Class Refinement as Semantics of Correct Subclassing

Ralph Back, Anna Mikhajlova, Joakim von Wright

Turku Centre for Computer Science
TUCS Technical Report No 147
December 1997
ISBN 952-12-0114-2
ISSN 1239-1891

Abstract

Constructing new classes from existing ones by inheritance or subclassing is a characteristic feature of object-oriented development. Imposing semantic constraints on subclassing allows us to ensure that subclass instances may be substituted for the corresponding superclass instances without invalidating the clients. We define the class refinement relation which captures these semantic constraints. Clients of class instances are modeled as programs making an iterative choice over invocation of class methods, and we prove that when a subclass refines its superclass, substituting subclass instances for superclass instances is refinement for the clients.

Keywords: correctness, class refinement, subclassing, code inheritance, substitutability, semantics of object-oriented constructs

TUCS Research Group
Programming Methodology Research Group

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. We consider here correctness of the class construction mechanism known as subclassing or inheritance. We formalize classes and relationships between them in a logic framework known as the *refinement calculus* [4, 17, 7], which is used for reasoning about correctness of programs and their refinement in a rigorous, mathematically precise manner. 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 formalization, 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.

Our formalization of classes is based on the simply typed lambda calculus; yet it is powerful enough to model subtyping polymorphism and dynamic binding of methods. For modeling *subtyping polymorphism* we employ sum types as suggested in [8] and further developed in [16]. Using summations allows us to model contravariance and covariance on method parameters in a simple way. Reasoning about method calls with dynamic binding is also quite simple in this model.

Behaviour of subclass instances must be consistent with that of superclass instances to allow polymorphic substitutability, which is central to the object-oriented approach. Subclass instances are guaranteed to behave as expected from instances of the corresponding superclass if the subclass is a *refinement* of the superclass. We give a definition of class refinement, which we regard as semantics of correct subclassing, and relate it to substitutability of subclass instances for superclass instances in clients. Namely, we prove that when a class C' refines a class C, substituting instances of C' for instances of C is refinement for the clients.

2 Refinement Calculus Basics

We formalize objects, classes, and relationships between them in the refinement calculus, which is a logic framework 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 modeled 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 $\mathcal{P}\Sigma$. The *entailment ordering* on predicates is defined by pointwise extension, so that for $p, q : \mathcal{P}\Sigma$,

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

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

A *relation* from Σ to Γ is a function of type $\Sigma \to \mathcal{P}\Gamma$ 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 $|\,f\,| : \Sigma \leftrightarrow \Gamma$:

$$|\,f\,|\,\sigma\,\gamma \;\;\widehat{=}\;\; f\,\sigma \;=\; \gamma$$

Functional and relational compositions are defined in a standard way. For functions $f : \Sigma \to \Gamma$ and $g : \Gamma \to \Delta$, their composition $(f; g)$ is equal to $\lambda\sigma \cdot g\,(f\,\sigma)$. For relations $P : \Sigma \leftrightarrow \Gamma$ and $Q : \Gamma \leftrightarrow \Delta$, we have

$$(P; Q)\,\sigma\,\delta \;\;\widehat{=}\;\; \exists\,\gamma \cdot P\,\sigma\,\gamma \wedge Q\,\gamma\,\delta$$

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

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

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 $S, T : \Sigma \mapsto \Gamma$,

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

2.2 Product Operators

Product operators combine predicates, relations, and predicate transformers by forming cartesian products of their state spaces. For predicates $p : \mathcal{P}\Sigma$ and $q : \mathcal{P}\Gamma$, their product $p \times q$ is a predicate of type $\mathcal{P}(\Sigma \times \Gamma)$ defined by

$$(p \times q)\,(\sigma, \gamma) \;\;\widehat{=}\;\; p\,\sigma \wedge q\,\gamma$$

A product $f \times g$ of two functions $f : \Sigma_1 \to \Gamma_1$ and $g : \Sigma_2 \to \Gamma_2$ is a function of type $(\Sigma_1 \times \Sigma_2) \to (\Gamma_1 \times \Gamma_2)$ defined by

$$(f \times g)\,(\sigma_1, \sigma_2) \;\;\widehat{=}\;\; (f\,\sigma_1, g\,\sigma_2)$$

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:

$$(S_1 \times S_2)\,q \;\widehat{=}\; (\cup\, q_1, q_2 \,|\, q_1 \times q_2 \subseteq q \cdot S_1\,q_1 \times S_2\,q_2)$$

In addition to many other useful properties, presented e.g., in [8, 6], 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 $S : \Sigma \mapsto \Sigma$ we define *lifting* to a product predicate transformer of type $\Sigma \times \Gamma \mapsto \Sigma \times \Gamma$ as $S \times \mathbf{skip}$. Similarly, lifting S to a product predicate transformer of type $\Gamma \times \Sigma \mapsto \Gamma \times \Sigma$ is defined by $\mathbf{skip} \times S$.

2.3 Sum Types and Operators

For modeling subtyping 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. Sum types, as well as product types, associate to the right, so that $\Sigma_1 + \Sigma_2 + \Sigma_3 = \Sigma_1 + (\Sigma_2 + \Sigma_3)$. Associated with the sum types, are the *injection functions* which map elements of the base type to elements of the summation

$$\iota_1 : \Sigma \to \Sigma + \Gamma \qquad \iota_2 : \Gamma \to \Sigma + \Gamma$$

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 $\pi_1^{-1} = |\,\iota_1\,|$, where $|\,\iota_1\,|$ 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 $\mathit{ran}\ \iota_1$ and $\mathit{ran}\ \iota_2$ partition $\Sigma + \Gamma$. For $\sigma : \Sigma + \Gamma$, the projection π_1 will relate it to a unique $\sigma' : \Sigma$ only if $\sigma \in \mathit{ran}\ \iota_1$, and similarly for π_2.

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

2.4 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 $q : \mathcal{P}\Gamma$ to the weakest precondition $p : \mathcal{P}\Sigma$ 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.

A statement S is said to be correct with respect to specification $(pre, post)$ if $pre \subseteq S\,post$. 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.

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.

