framework for reasoning about imperative programs. A tool supporting verification of correctness and refinement of imperative programs and known as the Refinement Calculator [LRW95] already exists and extending it to handling object-oriented programs based on the formalisation presented here appears to be only natural.

5.2. Related Work on Behavioural Compatibility of Objects

The general idea behind our approach and the research direction known as behavioural subtyping is essentially the same – to develop a specification and verification methodology for reasoning about correctness of object-oriented programs. Our work has been to a great extent inspired by works of Pierre America, Barbara Liskov, Jeannette Wing, Gary Leavens, and others [Ame87, Ame91, LiW94, LeW90, LeW95, DhL96]. However, our approach differs in a number of ways. First of all, as was already mentioned in the introduction, we consider it essential to separate decidable syntactic properties of interface conformance or subtyping from undecidable but provable properties of behavioural conformance or refinement. We use classes to express (at different abstraction levels) the behaviour of objects and class refinement to express behavioural conformance. Here we for simplicity consider systems where subclassing forms a basis for subtype polymorphism. However, our model of classes, subclassing, and subtype polymorphism as well as the definition of class refinement can be

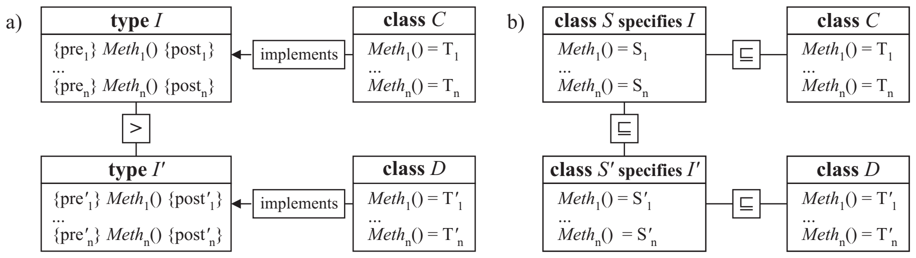

Fig. 5. Behavioural subtyping (a) and class refinement (b) in the case of separate interface and implementation inheritance hierarchies.

used to reason about the meaning of programs using separate subclassing and interface inheritance hierarchies. By associating a specification class with every interface type, we can reason about the behaviour of objects having this interface. All classes claiming to implement a certain interface must refine its specification class. Subclassing in this layout does not, in general, require establishing class refinement between the superclass and the subclass.

When used in the context of separate subclassing and subtyping hierarchies, class refinement is very similar to behavioural subtyping. Consider a graphical representation of the corresponding settings in Fig. 5. In both cases I and I' are certain interfaces (types) such that I' is a syntactic subtype of I. In the case of behavioural subtyping in Fig. 5 (a) the behaviour of methods is specified in terms of pre- and postconditions. To verify that I' is a behavioural subtype of I, written $I' < I$, America, Liskov, and Wing require proving that every precondition pre_i is stronger than the corresponding pre'_i and every postcondition $post_i$ is weaker than the corresponding $post'_i$, while Dhara and Leavens in [DhL96] weaken the requirement for postconditions. In addition to proving behavioural subtyping, one must also verify that the classes C and D claiming to implement the types I and I' respectively really do so. America in [Ame91] proposes a rigorous verification method that can be used for this purpose. For verifying, e.g., that C implements I, he uses a representation function mapping concrete states of C to the set of abstract states associated with I as well as a representation invariant constraining the values of attributes in C, and requires proving that every method T_i of C preserves the representation invariant and establishes $post_i$ coerced to the state space of C when pre_i also coerced to the state space of C holds. Since in [Ame91] and other works on behavioural subtyping no formal semantics is given to implementation constructs and mechanisms, such as, e.g., super-calls or dynamic binding, this verification can only be done semi-formally.

Consider now the diagram (b) of Fig. 5 illustrating class refinement. First of all, we can reason about specification classes S and S' and implementation classes C and D in a uniform manner, and the behavioural conformance between the participating classes is the class refinement. Since class refinement is transitive, we get directly that D, implementing I' by refining its specification S', also refines the specification S of I.

Class refinement can be used to verify correctness even if D happens to be a subclass of C. Dynamic binding of self-referential methods, which becomes possible in this case, can be resolved as described in Sec. 3.6, and then we can prove that, e.g., $S' \sqsubseteq D$ using the definition of class refinement. With behavioural subtyping, however, it is not clear how one can prove that a method satisfies certain pre- and postconditions in the presence of dynamic binding of self-referential methods.

