Formal Semantics of Inheritance and Object Substitutability

Ralph Back, Leonid Mikhajlov, Joakim von Wright

Turku Centre for Computer Science
TUCS Technical Report No 337

ISBN 952-12-0637-3
ISSN 1239-1891

Abstract

In this paper we study the problem of semantic substitutability of objects. First we give semantics of classes, objects and inheritance in the presence of dynamic binding. For this purpose we extend Cook and Palsberg's denotational semantics of stateless classes and inheritance by adding state. We build our theory on the basis of the refinement calculus, which is a logic framework for reasoning about conformance of programs to their specifications and program refinement. We derive class refinement from the notion of abstract data type refinement as a criterion of object substitutability. We illustrate our model of classes and objects with an example and show how refinement between classes can be proved in practice.

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

TUCS Research Group
Programming Methodology Research Group

1 Introduction

In recent years object-oriented programming methodologies have gained recognition as the industrial strength approach. However, research on foundations of object-oriented languages is still active.

Providing solid foundations for object-oriented constructs becomes particularly important in view of an emerging component market. Developers of a component should be able to create their components independently, relying exclusively on the specifications of the other components. Strongly typed object-oriented languages provide for syntactic compatibility of objects. However, this is insufficient to guarantee the behavioral compatibility of components. The behavioral compatibility can be established provided that every component correctly implements its specification. Thus a methodology for verifying correctness of a component with respect to the specification it implements is needed.

This objective requires giving formal semantics to OO programming language constructs. OO languages have been traditionally described by method-lookup semantics [14]. Cook and Palsberg's denotational semantics of stateless classes and inheritance [10] explains their essence much more comprehensively.

In this paper we give semantics of classes, objects and inheritance in the presence of dynamic binding. For this purpose we extend Cook and Palsberg's semantics of classes with state. We build our theory on the basis of *refinement calculus* [3, 23] in the higher-order logic formulation [8]. The refinement calculus is well suited for describing object-oriented programs, because it allows us to specify classes at various abstraction levels. The language of the refinement calculus does not differentiate between specifications and executable statements. This permits us to describe all development stages in a uniform manner.

We express class refinement in terms of the well established theory of data refinement. As the theory of data refinement was shown to be sound and complete [13], our definition of class refinement inherits these properties.

We present a comprehensive example demonstrating how our theory can be applied in practice.

2 Refinement Calculus Basics

This section is based on the work by Back and von Wright as presented in [3, 5, 6, 8]. The behavior of a program statement can be characterized by Dijkstra's weakest precondition predicate transformer [11]. For a statement S and a predicate p, the weakest precondition $wp(S, p)$ is such that the statement S terminates in a state satisfying the postcondition p. Since the relation between pre- and postconditions is all we are interested in for a statement, we can identify the statement with a function mapping postconditions to preconditions.

The predicates over a state space (type) Σ are the functions from Σ to *Bool*, denoted $\mathcal{P}\Sigma$. The relations from Σ to Γ are functions from Σ to a predicate (set of values) over Γ, denoted by $\Sigma \leftrightarrow \Gamma$. The predicate transformers from Σ to Γ are the functions mapping predicates over Γ to predicates over Σ, denoted $\Sigma \mapsto \Gamma$ (note the reversion of the direction):

$$\begin{array}{lcl} \mathcal{P}\Sigma & \mathrel{\widehat{=}} & \Sigma \to \mathit{Bool} \\ \Sigma \leftrightarrow \Gamma & \mathrel{\widehat{=}} & \Sigma \to \mathcal{P}\Gamma \\ \Sigma \mapsto \Gamma & \mathrel{\widehat{=}} & \mathcal{P}\Gamma \to \mathcal{P}\Sigma \end{array}$$

The entailment ordering $p \subseteq q$ on predicates $p, q : \mathcal{P}\Sigma$ is defined by universal implication:

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

The predicates *true* and *false* over Σ map every $\sigma : \Sigma$ to the boolean values T and F, respectively. Conjunction $p \wedge q$, disjunction $p \vee q$, and negation $\neg p$ are all defined by pointwise extension of the corresponding operations on *Bool*.