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 modeled by functional composition of predicate transformers. For $S : \Sigma \mapsto \Gamma$, $T : \Gamma \mapsto \Delta$ and $q : \mathcal{P}\Delta$,

$$(S; T)\, q \;\hat{=}\; S\,(T\, q)$$

The program statement **skip** is modeled by the identity predicate transformer and its execution has no effect on the program state.

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

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

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 [6], 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

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

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

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

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.

Lambda terms, unfortunately, do not maintain consistent naming of state components, due to the possibility of α-conversion of bound variables. To enforce the clearly desirable 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 modeled using the basic predicate transformers and operators presented above, using the program variable notation. For example, the (multiple) assignment statement may be modeled by the functional update:

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

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

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

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:

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

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

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

A variant of iteration, the *iterative choice* introduced in [7], allows the user to choose repeatedly an alternative that is enabled and have it executed until the user decides to stop:

$$\mathbf{do}\ g_1 :: S_1 \lozenge \ldots \lozenge\ g_n :: S_n\ \mathbf{od} \mathrel{\widehat{=}} (\mu X \cdot \{g_1\}; S_1; X \sqcup \ldots \sqcup \{g_n\}; S_n; X \sqcup \mathbf{skip})$$

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

Finally, the language supports blocks with *local variables*. Block beginning and end are modeled by demonic and functional updates respectively:

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

Here p is the predicate initializing the local variables. For example, the construct $(\mathbf{var}\ u \cdot \mathbf{begin}\ (\mathbf{var}\ x, u \cdot b); S; \mathbf{end})$ states that the program variable x, initialized according to the boolean expression b, is local to the statement S. When the variable declaration is clear from the context, we will, for simplicity, write just $\mathbf{begin\ var}\ x \cdot b; S; \mathbf{end}$.

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

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

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

As suggested in [7], 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** $x, u \cdot b$); S; **end**	*(block with local variables)*

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

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

Further on, we will explain how to extend this language with object-oriented constructs.

2.5 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 [5], the statement S is said to be data refined by S' via R, denoted $S \sqsubseteq_R S'$, if

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

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 $[R^{-1}]$ are each others inverses, in the sense that $\{R\}; [R^{-1}] \sqsubseteq$ **skip** and **skip** $\sqsubseteq [R^{-1}]; \{R\}$. Further on we will abbreviate $\{R\}; S; [R^{-1}]$ by $S{\downarrow}R$.

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 [7, 17]. Let us briefly present a few of these rules which will be later used in proofs. The *sequential* *composition rule* states that the data refinement of a sequential composition is refined by a sequential composition of the data refined components:

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

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:

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

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

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

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 $R : \Lambda' \leftrightarrow \Lambda$,

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

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:

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

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

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

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

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

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

3 Modeling Object-Oriented Constructs

We focus on modeling class-based object-oriented languages, which form the mainstream of object-oriented programming. Accordingly, we take a view that *objects* are instantiated by *classes*.

A class is a template used to describe objects with identical behaviour through specifying their *interface*. The interface represents the signatures of object methods, i.e. the method name and the types of value and result parameters. We consider the *object type* to be the type of object *attributes*. Without loss of generality, we consider all attributes as private or hidden, and all methods as public or visible to clients of the object.

A class can be given by the following declaration:

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

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 $()$, 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. We will use an identifier *self* for the tuple $(attr_1, \ldots, attr_m)$. The type of *self* is then $\Sigma \;=\; \Sigma_1 \times \ldots \times \Sigma_m$.[1]

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. The statement $K : \Gamma_0 \mapsto \Sigma \times \Gamma_0$, representing the body of the constructor, introduces the attributes into the state space and initializes them using the input parameter $x_0 : \Gamma_0$. Methods $Meth_1$ through $Meth_n$, specified by bodies $M_1, \ldots, M_n$, operate on the attributes and realize 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.

[1] 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.

3.1 Object Instantiation and Method Invocations

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

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

where $\mathit{Swap} = \langle \lambda x, y, z \cdot y, x, z\rangle$. A variable $x_0 : \Gamma_0$ is first entered into the state space and initialized 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 variable c and initializes 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 object type initialized in this way can be local to a block:

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

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

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

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

3.2 Modeling 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:

$$(\textbf{var } c, u \cdot \textbf{begin var } l \cdot p; \textbf{do } \between_{i=1}^{n} q_i :: c.\mathit{Meth}_i(g_i, d_i); L_i \textbf{ od}; \textbf{end})$$

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

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 $Meth_j(\mathbf{val}\ x_j : \Gamma_j, \mathbf{res}\ y_j : \Delta_j)$ can, therefore, be described by

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

where *self* is free in p, 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 refinement ordering.

3.3 Modeling Dynamic Objects

Following [7], 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:

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

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:

$$\begin{array}{l} \mathbf{type\ pointer\ to}\ \tau(C)\ \widehat{=}\ \mathit{Nat}; \\ \mathbf{var}\ \mathit{nil} : \mathbf{pointer\ to}\ \tau(C) := 0; \\ \mathbf{var}\ \mathit{new} : \mathbf{pointer\ to}\ \tau(C) := 1; \end{array}$$

Dynamic creation of an object of type $\tau(C)$ and association of a pointer $p : \mathbf{pointer\ to}\ \tau(C)$ with this object are modeled as follows:

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

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

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.

```
TextDoc = class
  var text : String,
      views : set of pointer to τ(View)

  TextDoc (val t : String) =
    begin var (text, views) ·
      text = t ∧ views = {},

  AddView (val v : pointer to τ(View)) =
    [v ≠ nil]; views := views ∪ {v},

  AddText (val t : String) =
    text := text⌒t; self.Notify(),

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

  Notify () =
    begin var (vs, v) · vs = views;
      while vs ≠ {} do
        [v := v' | v' ∈ vs];
        vs := vs \ v; v↑.Update()
      od;
    end
end
```

```
View = class
  var txt : String,
      doc : pointer to τ(TextDoc)

  View (val d : pointer to τ(TextDoc)) =
    [d ≠ nil];
    begin var (txt, doc) · doc = d;
      doc↑.GetText(txt),

  AddText (val t : String) =
    doc↑.AddText(t),

  Update () = doc↑.GetText(txt)
end
```

```
TextEditor = class
  ...
  OpenTextDoc () =
    begin var d : pointer to τ(TextDoc); d := new TextDoc("");
      begin var (v1, v2 : pointer to τ(View));
        v1 := new View(d); d↑.AddView(v1);
        v2 := new View(d); d↑.AddView(v2);
      end;
    end,
  ...
end
```

Figure 1: Example of class specification

Views are responsible for presenting the textual data in various windows and providing operations for changing it. A client of a text editor can open a new text document, displayed in two different windows, by invoking the method *OpenTextDoc*. When one of these 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 [11] 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 [11], we only would like to notice 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] We limit our consideration of class construction to inheritance and overriding.