When used for reasoning about systems with unified subclassing and subtyping, our approach eliminates a significant amount of proof obligations as compared to behavioural subtyping. We do not need to prove separately that a class and its subclass implement the corresponding type and its behavioural subtype, all that needs to be proved is class refinement between the subclass and the superclass.

Researchers working in the area of behavioural subtyping, e.g., America in [Ame91], maintain that specifications in terms of pre- and postconditions are more abstract and easier to understand than those in a more operational style, capturing method invocation order. We feel that the essence of object-oriented programs is in invoking methods on objects, and, as our *TextDoc-View* example shows, it might be necessary to specify explicitly that a certain method calls other methods. When reasoning about correctness, it is often necessary to know the method invocation order, which is more difficult to specify in terms of pre- and postconditions. Therefore, we consider it essential for a specification language to support both declarative and operational specification styles, permitting abstract specifications when it is desirable to abstract away from implementation details and also permitting capturing method invocation order when it is essential. Similar ideas are supported by Richard Helm et al. in [HHG90]. They include method calls in abstract specifications of contracts to express behavioural dependencies between co-operating objects. Martin Büchi and Wolfgang Weck in [BüW97] also advocate a specification language combining specification statements with method calls.

Mark Utting in his PhD thesis [Utt92] extends the refinement calculus to support a variety of object-oriented programming styles. One of the main contributions of [Utt92] is a formal definition in the refinement calculus of modular reasoning advocated by Leavens in [LeW90]. It is assumed that all objects are ordered by a substitution relation $\leqslant$ which must be a preorder but otherwise is unrestricted. An object-oriented system is defined to support modular reasoning if methods of an object a, such that $a \leqslant b$, are refined by the corresponding methods of b. Clearly our methodology of object-oriented system development supports modular reasoning, because, if the substitution ordering is chosen so that $a \leqslant b$ whenever the class of a is refined by the class of b, then the corresponding methods are in refinement. Our definition of class refinement is constructive, meaning that it can be used to formally verify behavioural conformance between given classes. Proving refinement between classes guarantees correctness of substitutability in all clients of the objects these classes instantiate. Utting's definition of modular reasoning, on the other hand, is non-constructive; to cite Liskov and Wing's description in [LiW94], "it tells you what to look for, but not how to prove that you got it".

As it follows the style of behavioural subtyping, the approach reported in [Utt92] separates implementations and specifications (types) and checks behavioural conformance of types to their supertypes. Data refinement is only allowed between the implementation and a specification of an object, although a way of generalizing data refinement for the (behavioural) subtyping is discussed in the future work section. Utting's approach to formalisation of object-oriented programs differs from ours in several aspects, motivated primarily by the fact that the refinement calculus used as the basis for his object-oriented extensions was formalised within infinitary rather than higher-order logic. In particular, with the state space modelled by a product space as we have here, encapsulation is built-in rather naturally in the model: methods operate only on the instances of the corresponding class and cannot access or modify instances of other classes. In [Utt92] the state is not considered to be a tuple of state components, but rather a function from all variables (including object variables) to all values (including object values) in the program. Methods of all objects operate on the global state and encapsulation is only assumed.

Behavioural dependencies in the presence of subclassing have also been studied in various extensions of Z specification languages, e.g., [LaH92, Cus91], but only between class specifications and not implementations. By having specification constructs as part of the (extended) programming language, we do not have to treat specifications and implementations separately.

Data refinement of modules, abstract data types, and abstract machines as, e.g., in [Hoa72, Mor90, Abr96] forms a basis for class refinement. The latter, however, has special features due to subtype polymorphism and dynamic binding. Our definition of class refinement is based on the method of proving data refinement known as forward data refinement or downward simulation. Although it was shown to be incomplete, this method is most widely used as it is sufficient for most cases in practical program development.

Our treatment of new methods follows that of Liskov and Wing as presented in [LiW94]. They describe two approaches to dealing with new method consistency. The first approach requires that new methods satisfy the explicit class invariant and the history constraint, whereas the second approach forces new methods to preserve the strongest superclass invariant. Here we do not consider explicit class invariants and refer to [Mik99] for a detailed analysis of consistency requirements that must be imposed in the presence of explicit invariants. In this paper we present a formal analysis of the requirements that, when satisfied by new methods, are guaranteed to preserve the strongest superclass invariant. Our definition of new method consistency is more permissive than that of Liskov and Wing. They informally require that "for each extra method an explanation be given of how its behaviour could be effected by just those methods already defined for the supertype". Our definition of consistency permits new methods not only to be composed of calls to existing methods, but also refine an arbitrary combination of the old methods as defined in the subclass or data refine an arbitrary combination of the old methods as defined in the superclass.