The cartesian product of state spaces Σ_1 and Σ_2 is written $\Sigma_1 \times \Sigma_2$. The cartesian product of state spaces associates to the right. For relations $P_1 : \Sigma_1 \leftrightarrow \Gamma_1$ and $P_2 : \Sigma_2 \leftrightarrow \Gamma_2$, their *product* $P_1 \times P_2$, is a relation of type $(\Sigma_1 \times \Sigma_2) \leftrightarrow (\Gamma_1 \times \Gamma_2)$, where for $\sigma_1 : \Sigma_1$, $\sigma_2 : \Sigma_2$, $\gamma_1 : \Gamma_1$ and $\gamma_2 : \Gamma_2$, we have:

$$(P_1 \times P_2).(\sigma_1, \sigma_2).(\gamma_1, \gamma_2) \ \mathrel{\widehat{=}}\ (P_1.\sigma_1.\gamma_1) \ \wedge\ (P_2.\sigma_2.\gamma_2)$$

Product operators combine predicates 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

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

A predicate transformer $S : \Sigma \mapsto \Gamma$ is said to be *monotonic* if for all predicates p and q, $p \subseteq q$ implies $S.p \subseteq S.q$. Statements from Σ to Γ are identified with monotonic predicate transformers from Σ to Γ. Statements of this kind may be concrete, i.e. executable, or abstract, i.e. specifications, and may have different initial and final state spaces.

The sequential composition of statements $S : \Sigma \mapsto \Gamma$ and $T : \Gamma \mapsto \Delta$ is modeled by their functional composition. Let q be a predicate over Δ, then

$$(S;T).q \ \mathrel{\widehat{=}}\ S.(T.q).$$

Meet of (similarly-typed) predicate transformers is defined pointwise:

$$(S \sqcap T).q \ \mathrel{\widehat{=}}\ (S.q \ \wedge\ T.q)$$

Meet of statements models *nondeterministic choice* between executing statements S and T. It is required that both alternatives establish the postcondition. 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:

$$(S_1 \times S_2).\, q \;\hat{=}\; (\vee\, q_1, q_2 \,|\, q_1 \times q_2 \subseteq q \cdot S_1.\, q_1 \times S_2.\, q_2)$$

The **abort** statement does not guarantee any outcome or termination, therefore, it maps every postcondition to *false*. The **magic** statement is *miraculous*, since it is always guaranteed to establish any postcondition. The **skip** statement leaves the state unchanged. We define for any state space Σ and $q : \mathcal{P}\Sigma$

$$\begin{array}{lcl} \textbf{abort}.\, q & \hat{=} & \mathit{false} \\ \textbf{magic}.\, q & \hat{=} & \mathit{true} \\ \textbf{skip}.\, q & \hat{=} & q. \end{array}$$

For a predicate $p : \mathcal{P}\Gamma$, the assertion $\{p\}$ behaves as **abort** if p does not hold, and as **skip** otherwise. The *guard* statement $[p]$ behaves as **skip** if p holds, otherwise it behaves as **magic**. Let q be a predicate over Γ, then

$$\begin{array}{lcl} \{p\}.\, q & \hat{=} & p \wedge q \\ [p].\, q & \hat{=} & p \Rightarrow q \end{array}$$

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**.

Following [7], 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

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

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.

To enforce variable 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.

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:

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

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

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

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.

For statements S and T of the same type and a predicate p, the conditional is defined by:

$$\mathbf{if}\ p\ \mathbf{then}\ S\ \mathbf{else}\ T\ \mathbf{fi} \;\;\widehat{=}\;\; ([p]; S) \sqcap ([\neg p]; S)$$

The refinement ordering $S \sqsubseteq T$, read S *is refined by* T, on statements $S, T : \Sigma \mapsto \Gamma$ is defined as a universal entailment:

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

An iteration statement is defined by the least fixed point of a function F with respect to the refinement ordering:

$$\begin{array}{l} \mathbf{while}\ p\ \mathbf{do}\ S\ \mathbf{od} \;\;\widehat{=}\;\; \mu\, F, \\ \text{where } F.\, X = \mathbf{if}\ p\ \mathbf{then}\ S; X\ \mathbf{else\ skip\ fi} \end{array}$$

According to the theorem of Knaster-Tarski [26], a monotonic function has a unique least fixed point in a complete lattice. Statements form a complete lattice with the refinement ordering $\sqsubseteq$, and the statement **if** $p(x)$ **then** $S; X$ **else skip fi** is monotonic with respect to the refinement ordering $\sqsubseteq$ in X, therefore, $\mu\ F$ exists and is unique. Intuitively, defining iteration in this way implies that a non-terminating loop behaves as **abort**.

A *block* construct allows temporary adding a new *local* state component to the present *global* state. Block beginning and end are modeled by demonic and functional updates respectively:

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

Here **begin** introduces a new variable and initializing it to an arbitrary value, and **end** removes this variable. The construct **begin** $x \cdot S;$ **end** states that the program variable x, introduced by **begin** into the state space stored in the global variable u, is local to the statement S. This is a syntactic convention representing the statement **begin**; (**var** $x, u \cdot S$); **end**.

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

$$(\textbf{var}\ u \cdot [x := x' \,|\, x' \geq 0]; y := x) \;=\; [\textbf{var}\ u \cdot x := x' \,|\, x' \geq 0]; (\textbf{var}\ u \cdot y := x)$$

In general, different state spaces can be coerced using *encoding* and *decoding* operators [9, 17]. Statements S and S' operating on state spaces Σ and Σ' respectively can be combined using a relation $R : \Sigma' \leftrightarrow \Sigma$ which, when lifted to predicate transformers, gives the update statements $\{R\} : \Sigma' \mapsto \Sigma$ and $[R^{-1}] : \Sigma \mapsto \Sigma'$. These statements are used to define encoding $\downarrow$ and decoding $\uparrow$ operators as follows:

$$\begin{array}{lcl} S\!\downarrow\!R & \mathrel{\widehat{=}} & \{R\}; S; [R^{-1}] \\ S'\!\uparrow\!R & \mathrel{\widehat{=}} & [R^{-1}]; S'; \{R\} \end{array}$$

Thus, we have that the statements $S\downarrow R$ and $S'\uparrow R$ operate on the state spaces Σ' and Σ respectively. Encoding and decoding operators can be extended to work with tuples of statements:

$$\begin{array}{lcl} (S_1, ..., S_n)\!\downarrow\!R & \mathrel{\widehat{=}} & (S_1\!\downarrow\!R, ..., S_n\!\downarrow\!R) \\ (S'_1, ..., S'_n)\!\uparrow\!R & \mathrel{\widehat{=}} & (S'_1\!\uparrow\!R, ..., S'_n\!\uparrow\!R) \end{array}$$

The encoding and decoding operators are left associative and have higher precedence than function application.

When proving data refinement in practice, we have to reason about the effect of angelic update $\{R\} : \Sigma' \mapsto \Sigma$ and demonic update $[R^{-1}] : \Sigma \mapsto \Sigma'$ in states carried in some program variables $x : \Sigma'$ and $y : \Sigma$ respectively. Intuitively, the statement $\{R\}$, transforming one state space to another, would first introduce a new variable $y : \Sigma$, assign it a value y', satisfying $R.x.y'$, and then eliminate the redundant variable x. This intuition is supported by the following rule:

$$\{R\} \;=\; \mathbf{begin}\, y \cdot \{\mathbf{var}\ y, x \cdot y := y' \mid R.x.y'\}; \langle sw \rangle; \mathbf{end}$$

Similarly, for changing the state space in the other direction, we have a rule

$$[R^{-1}] \;=\; \mathbf{begin}\, x \cdot [\mathbf{var}\ x, y \cdot x := x' \mid R^{-1}.y.x']; \langle sw \rangle; \mathbf{end}$$

where sw is the function swapping two state components, such that $sw \;=\; \lambda x, y \cdot y, x$.