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

$$\begin{array}{l}
C' \;=\; \textbf{subclass of } C \\
\quad \textbf{var } attr_1 : \Sigma_1, \ldots, attr_i : \Sigma_i, attr'_1 : \Sigma'_1, \ldots, attr'_p : \Sigma'_p \\
\quad C'\,(\textbf{val } x_0 : \Gamma_0) \;=\; K', \\
\quad Meth_1\,(\textbf{val } x_1 : \Gamma_1, \textbf{res } y_1 : \Delta_1) \;=\; M'_1, \\
\quad \ldots \\
\quad Meth_k\,(\textbf{val } x_k : \Gamma_k, \textbf{res } y_k : \Delta_k) \;=\; M'_k \\
\textbf{end}
\end{array}$$

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. Methods defined in C and operating only on the attributes inherited by C' can be invoked from $M'_1, \ldots, M'_k$ using a special identifier *super*. For example, a method $Meth_i$ defined in C by $M \times \textbf{skip}$, which skips on $attr_{i+1}, \ldots, attr_m$, can be super-called inside any of $M'_1, \ldots, M'_k$ as

$$super.Meth_i() \;\;\widehat{=}\;\; M \times \textbf{skip}$$

with **skip** operating on $attr'_1, \ldots, attr'_p$ in the latter case. Methods of the superclass operating only on the inherited attributes 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

[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.

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. Syntactic subclassing implies conformance of interfaces, in the sense that a subclass specifies an interface conforming to the one specified by its superclass. In the simple case the interface specified by a subclass is the same as that of the superclass. In the next section we explain how this requirement can be relaxed.

3.6 Modeling Subtyping Polymorphism and Dynamic Binding

To model subtyping 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

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

Suppose method $Meth_i\,(\textbf{val}\ x_i : \Gamma_i, \textbf{res}\ y_i : \Delta_i)$ is specified in C and D by statements M_i and M_i' respectively. An invocation of this method on an object p of type $\tau(C)^+$ is modeled 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:

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

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 modeled 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, $\textbf{abort}; S = \textbf{abort}$.

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:

$$\textbf{create var}\ p.C^+(e) \ \hat{=} \ \begin{pmatrix} \textbf{create var}\ c.C(e); \\ \textbf{begin var}\ p \cdot \pi_{\tau(C)}\, p\, c; \\ \quad \mathit{Swap}; \\ \textbf{end} \end{pmatrix} \sqcap \begin{pmatrix} \textbf{create var}\ d.D(e); \\ \textbf{begin var}\ p \cdot \pi_{\tau(D)}\, p\, d; \\ \quad \mathit{Swap}; \\ \textbf{end} \end{pmatrix}$$

Intuitively, the demonic choice can be interpreted as underspecification, which would need to 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.

Modeling of both the invocation of a method on a polymorphic variable and the instantiation of such a variable generalizes to class hierarchies with several classes in a straightforward way, recursively.

Being equipped with subtyping polymorphism, we can allow overriding methods in a subclass to be *generalized* on the type of value parameters or *specialized* 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, $Meth_i\,(\textbf{val}\ x_i : \Gamma_i, \textbf{res}\ y_i : \Delta_i)$ 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 $Meth_i\,(\textbf{val}\ x'_i : \Gamma'_i, \textbf{res}\ y'_i : \Delta'_i)$ 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 modeled by

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

As before, $u : \Phi$ are the global variables, including d_i. The value parameter x'_i is initialized 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.

[3]For a more extensive explanation of covariance and contravariance see, e.g., [1].

Subtyping 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 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 takes place 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

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

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:

$$\begin{array}{l}(\mathbf{var}\ self, x_j, y_j \cdot super.Meth_j(x_j, y_j)) \ \widehat{=} \\ \quad (\mathbf{var}\ self, x_j, y_j \cdot \ \mathbf{begin}\ \langle\rho\rangle\, p; \\ \qquad\qquad \mathbf{do}\ \between_{i=1}^{n} \langle\rho\rangle\, q_i :: self.Meth_i(g_i, d_i); L_i\!\downarrow |\,\rho\,|\ \ \mathbf{od}; \\ \qquad \mathbf{end})\end{array}$$

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 $p, q_1, \ldots, q_n$ 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.2. 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*

$$
\begin{array}{l}
Bag \;=\; \mathbf{class} \\
\quad \mathbf{var}\ b : \mathbf{bag\ of}\ Char \\
\quad Bag() \;=\; \mathbf{begin\ var}\ b \cdot b = \llbracket\rrbracket, \\
\quad Add(\mathbf{val}\ c : Char) \;=\; b := b \cup \llbracket c \rrbracket, \\
\quad AddAll(\mathbf{val}\ nb : \mathbf{bag\ of}\ Char) \;= \\
\qquad \mathbf{while}\ nb \neq \llbracket\rrbracket\ \mathbf{do} \\
\qquad\quad \mathbf{begin\ var}\ c \cdot c \in nb; \\
\qquad\qquad self.Add(c); nb := nb - \llbracket c \rrbracket; \\
\qquad\quad \mathbf{end} \\
\qquad \mathbf{od}; \\
\mathbf{end}
\end{array}
$$

$$
\begin{array}{l}
CountingBag \;=\; \mathbf{subclass\ of}\ Bag \\
\quad \mathbf{var}\ b : \mathbf{bag\ of}\ Char, n : Nat \\
\quad CountingBag() \;= \\
\qquad \mathbf{begin\ var}\ (b,n) \cdot b = \llbracket\rrbracket \wedge n = 0, \\
\quad Add(\mathbf{val}\ c : Char) \;= \\
\qquad n := n+1; super.Add(c) \\
\mathbf{end}
\end{array}
$$