Acknowledgments

The authors would like to thank Emil Sekerinski, Leonid Mikhajlov, Michael Butler, and Martin Büchi for valuable comments on this paper.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AbC96] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

[AbL97] Martín Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *Proceedings of TAPSOFT'97*, LNCS 1214, pages 682–696. Springer, April 1997.

[Ame87] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of ECOOP'87*, LNCS 276, pages 234–242, Paris, France, 1987. Springer-Verlag.

[Ame91] Pierre America. Designing an object-oriented programming language with behavioral subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, LNCS 489, pages 60–90, New York, N.Y., 1991. Springer-Verlag.

[Bac89] R. J. R. Back. Changing data representation in the refinement calculus. In *21st Hawaii International Conference on System Sciences.* IEEE, January 1989.

[BaB95] R. J. R. Back and M. J. Butler. Exploring summation and product operators in the refinement calculus. In B. Möller, editor, *Mathematics of Program Construction, 1995*, volume 947. Springer-Verlag, 1995.

[BMW99] R. J. R. Back, Anna Mikhajlova, and Joakim von Wright. Reasoning about interactive systems. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of the World Congress on Formal Methods* (*FM'99*), volume 1709 of *LNCS*, pages 1460–1476. Springer-Verlag, September 1999.

[BMW00] Ralph-Johan Back, Anna Mikhajlova, and Joakim von Wright. Class refinement as semantics of correct object substitutability. Technical Report 333, Turku Centre for Computer Science, February 2000.

[BvW97] R. J. R. Back and J. von Wright. Programs on product spaces. Technical Report 143, Turku Centre for Computer Science, November 1997.

[BvW98] R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, April 1998.

[BvW99] R. J. R. Back and Joakim von Wright. Encoding, decoding and data refinement. Technical Report 236, Turku Centre for Computer Science, March 1999.

[BüW97] Martin Büchi and Wolfgang Weck. A plea for grey-box components. Technical Report 122, Turku Center for Computer Science, Presented at the Workshop on Foundations of Component-Based Systems, Zurich, September 1997.

[CoP89] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings OOPSLA'89*, volume 24, pages 433–443. ACM SIGPLAN notices, October 1989.

[Cus91] Elspeth Cusack. Inheritance in object-oriented Z. In P. America, editor, *Proceedings of ECOOP'91*, LNCS 512, pages 167–179, Geneva, Switzerland, July 15–19, 1991. Springer-Verlag.

[DhL95] Krishna Kishore Dhara and Gary T. Leavens. Weak behavioral subtyping for types with mutable objects. In *Mathematical Foundations of Programming Semantics*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[DhL96] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, pages 258–267, Berlin, Germany, 1996.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GoM93] M. J. C. Gordon and Thomas F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP'90*, ACM SIGPLAN Notices, pages 169–180, October 1990.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–583, 1969.

[Hoa72] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[JBH98] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrick Tews. Reasoning about Java classes (preliminary report). In *Proceedings of OOPSLA'98*, pages 329–340, Vancouver, Canada, October 1998. Association for Computing Machinery.

[LaH92] K. Lano and H. Haughton. Reasoning and refinement in object-oriented specification languages. In O. Lehrmann Madsen, editor, *Proceedings of ECOOP'92*, LNCS 615. Springer-Verlag, 1992.

[LRW95] T. Långbacka, R. Ruksenas, and J. von Wright. TkWinHOL: A tool for window inference in HOL. *Higher Order Logic Theorem Proving and its Applications: 8th International Workshop*, 971:245–260, September 1995.

[LeW90] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In *Proceedings of OOPSLA/ECOOP'90*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223, 1990.

[LiW94] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LeW95] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[Mik99] Anna Mikhajlova. Consistent extension of components in the presence of explicit invariants. In *Technology of Object-Oriented Languages and Systems* (*TOOLS 29*), pages 76–85. IEEE Computer Society Press, June 1999.

[Mik00] Anna Mikhajlova. Combining code with specifications: How to document and verify frameworks. *Special Issue of L'Objet on Formal Methods for Object Systems*, 6(1), 2000.