3 Formalization of Classes and Modifiers

We model classes as self-referential structures as proposed by Cook and Palsberg in [10]. However, unlike in their model, classes have internal state in our formalization. This model of classes and modifiers was first developed in [21]. Unlike in [21], in this paper a modifier can introduce additional state, so that the resulting class works on the inherited and the added states. Also the modifier can have more methods than the base class it is to be applied to. Introducing additional state and new methods in modifiers required making changes that propagated throughout the formal model.

For simplicity we model method parameters by global variables that both methods of a class and its clients can access. We precede a formal method parameter with a keyword **val** to indicate that the method only reads the value of this parameter without changing it. Similarly we precede the formal parameter with **res** to indicate that the method returns a value in this parameter.

In practice a call of a method m_j by a method m_i of the same class has the form *self*.m_j. Due to inheritance and dynamic binding such a call can get redirected to a definition of m_j in an extension class. We model the method m_i as a function of a called method. As in general, a method of a class may invoke all the other methods of the same class, in the case of n methods we have

$$m_i \;=\; \lambda(x_1, ..., x_n).c_i$$

where c_i is a statement representing the body of the method m_i. Accordingly, we define a class C by

$$C \;=\; (c_0, cm), \text{where } cm \;=\; \lambda \mathit{self} \cdot (c_1, ..., c_n),$$

and where $c_0 : \Sigma$ is an initial value of the internal class state, and *self* is an abbreviation for the tuple $x_1, ..., x_n$. We assume that cm is monotonic in the *self* parameter with respect to refinement relation.

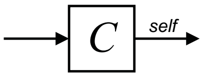

Figure 1: Illustration of a class

Let us consider now the typing of C. The type of its instance variables is taken to be Σ. As we have stated before, some global variables $d_1 : \Delta_1, ..., d_m : \Delta_m$ are used for parameter passing. Due to dynamic binding, methods of C operate not only on the state space $\Sigma \times \Delta$, where $\Delta = \Delta_1 \times ... \times \Delta_m$, but also on the state space of some modifier whose type is unknown until the modifier is applied.[1] If we introduce an abbreviation $\Pi_n(\Psi)$ to stand for $(\Psi \mapsto \Psi) \times ... \times (\Psi \mapsto \Psi)$ with n components in the cross product, we can write the type of cm as:

$$\Pi_n(\alpha \times \Sigma \times \Delta) \to \Pi_n(\alpha \times \Sigma \times \Delta)$$

where α is a type variable. It is instantiated with the type of modifier instance variables at modifier application. We assume that methods of the base class C have a structure **skip** $\times\, S$, where **skip** is executed on the α component of the state and S is executed on the component $\Sigma \times \Delta$.

We declare the class C, defined in this way, as follows:

$$C \;=\; \textbf{class}\; c := c_0,\; m_1 \;\widehat{=}\; c_1, ..., m_n \;\widehat{=}\; c_n \;\textbf{end}$$

A class can be depicted as in Fig. 1. The incoming arrow represents calls to the class C, the outgoing arrow stands for self-calls of C.

In our model classes are used as templates for creating objects. Objects have all self-calls resolved with the methods of the same object. Modeling this formally amounts to taking the least fixed point of the function cm. For tuples of statements $(c_1, .., c_n)$ and $(c'_1, ..., c'_n)$, where c_i and c'_i are of the same type, the refinement ordering is defined elementwise:

$$(c_1, ..., c_n) \sqsubseteq (c'_1, ..., c'_n) \;\widehat{=}\; c_1 \sqsubseteq c'_1 \;\wedge\; ... \;\wedge\; c_n \sqsubseteq c'_n$$

Statement tuples form a complete lattice with the above refinement ordering. Also cm is required to be monotonic in its *self* argument. These two conditions are sufficient to guarantee that the least fixed point of the function cm exists and is unique. If we model objects as tuples of instance variable values and method bodies, we can define the operation of creating an instance of a class as follows:

$$\mathit{create}\; C \;\widehat{=}\; (c_0, (\mu\; cm)\!\uparrow\!\pi_2),$$

[1]This issue is discussed in detail further in the paper when we consider inheritance.