Figure 2: Example of subclassing with dynamic binding of self-referential methods

as defined in *Bag*:

$$
\begin{array}{lll}
Bag :: AddAll(\mathbf{val}\ nb : \mathbf{bag\ of}\ Char) & = & \mathbf{while}\ nb \neq \llbracket\rrbracket\ \mathbf{do} \\
 & & \quad \mathbf{begin\ var}\ c \cdot c \in nb; \\
 & & \qquad b := b \cup \llbracket c \rrbracket; nb := nb - \llbracket c \rrbracket; \\
 & & \quad \mathbf{end} \\
 & & \mathbf{od}
\end{array}
$$

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

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

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

$$
\begin{array}{l}
(\mathbf{var}\ (b,n), nb \cdot \mathbf{while}\ \langle\rho\rangle\, (\mathbf{var}\ b, nb \cdot nb \neq \llbracket\rrbracket)\ \mathbf{do} \\
\qquad\qquad \mathbf{begin}\ \langle\rho\rangle\, (\mathbf{var}\ c, b, nb \cdot c \in nb); \\
\qquad\qquad\quad self.Add(c); (\mathbf{var}\ c, b, nb \cdot nb := nb - \llbracket c \rrbracket){\downarrow} \mid \rho \mid; \\
\qquad\qquad \mathbf{end} \\
\qquad \mathbf{od})
\end{array}
$$

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

$$
\begin{array}{l}
(\mathbf{var}\ (b,n), nb \cdot \mathbf{while}\ (\mathbf{var}\ (b,n), nb \cdot nb \neq \llbracket\rrbracket)\ \mathbf{do} \\
\qquad\qquad \mathbf{begin}\ (\mathbf{var}\ c, (b,n), nb \cdot c \in nb); \\
\qquad\qquad\quad self.Add(c); (\mathbf{var}\ c, (b,n), nb \cdot nb := nb - \llbracket c \rrbracket) \\
\qquad\qquad \mathbf{end} \\
\qquad \mathbf{od})
\end{array}
$$

The self-call, being on $self = (b, n)$, 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. Suppose classes C and C' specify interfaces

$$\begin{array}{l} Meth_1\,(\mathbf{val}\ :\Gamma_1, \mathbf{res}\ :\Delta_1), \ldots, Meth_n\,(\mathbf{val}\ :\Gamma_n, \mathbf{res}\ :\Delta_n) \text{ and} \\ Meth_1\,(\mathbf{val}\ :\Gamma'_1, \mathbf{res}\ :\Delta'_1), \ldots, Meth_n\,(\mathbf{val}\ :\Gamma'_n, \mathbf{res}\ :\Delta'_n) \end{array}$$

respectively. Let C and C' be modeled 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 are either the same or contravariant, so that $\Gamma_0 <: \Gamma'_0$ and $\Gamma_i <: \Gamma'_i$. The result parameter types of the 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 $(R = \lambda c \cdot \{a \mid R\, c\, a\})$. The refinement of class constructors K and K' with respect to R is defined by

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

where π_{Γ_0} is the projection relation coercing Γ'_0 to Γ_0. Alternative and equivalent definitions of class constructor refinement are

$$\begin{array}{rl} K \sqsubseteq_R K' & \equiv\; \{\pi_{\Gamma_0}\}; K; [(R \times \pi_{\Gamma_0})^{-1}] \;\sqsubseteq\; K' \\ & \equiv\; K; [(R \times \pi_{\Gamma_0})^{-1}] \;\sqsubseteq\; [\pi_{\Gamma_0}^{-1}]; K' \\ & \equiv\; K \;\sqsubseteq\; [\pi_{\Gamma_0}^{-1}]; K'; \{R \times \pi_{\Gamma_0}\} \end{array}$$

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, $|\iota_{\Delta'_i}| = Id$.

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

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

As was proved in [16], the class refinement relation is reflexive and transitive.

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 any relation $R : \Sigma' \leftrightarrow \Sigma$, any statement $S : \Xi(\Sigma \times \Phi)$, and any 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 any relation $R : \Sigma' \leftrightarrow \Sigma$ and any input argument $g_i : \Gamma_i$,*

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

Proofs of Lemma 1 and Lemma 2 are presented in the Appendix.

The following theorem proves that clients using objects instantiated by a more abstract class are refined when using objects instantiated by a more concrete class.

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

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

Proof Rewriting this implication with the definition of class refinement, we get

$$(\exists R \cdot K \sqsubseteq_R K' \wedge (\forall i \mid 1 \le i \le n \cdot M_i \sqsubseteq_R M_i')) \Rightarrow \\ \quad \textbf{create var } c.C(e); \mathcal{K}\ [c]; \textbf{end} \sqsubseteq \textbf{create var } c'.C'(e); \mathcal{K}\ [c']; \textbf{end}$$

Assume that R is a relation that satisfies the left-hand side of the implication. Using the first conjunct of the antecedent and Lemma 1, the implication is then reduced to

$$(\forall i \mid 1 \le i \le n \cdot M_i \sqsubseteq_R M_i') \Rightarrow \\ \quad (\textbf{var } c, u \cdot \mathcal{K}\ [c]){\downarrow}(R \times Id) \sqsubseteq (\textbf{var } c', u \cdot \mathcal{K}\ [c'])$$

Assuming the antecedent and expressing $\mathcal{K}$ as an iterative choice of method invocations, we get

$$\textbf{begin } (\textbf{var } l, c, u \cdot p); \textbf{do } \between_{i=1}^{n} q_i :: c.Meth_i(g_i, d_i); L_i \textbf{ od}; \textbf{end}{\downarrow}(R \times Id) \sqsubseteq \\ \textbf{begin } (\textbf{var } l, c, u \cdot p); \textbf{do } \between_{i=1}^{n} q_i :: c'.Meth_i(g_i, d_i); L_i \textbf{ od}; \textbf{end}$$