[Mor90] C. C. Morgan. *Programming from Specifications.* Prentice–Hall, 1990.

[MiS97] Anna Mikhajlova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In *Proceedings of the 4th International Formal Methods Europe Symposium, FME'97*, LNCS 1313, pages 82–101. Springer, 1997.

[MiS98] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *Proceedings of ECOOP'98*, pages 355–382. Springer, July 1998.

[Nau94] D. A. Naumann. Predicate transformer semantics of an Oberon-like language. In Ernst-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 460–480, San Miniato, Italy, 1994.

[Nau99] D. A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtypes. *Science of*

Computer Programming, 1999. To appear. URL: `http://guinness.cs.stevens-tech.edu/~naumann/publications.html`.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Sek96] E. Sekerinski. A type-theoretic basis for an object-oriented refinement calculus. In S. J. Goldsack and S. J. H. Kent, editors, *Formal Methods and Object Technology*. Springer-Verlag, 1996.

[Utt92] Mark Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. PhD thesis, University of New South Wales, Kensington, Australia, 1992.

Appendix

Here we present refinement and correctness rules that will be used in proofs of lemmas and theorems presented in the paper. Proofs of these rules can be found in [BvW98, BvW97, BvW99].

Correctness Rules

(a) $p \;\{\!|\, S_1 ; S_2 \,|\!\}\; q \;=\; (\exists r \cdot p \;\{\!|\, S_1 \,|\!\}\; r \wedge r \;\{\!|\, S_2 \,|\!\}\; q)$
(b) $p \;\{\!|\, (\sqcap i \in I \cdot S_i) \,|\!\}\; q \;=\; (\forall i \in I \cdot p \;\{\!|\, S_i \,|\!\}\; q)$
(c) $p \;\{\!|\, [r] \,|\!\}\; q \;=\; p \cap r \subseteq q$
(d) $r \;\{\!|\, S \,|\!\}\; r \;\Rightarrow\; r \;\{\!|\, S^* \,|\!\}\; r$
(e) $r \;\{\!|\, S \,|\!\}\; r \;\Rightarrow\; (\mathit{true} \times r) \;\{\!|\, \mathbf{skip} \times S \,|\!\}\; (\mathit{true} \times r)$
(f) $\mathit{true} = S\ \mathit{true} \;\Rightarrow\; (\mathit{true} \times r) \;\{\!|\, S \times \mathbf{skip} \,|\!\}\; (\mathit{true} \times r)$
(g) $(\mathit{true} \times p) \cap (\mathbf{var}\ x, u \cdot b) \;\{\!|\, S \,|\!\}\; (\mathit{true} \times q) \;\Rightarrow\; p \;\{\!|\, \mathbf{begin}\ (\mathbf{var}\ x, u \cdot b) ; S ; \mathbf{end} \,|\!\}\; q$

Algorithmic Refinement Rules

Skip is unit of sequential composition :

$$S ; \mathbf{skip} \;=\; S \;=\; \mathbf{skip} ; S$$

Relational product distribution through composition :

$$(P_1 \times Q_1) ; (P_2 \times Q_2) \;=\; (P_1 ; P_2) \times (Q_1 ; Q_2)$$

Distribution of sequential composition through updates :

(a) $\langle f \rangle ; \langle g \rangle \;=\; \langle f ; g \rangle$
(b) $[P] ; [Q] \;=\; [P ; Q]$
(c) $\{P\} ; \{Q\} \;=\; \{P ; Q\}$

Product distribution through updates :

(a) $\langle f \rangle \times \langle g \rangle \;=\; \langle f \times g \rangle$
(b) $[P] \times [Q] \;=\; [P \times Q]$
(c) $\{P\} \times \{Q\} \;=\; \{P \times Q\}$

Product distribution through sequential composition :

(a) $(S_1 ; T_1) \times (S_2 ; T_2) \;\sqsubseteq\; (S_1 \times S_2) ; (T_1 \times T_2)$
(b) $(S_1 \times \mathbf{skip}) ; (S_2 \times \mathbf{skip}) \;=\; (S_1 ; S_2) \times \mathbf{skip}$
(c) $\mathbf{skip} \times \{R\} ; (S \times \mathbf{skip}) \;=\; S \times \{R\}$
(d) $[P \times Q] \;=\; [P \times Id] ; [Id \times Q] \;=\; [Id \times Q] ; [P \times Id]$
(e) $\{P \times Q\} \;=\; \{P \times Id\} ; \{Id \times Q\} \;=\; \{Id \times Q\} ; \{P \times Id\}$