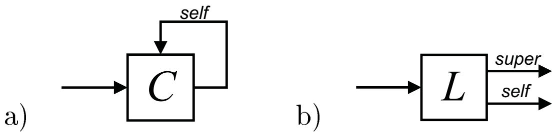

Figure 2: a) Illustration of a class instance creation, b) Illustration of modifiers

where $\pi_2(x' : \alpha, y' : \beta)(y : \beta) \; = \; y' = y$ is a projection relation, projecting away α component of the state. Thus methods of the resulting object have the type $\Pi_n(\Sigma \times \Delta)$. Fig. 2 a) illustrates creating an instance of the class C.

For modeling single inheritance we adopt the notion of modifiers as proposed by Cook and Palsberg in [10].[2] This model of inheritance was proved to correspond to the form of inheritance used in object-oriented systems. We extend this model of inheritance by allowing modifiers to have internal state.

Assume that we have a base class C and an extension class E inheriting from it. We say that E is equivalent to $(L \ \mathbf{mod} \ C)^3$, where L corresponds to the extending part of the definition of E, and the operator **mod** combines L with the inherited part C. We refer to L as a *modifier* [27].

It suffices to consider only those modifiers which redefine all methods of the base class, and introduce some new ones. In case some method should remain unchanged, the corresponding method of the modifier calls the former via *super*.

A modifier

$$L \; = \; \mathbf{modifier} \; l := l_0, \; m_1 \mathrel{\widehat{=}} l_1, ..., m_n \mathrel{\widehat{=}} l_n \; \mathbf{end}$$

is modeled by a pair

$$L \; = \; (l_0, \; lm), \text{where } lm \; = \; \lambda \mathit{self} \cdot \lambda \mathit{super} \cdot (l_1, ..., l_n, l'_1, ..., l'_k),$$

where $l_0 : \Gamma$ are initial values of new instance variables l and lm is a function, bounded variables *self* and *super* are abbreviations of the tuples $x_1, ..., x_n, x'_1, ..., x'_k$ and $y_1, ..., y_n$ respectively, $l_1, ..., l_n$ are the overriding methods and $l'_1, ..., l'_k$ are the new methods. We assume that L is monotonic in both arguments. See Fig. 2 b) for an illustration of modifiers. As with the class diagram, the incoming arrow represents calls to methods of the modifier, whereas outgoing arrows stand for self and super-calls of the modifier.

[2]In their paper modifiers are referred to as wrappers. We prefer the term modifier, because the term wrapper is usually used in the context of object aggregation.

[3]We read **mod** as modifies.

Under the condition that the signatures of overriding methods of the modifier match those of the base class, the modifier can be applied to an arbitrary base class. We make a restriction that the methods of the modifier are not allowed to access the state of the base class directly, but only by making super-calls. As was pointed out by Alan Snyder in [25], "Because the instance variables are accessible to clients of the class, they are (implicitly) part of the contract between the designer of the class and the designers of descendant classes. Thus, the freedom of the designer to change the implementation of a class is reduced". Accessing the base class state from the modifier directly is recognized as bad programming practice.

As the state space of the base class is unknown until modifier application, we say that the methods of the modifier L operate on the state space $\alpha' \times \beta \times \Gamma \times \Delta \times \Phi$, where β is a type variable to be instantiated with the type of the internal state of the base class while modifier application, Δ is a type of the state component representing all parameters of the overriding methods of the modifier, and Φ is a type of the state component representing all parameters of the new methods of the modifier. The role of type variable α' is explained after we define creation of new classes by applying modifiers. Hence, the type of lm is as follows:

$$\Pi_{n+k}(\alpha' \times \beta \times \Gamma \times \Delta \times \Phi) \rightarrow \Pi_n(\alpha' \times \beta \times \Gamma \times \Delta \times \Phi) \rightarrow \Pi_{n+k}(\alpha' \times \beta \times \Gamma \times \Delta \times \Phi)$$

We assume that methods of the modifier L have a structure $\mathbf{skip} \times (\mathbf{skip} \times S)$, where **skip** is executed on the α' and β components of the state and S is executed on the component $\Gamma \times \Delta \times \Phi$.