The indifferent block rule allows us to reduce this goal to

$$(\mathbf{var}\ l, c, u\ \cdot\ \mathbf{do}\ \lozenge_{i=1}^{n} q_i :: c.Meth_i(g_i, d_i); L_i\ \mathbf{od}){\downarrow}(Id \times R \times Id)\ \sqsubseteq$$
$$(\mathbf{var}\ l, c', u\ \cdot\ \mathbf{do}\ \lozenge_{i=1}^{n} q_i :: c'.Meth_i(g_i, d_i); L_i\ \mathbf{od})$$

which we prove by beginning with the left-hand side and refining it to the right-hand side as follows:

$$(\mathbf{var}\ l, c, u\ \cdot\ \mathbf{do}\ \lozenge_{i=1}^{n} q_i :: c.Meth_i(g_i, d_i); L_i\ \mathbf{od}){\downarrow}(Id \times R \times Id)$$

$\sqsubseteq$ { iterative choice rule, since every q_i is of the form $q_i' \times \mathit{true} \times q_i''$ }

$$(\mathbf{var}\ l, c, u\ \cdot\ \mathbf{do}\ \lozenge_{i=1}^{n} q_i :: (c.Meth_i(g_i, d_i); L_i){\downarrow}(Id \times R \times Id)\ \mathbf{od})$$

$\sqsubseteq$ $\left\{\begin{array}{l}\text{sequential composition and indifferent statement rules,}\\ \text{since all statements } L_i \text{ are indifferent to } Id \times R \times Id\end{array}\right\}$

$$(\mathbf{var}\ l, c, u\ \cdot\ \mathbf{do}\ \lozenge_{i=1}^{n} q_i :: c.Meth_i(g_i, d_i){\downarrow}(Id \times R \times Id); L_i\ \mathbf{od})$$

$\sqsubseteq$ { Lemma 2, using the assumption }

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

This completes the proof. □

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.

As we have already stated, any concrete instantiation of a polymorphic variable refines the polymorphic instantiation. It is also interesting to examine the relationship between method calls of the same method from a variable of some object type and a variable of its subtype. The following two lemmas prove the complementing results about such method invocations. The first lemma states that invoking a method on a variable of a polymorphic object type refines invocation of the same method on a variable of its subtype with respect to the corresponding projection.

Lemma 3 *Assume that $Meth_i$ is specified in class C and all its subclasses. Let $\tau(C)^+$ be a sum type of $\tau(C)$ and $\tau(C')^+$, variables c and c' be of types $\tau(C)^+$ and $\tau(C')^+$ respectively, a global state $u : \Phi$ include a variable $d_i : \Delta_i$, and $g_i : \Gamma_i$ be some expression. Then for the projection $\pi_{\tau(C')^+} : \tau(C)^+ \leftrightarrow \tau(C')^+$,*

$$(\mathbf{var}\ c', u\ \cdot\ c'.Meth_i(g_i, d_i)){\downarrow}(\pi_{\tau(C')^+} \times Id)\quad \sqsubseteq\quad (\mathbf{var}\ c, u\ \cdot\ c.Meth_i(g_i, d_i))$$

The proof is straightforward, by rewriting with definitions and using the weakening property $S \sqsubseteq T \sqcup S$, which holds for all statements S and

T. This property may at first seem counterintuitive, since a client using instances of some subclass of C in a refinement may use instances of C as well. The intuitive justification behind this property is that a refined client can *do more* than the original client by being able to work with superclass instances along with subclass instances, while the original one worked with subclass instances only.

The second lemma proves a dual result, namely, that invoking a method on a variable of a polymorphic object type is refined by invocation of the same method on a variable of its subtype with respect to the corresponding injection relation.

Lemma 4 *Assume that $Meth_i$ is specified in class C and all its subclasses. Let $\tau(C)^+$ be a sum type of $\tau(C)$ and $\tau(C')^+$, variables c and c' be of types $\tau(C)^+$ and $\tau(C')^+$ respectively, a global state $u : \Phi$ include a variable $d_i : \Delta_i$, and $g_i : \Gamma_i$ be some expression. Then for the injection $\iota_{\tau(C')^+} : \tau(C')^+ \to \tau(C)^+$,*

$$(\mathbf{var}\ c, u \cdot c.Meth_i(g_i, d_i))\downarrow(|\iota_{\tau(C')^+}| \times Id) \quad \sqsubseteq \quad (\mathbf{var}\ c', u \cdot c'.Meth_i(g_i, d_i))$$

The proof depends on the fact that the definition of method invocation on a polymorphic variable could also be written as a demonic choice of guarded alternatives with disjoint conditions ($c \in ran\ \iota_{\tau(C)}$) and ($c \in ran\ \iota_{\tau(C')^+}$). The property is proved by rewriting with definitions and using the weakening property $S \sqcap T \sqsubseteq T$, which holds for all statements S and T.

5 Conclusions and Related work

We build a logical framework for reasoning about object-oriented programs as a conservative extension of refinement calculus, and give a definition of class refinement which we regard as semantics of correct subclassing. This work is based on [16], but concentrates on class refinement and its special cases for generic classes. One of the main contributions of the present paper is in modeling clients of class instances as 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 relate the notion of correct subclassing to substitutability of class instances in clients, and prove that substituting instances of a refined class for instances of the original class is refinement for the clients.

Related work in formalization of object-oriented concepts includes [9, 18, 19]. Cook and Palsberg in [9] 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 procedural methods as well.

Semantics of a simple Oberon-like programming language with similar specification constructs as here, also based on predicate transformers, is defined by Naumann in [18]. Sekerinski [19] 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 Back and Butler in [8] and further elaborated in [16]. One motivation for moving to sum types is to avoid the 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 [19]. It turned out that this leads to complications when reasoning about method calls, which are not present when using the model of sum types.

Data refinement of modules, abstract data types, and abstract machines as e.g., in [13, 17, 2] forms a basis for class refinement. The latter, however, has special features due to subtyping polymorphism and dynamic binding. Behavioural dependencies in presence of subclassing have also been studied in various extensions of Z specification languages, e.g. [14], but only between class specifications and not implementations. Behavioural subtyping is another area of related work [3, 15, 10] where a distinction is also made between the specification of a class and its implementation. By having specification constructs as part of the (extended) programming language, this distinction becomes unnecessary.