Data Refinement Rules

The *sequential composition rule* states that the data refinement of a sequential composition is refined by a sequential composition of the data refined components:

$$(S_1; S_2)\downarrow R \ \sqsubseteq\ (S_1 \downarrow R); (S_2 \downarrow R)$$

Data refinement also distributes through demonic and angelic choice:

$$(S \sqcap T)\downarrow R \ \sqsubseteq\ S\downarrow R \sqcap T\downarrow R$$
$$(S \sqcup T)\downarrow R \ \sqsubseteq\ S\downarrow R \sqcup T\downarrow R$$

The *indifferent block rule* reduces data refinement of a block with local variables to a data refinement of a statement inside that block, retaining the local variables:

$$\textbf{begin}\ (p \times \mathit{true}); S; \textbf{end}\downarrow R \ \sqsubseteq\ \textbf{begin}\ (p \times \mathit{true}); S\downarrow(Id \times R); \textbf{end}$$

When the initializing predicate is effected by the data refinement, the *block rule* requires that this predicate is coerced accordingly:

$$\textbf{begin}\ p; S; \textbf{end}\downarrow R \ \sqsubseteq\ \textbf{begin}\ p'; S\downarrow(Id \times R); \textbf{end},$$
$$\text{where } p' \subseteq (\lambda(x, y') \cdot \exists\, y \cdot R\, y'\, y \ \wedge\ p(x, y))$$

There are also two auxiliary block begin rules:

(a) $\textbf{begin}\ p; [(R \times Id)^{-1}] \ \sqsubseteq\ \textbf{begin}\ p'$,
where $p' \subseteq (\lambda(x', y) \cdot \exists x \cdot R\, x'\, x \ \wedge\ p(x, y))$

(b) $\{R\}; \textbf{begin}\ p \ \sqsubseteq\ \textbf{begin}\ p'; \{Id \times R\}$,
where $p' \subseteq (\lambda(x, y') \cdot \exists\, y \cdot R\, y'\, y \ \wedge\ p(x, y))$

Another block-related *local variable rule* allows us to change local variables in a refinement. For any $S : \Xi(\Lambda \times \Sigma)$ and $R : \Lambda' \leftrightarrow \Lambda$,

$$\textbf{begin}\ p; S; \textbf{end} \ \sqsubseteq\ \textbf{begin}\ p'; S\downarrow(R \times Id); \textbf{end},$$
$$\text{where } p' \subseteq (\lambda(x', y) \cdot \exists x \cdot R\, x'\, x \ \wedge\ p(x, y))$$

Using the program variable notation, this rule can be expressed as follows:

$$\textbf{begin}\ (\textbf{var}\ l, u \cdot b); S; \textbf{end} \ \sqsubseteq\ \begin{array}{l}\textbf{begin}\ (\textbf{var}\ l', u \cdot (\exists\, l \cdot R\, l'\, l \wedge b));\\ \quad S\downarrow(R \times Id);\\ \textbf{end}\end{array}$$

The *indifferent statement rule* describes the cases when a statement is not affected by data refinement:

$$(\textbf{skip} \times S)\downarrow(R \times Id) \ \sqsubseteq\ \textbf{skip} \times S \quad \text{and} \quad (S \times \textbf{skip})\downarrow(Id \times R) \ \sqsubseteq\ S \times \textbf{skip}$$

The *iterative choice rule* states that for any $S_i : \Sigma \mapsto \Sigma, R : \Sigma' \leftrightarrow \Sigma$, and any $q_i : \mathscr{P}\Sigma$ indifferent to R,

$$\textbf{do}\ \between_{i=1}^{n} q_i :: S_i\ \textbf{od}\downarrow R \ \sqsubseteq\ \textbf{do}\ \between_{i=1}^{n} q_i :: S_i\downarrow R\ \textbf{od}$$

where indifference means that $q_i = q_i' \times \mathit{true}$ when R is of the form $Id \times R'$, and $q_i = \mathit{true} \times q_i'$ when $R = R' \times Id$.

Finally, the *identity of inverse coercion rule* states that wrapping the statement $S \downarrow R$ in $\uparrow R$ undoes the effect of wrapping S in $\downarrow R$:

$$S \ \sqsubseteq\ (S\downarrow R)\uparrow R$$

Received May 1999
Accepted in revised form March 2000 by E. C. R. Hehner