Stata and Guttag in [20] use specialization specifications in style of behavioural subtyping and require that specification of subclasses be constrained. They informally define correctness of a class with respect to its specialization specification, and require verification of correctness as well as verification of specification constraining, using "standard simulation techniques". In our approach there is no gap between specifications and implementations, eliminating a considerable amount of proof obligations. Further on, Stata and Guttag claim that declarative specifications are more abstract and easier to understand than operational ones capturing method invocation order. We, on the contrary, 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 impossible to specify in terms of pre- and post-conditions. Similar ideas are supported by Helm et al. in [12]. They include method calls in abstract specifications of contracts to express behavioural dependencies between cooperating objects.

A number of issues have not been treated in this work, in particular, addition of extra methods in subclassing. This issue is rather non-trivial because of inconsistencies possibly introduced by extra methods, which become apparent in presence of subtype aliasing, and constitutes the topic of current research.

Acknowledgments

The authors would like to thank Emil Sekerinski and Leonid Mikhajlov for valuable comments on this paper.

References

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

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

[3] Pierre America. A behavioural approach to subtyping in object-oriented programming languages. Technical Report 443, Philips Research Laboratories, April 1989.

[4] R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Center Tracts*. Mathematical Centre, Amsterdam, 1980.

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

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

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

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] 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.

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

[16] A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97*, LNCS 1313, pages 82–101. Springer, 1997.

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

[18] 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.

[19] 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.

[20] Raymie Stata and John V. Guttag. Modular reasoning in the presence of subclassing. In *Proceedings of OOPSLA'95*, pages 200–214. ACM SIGPLAN notices, October 1995.

Appendix

Additional Algebraic 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) $\{P \times Q\} \;=\; \{P \times Id\}; \{Id \times Q\} \;=\; \{Id \times Q\}; \{P \times Id\}$

Additional Data Refinement Rules

Angelic choice rule : $(S \sqcup T){\downarrow}R \;\sqsubseteq\; S{\downarrow}R \sqcup T{\downarrow}R$

Block begin rules :

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

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

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 any relation $R : \Sigma' \leftrightarrow \Sigma$, any statement $S : \Xi(\Sigma \times \Phi)$, and any constructor input argument $e : \Gamma_0$,

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

Proof We assume the antecedent and prove the consequent by refining the left-hand side to the right-hand side:

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

$$\equiv \left\{ \text{definition of constructor invocation} \right\}$$

$$\begin{array}{l} \textbf{begin } (\textbf{var } x_0, u \cdot x_0 = e); K \times \textbf{skip}; \\ \textbf{begin } (\textbf{var } c, (\mathit{self}, x_0), u \cdot c = \mathit{self}); \mathit{Swap}; \textbf{end}; S; \textbf{end} \end{array}$$

$$\equiv \left\{ \text{definitions} \right\}$$

$$\begin{array}{l} [\lambda u \cdot \lambda(x_0, u') \cdot x_0 = e \wedge u = u']; K \times \textbf{skip}; \\ [\lambda((\mathit{self}, x_0), u) \cdot \lambda(c, (\mathit{self}', x_0'), u') \cdot c = \mathit{self} \wedge \\ \quad ((\mathit{self}, x_0), u) = ((\mathit{self}', x_0'), u')]; \\ \langle \lambda(c, (\mathit{self}, x_0), u) \cdot ((\mathit{self}, x_0), c, u) \rangle; \\ \langle \lambda((\mathit{self}, x_0), c, u) \cdot (c, u) \rangle; S; \langle \lambda(c, u) \cdot u \rangle \end{array}$$

$$\equiv \left\{ \begin{array}{l} \text{demonic update of a functional relation } |f| \text{ is equal to} \\ \text{a functional update of a function } f\text{: } [|f|] = \langle f \rangle, \\ \text{distribution of sequential composition through functional updates,} \\ \text{definition of functional composition, logic} \end{array} \right\}$$

$$\begin{array}{l} [\lambda u \cdot \lambda(x_0, u') \cdot x_0 = e \wedge u = u']; K \times \textbf{skip}; \\ \langle \lambda((\mathit{self}, x_0), u) \cdot (\mathit{self}, (\mathit{self}, x_0), u) \rangle; \\ \langle \lambda(c, (\mathit{self}, x_0), u) \cdot (c, u) \rangle; S; \langle \lambda(c, u) \cdot u \rangle \end{array}$$

$$\sqsubseteq \left\{ \text{focus on a subexpression} \right\}$$

$$[\lambda u \cdot \lambda(x_0, u') \cdot x_0 = e \wedge u = u']; K \times \textbf{skip};$$

$$\sqsubseteq \left\{ \text{general rule } \textbf{skip} \sqsubseteq [R^{-1}]; \{R\} \right\}$$

$$\begin{array}{l} [\lambda u \cdot \lambda(x_0, u') \cdot x_0 = e \wedge u = u']; \\ [(\pi_{\Gamma_0} \times \mathit{Id})^{-1}]; \{\pi_{\Gamma_0} \times \mathit{Id}\}; K \times \textbf{skip} \end{array}$$

$$\equiv \left\{ \begin{array}{l} \text{distribution of sequential composition through demonic updates,} \\ \text{definition of relational composition, logic} \end{array} \right\}$$

$$[\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\, e \wedge u = u']; \{\pi_{\Gamma_0} \times \mathit{Id}\}; K \times \textbf{skip}$$

$$\equiv \left\{ \begin{array}{l} \text{product distribution through angelic updates, then} \\ \text{product distribution through sequential composition (b)} \end{array} \right\}$$

$$[\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\, e \wedge u = u']; (\{\pi_{\Gamma_0}\}; K) \times \textbf{skip}$$

$$\sqsubseteq \left\{ \text{assumption} \right\}$$

$$[\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\, e \wedge u = u']; (K'; \{R \times \pi_{\Gamma_0}\}) \times \textbf{skip}$$

$$\equiv \left\{ \begin{array}{l} \text{product distribution through sequential composition (b), then} \\ \text{product distribution through angelic updates} \end{array} \right\}$$

$$[\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\, e \wedge u = u']; K' \times \textbf{skip}; \{(R \times \pi_{\Gamma_0}) \times \mathit{Id}\}$$

$$
\begin{aligned}
&\equiv \\
&\quad [\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\, e \wedge u = u']; K' \times \mathbf{skip}; \\
&\quad \{(R \times \pi_{\Gamma_0}) \times Id\}; \\
&\quad \langle \lambda((\mathit{self}, x_0), u) \cdot (\mathit{self}, (\mathit{self}, x_0), u)\rangle; \\
&\quad \langle \lambda(c, (\mathit{self}, x_0), u) \cdot (c, u)\rangle; S; \langle \lambda(c, u) \cdot u\rangle \\
&\sqsubseteq \left\{\begin{array}{l}\text{definition of sequential composition,}\\ \text{definitions of angelic and functional updates, logic}\end{array}\right\} \\
&\quad [\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\, e \wedge u = u']; K' \times \mathbf{skip}; \\
&\quad \langle \lambda((\mathit{self}', x_0'), u) \cdot (\mathit{self}', (\mathit{self}', x_0'), u)\rangle; \\
&\quad \{R \times (R \times \pi_{\Gamma_0}) \times Id\}; \\
&\quad \langle \lambda(c, (\mathit{self}, x_0), u) \cdot (c, u)\rangle; S; \langle \lambda(c, u) \cdot u\rangle \\
&\sqsubseteq \left\{\text{definitions, logic}\right\} \\
&\quad [\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\, e \wedge u = u']; K' \times \mathbf{skip}; \\
&\quad \langle \lambda((\mathit{self}', x_0'), u) \cdot (\mathit{self}', (\mathit{self}', x_0'), u)\rangle; \\
&\quad \langle \lambda(c', (\mathit{self}', x_0'), u) \cdot (c', u)\rangle; \\
&\quad \{R \times Id\}; \\
&\quad S; \langle \lambda(c, u) \cdot u\rangle \\
&\sqsubseteq \left\{\text{general rule } \mathbf{skip} \sqsubseteq [R^{-1}]; \{R\}, \text{ definition of } \downarrow\right\} \\
&\quad [\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\, e \wedge u = u']; K' \times \mathbf{skip}; \\
&\quad \langle \lambda((\mathit{self}', x_0'), u) \cdot (\mathit{self}', (\mathit{self}', x_0'), u)\rangle; \\
&\quad \langle \lambda(c', (\mathit{self}', x_0'), u) \cdot (c', u)\rangle; \\
&\quad S\!\downarrow\!(R \times Id); \\
&\quad \{R \times Id\}; \langle \lambda(c, u) \cdot u\rangle \\
&\sqsubseteq \left\{\text{definitions, logic}\right\} \\
&\quad [\lambda u \cdot \lambda(x_0', u') \cdot x_0' = \iota_{\Gamma_0}\, e \wedge u = u']; K' \times \mathbf{skip}; \\
&\quad \langle \lambda((\mathit{self}', x_0'), u) \cdot (\mathit{self}', (\mathit{self}', x_0'), u)\rangle; \\
&\quad \langle \lambda(c', (\mathit{self}', x_0'), u) \cdot (c', u)\rangle; S\!\downarrow\!(R \times Id); \langle \lambda(c', u) \cdot u\rangle \\
&\equiv \left\{\text{definitions}\right\} \\
&\quad \mathbf{create\ var}\ c'.C'(e); S\!\downarrow\!(R \times Id); \mathbf{end}
\end{aligned}
$$

$\square$

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 $d_i : \Delta_i, u : \Phi$, for any relation $R : \Sigma' \leftrightarrow \Sigma$ and any input argument $g_i : \Gamma_i$,*

$$
\begin{aligned}
&M_i \sqsubseteq_R M_i' \;\Rightarrow \\
&\quad (\mathbf{var}\ c, d_i, u \cdot c.Meth_i(g_i, d_i))\!\downarrow\!(R \times Id) \;\sqsubseteq\; (\mathbf{var}\ c', d_i, u \cdot c'.Meth_i(g_i, d_i))
\end{aligned}
$$

Proof We prove the goal by assuming the antecedent and deriving the right-hand side from the left-hand side as follows:

$$
\begin{array}{ll}
& (\mathbf{var}\ c, d_i, u \cdot c.Meth_i(g_i, d_i)) \downarrow (R \times Id) \\
\equiv & \left\{ \text{definition of method invocation} \right\} \\
& (\mathbf{var}\ c, d_i, u \cdot \mathbf{begin}\ (\mathbf{var}\ (self, x_i, y_i), c, d_i, u \cdot self = c \wedge x_i = g_i); \\
& M_i \times \mathbf{skip}; c, d_i := self, y_i; \mathbf{end}) \downarrow (R \times Id) \\
\equiv & \left\{ \text{definitions} \right\} \\
& \{R \times Id\}; [\lambda(c, d_i, u) \cdot \lambda((self, x_i, y_i), c', d_i', u') \cdot \\
& \quad self = c \wedge x_i = g_i \wedge (c, d_i, u) = (c', d_i', u')]; \\
& M_i \times \mathbf{skip}; \\
& \langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u) \rangle; \\
& \langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u) \rangle; [(R \times Id)^{-1}] \\
\equiv & \left\{ \begin{array}{l} \text{demonic and angelic updates of a functional relation are equal,} \\ \text{distribution of sequential composition through angelic updates} \end{array} \right\} \\
& \{R \times Id; \lambda(c, d_i, u) \cdot \lambda((self, x_i, y_i), c', d_i', u') \cdot \\
& \quad self = c \wedge x_i = g_i \wedge (c, d_i, u) = (c', d_i', u')\}; \\
& M_i \times \mathbf{skip}; \\
& \langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u) \rangle; \\
& \langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u) \rangle; [(R \times Id)^{-1}] \\
\sqsubseteq & \left\{ \text{definition of relational composition, logic} \right\} \\
& \{\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot \\
& \quad self' = c' \wedge x_i' = \iota_{\Gamma_i}\, g_i \wedge (c', d_i, u) = (c'', d_i', u'); \\
& (R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|) \times R \times Id\}; \\
& M_i \times \mathbf{skip}; \\
& \langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u) \rangle; \\
& \langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u) \rangle; [(R \times Id)^{-1}] \\
\equiv & \left\{ \begin{array}{l} \text{distribution of sequential composition through angelic updates,} \\ \text{demonic and angelic updates of a functional relation are equal} \end{array} \right\} \\
& [\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot \\
& \quad self' = c' \wedge x_i' = \iota_{\Gamma_i}\, g_i \wedge (c', d_i, u) = (c'', d_i', u')]; \\
& \{(R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|) \times R \times Id\}; \\
& M_i \times \mathbf{skip}; \\
& \langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u) \rangle; \\
& \langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u) \rangle; [(R \times Id)^{-1}] \\
\equiv & \left\{ \begin{array}{l} \text{product distribution through sequential composition} \\ \text{rules (e), (c) then } \mathbf{skip} \text{ is unit of sequential composition} \end{array} \right\}
\end{array}
$$

$$
\begin{aligned}
&[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot \\
&\qquad self' = c' \wedge x_i' = \iota_{\Gamma_i}\, g_i \wedge (c', d_i, u) = (c'', d_i', u')]; \\
&\{(R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|)\} \times \mathbf{skip}; \\
&(M_i; \mathbf{skip}) \times (\mathbf{skip}; \{R \times Id\}); \\
&\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u)\rangle; \\
&\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}]
\end{aligned}
$$

$$
\sqsubseteq \left\{ \begin{array}{l}\text{product distribution through sequential composition} \\ \text{rules (a) then (b)}\end{array} \right\}
$$

$$
\begin{aligned}
&[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot \\
&\qquad self' = c' \wedge x_i' = \iota_{\Gamma_i}\, g_i \wedge (c', d_i, u) = (c'', d_i', u')]; \\
&(\{R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|\}; M_i) \times \mathbf{skip}; \{Id \times R \times Id\}); \\
&\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u)\rangle; \\
&\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}]
\end{aligned}
$$

$$
\sqsubseteq \left\{ \text{assumption, using general rule } S{\downarrow}R \sqsubseteq S' \;=\; \{R\}; S \sqsubseteq S'; \{R\} \right\}
$$

$$
\begin{aligned}
&[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot \\
&\qquad self' = c' \wedge x_i' = \iota_{\Gamma_i}\, g_i \wedge (c', d_i, u) = (c'', d_i', u')]; \\
&(M_i'; \{R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|\}) \times \mathbf{skip}; \{Id \times R \times Id\}); \\
&\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u)\rangle; \\
&\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}]
\end{aligned}
$$

$$
\equiv \left\{ \text{product distribution through sequential composition rule (b), then (e)} \right\}
$$

$$
\begin{aligned}
&[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot \\
&\qquad self' = c' \wedge x_i' = \iota_{\Gamma_i}\, g_i \wedge (c', d_i, u) = (c'', d_i', u')]; \\
&M_i' \times \mathbf{skip}; \{(R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|) \times R \times Id\}; \\
&\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot ((self, x_i, y_i), self, y_i, u)\rangle; \\
&\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}]
\end{aligned}
$$

$$
\sqsubseteq \left\{ \begin{array}{l}\text{definition of sequential composition,} \\ \text{definitions of angelic and functional updates, logic}\end{array} \right\}
$$

$$
\begin{aligned}
&[\lambda(c', d_i, u) \cdot \lambda((self', x_i', y_i'), c'', d_i', u') \cdot \\
&\qquad self' = c' \wedge x_i' = \iota_{\Gamma_i}\, g_i \wedge (c', d_i, u) = (c'', d_i', u')]; \\
&M_i' \times \mathbf{skip}; \\
&\langle \lambda((self', x_i', y_i'), c', d_i, u) \cdot ((self', x_i', y_i'), self', \iota_{\Delta_i'}\, y_i, u)\rangle; \\
&\{(R \times \pi_{\Gamma_i} \times |\iota_{\Delta_i'}|) \times R \times Id\}; \\
&\langle \lambda((self, x_i, y_i), c, d_i, u) \cdot (c, d_i, u)\rangle; [(R \times Id)^{-1}]
\end{aligned}
$$

$$
\sqsubseteq \left\{ \begin{array}{l}\text{definition of sequential composition,} \\ \text{definitions of angelic and functional updates, logic}\end{array} \right\}
$$

$$
\begin{array}{l}
\quad [\lambda(c', d_i, u) \cdot \lambda((\mathit{self}', x_i', y_i'), c'', d_i', u') \cdot \\
\qquad \mathit{self}' = c' \wedge x_i' = \iota_{\Gamma_i}\, g_i \wedge (c', d_i, u) = (c'', d_i', u')]; \\
\quad M_i' \times \mathbf{skip}; \\
\quad \langle \lambda((\mathit{self}', x_i', y_i'), c', d_i, u) \cdot ((\mathit{self}', x_i', y_i'), \mathit{self}', \iota_{\Delta_i'}\, y_i, u)\rangle; \\
\quad \langle \lambda((\mathit{self}', x_i', y_i'), c', d_i, u) \cdot (c', d_i, u)\rangle; \{R \times Id\}; [(R \times Id)^{-1}] \\
\sqsubseteq \left\{ \text{general rule } \{R\}; [R^{-1}] \ \sqsubseteq\ \mathbf{skip} \right\} \\
\quad [\lambda(c', d_i, u) \cdot \lambda((\mathit{self}', x_i', y_i'), c'', d_i', u') \cdot \\
\qquad \mathit{self}' = c' \wedge x_i' = \iota_{\Gamma_i}\, g_i \wedge (c', d_i, u) = (c'', d_i', u')]; \\
\quad M_i' \times \mathbf{skip}; \\
\quad \langle \lambda((\mathit{self}', x_i', y_i'), c', d_i, u) \cdot ((\mathit{self}', x_i', y_i'), \mathit{self}', \iota_{\Delta_i'}\, y_i, u)\rangle; \\
\quad \langle \lambda((\mathit{self}', x_i', y_i'), c', d_i, u) \cdot (c', d_i, u)\rangle \\
\equiv \left\{ \text{definition of method invocation} \right\} \\
\quad (\mathbf{var}\ c', d_i, u \cdot c'.Meth_i(g_i, d_i))
\end{array}
$$

$\square$

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.abo.fi

University of Turku
- **Department of Mathematical Sciences**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**