The operator **mod** can be used for creating an extension class from the base class C and the modifier L:

$$L \ \mathbf{mod} \ C \ \widehat{=} \ ((l_0, c_0), \ \lambda self \cdot \ (lm.\, self{\uparrow}P.\, (cm.\, self{\uparrow}Q){\downarrow}Q{\uparrow}P){\downarrow}P)$$

Here P and Q are defined as follows:

$$\begin{aligned}
&P.\,(a', (\gamma', \sigma'), (\delta', \phi')).\,(a, \sigma, \gamma, \delta, \phi) \ = \\
&\qquad\qquad a' = a \ \wedge \ \gamma' = \gamma \ \wedge \ \sigma' = \sigma \ \wedge \ \delta' = \delta \ \wedge \ \phi' = \phi \\
&Q.\,(a', (\gamma', \sigma'), (\delta', \phi')).\,((\phi, \gamma), (\sigma, \delta)) \ = \\
&\qquad\qquad \gamma' = \gamma \ \wedge \ \sigma' = \sigma \ \wedge \ \delta' = \delta \ \wedge \ \phi' = \phi
\end{aligned}$$

In the definition above the encodings and decodings are only necessary to adjust the state spaces of C and L. Figure 3 illustrates application of the modifier L to the base class C to construct class (L **mod** C). Note, that in the resulting class self-calls of C become down-calls. Figure 4 illustrates how the state spaces change while abstractly coercing *self* with Q.

For conciseness we have written out the type of *self* as a statement, while in reality it is a tuple of statements. Note how the component α' is first introduced and initialized with an arbitrary value, and then removed from

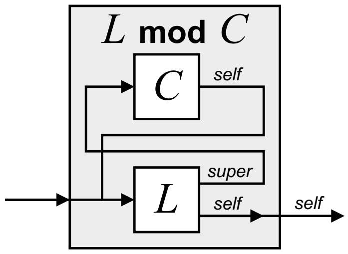

Figure 3: Illustration of the operator **mod**

the state space while decoding. However, as methods of (L **mod** C) only skip on the α' state component, such behavior does not constitute a problem.

Application of the modifier L instantiates its type variable β with the type Σ of the base class C. Simultaneously, the type variable α of C is instantiated with the state space $(\Phi \times \Gamma)$ of the modifier L. Hence, after the corresponding state space rearrangement, the constructed class (L **mod** C) has methods operating on $\alpha' \times ((\Gamma \times \Sigma) \times (\Delta \times \Phi))$, where α' is a type variable to be instantiated in the next modifier application, $\Gamma \times \Sigma$ is the type of instance variables of (L **mod** C), and $\Delta \times \Phi$ is the type of all parameters of all methods of this class. For an illustration of instantiation of type variables see Fig. 5.

At this point an appropriate question to ask is why all these type variables are necessary at all. Why not choose the state space of the base class to be just $\Sigma \times \Delta$ and the state space of the modifier to be $\Gamma \times \Delta \times \Phi$? In this case we would need the coercing relations to be

$$\begin{aligned} P.\,((\gamma' : \Gamma, \sigma' : \Sigma), (\delta' : \Delta, \phi' : \Phi)).\,(\gamma, (\delta, \phi)) \;&=\; \sigma' = \sigma \;\wedge\; \delta' = \delta \;\wedge\; \phi' = \phi \\ Q.\,(a' : \alpha', ((\gamma' : \Gamma, \sigma' : \Sigma), (\delta' : \Delta, \phi' : \Phi)).\,(\sigma, \delta) \;&=\; \sigma' = \sigma \;\wedge\; \delta' = \delta \end{aligned}$$

Abstractly coercing *self* with these relations would lead to introducing and removing the state components Σ (in case of P) and Γ and Φ (in case of Q) in the manner described above. While this is acceptable for the void α' component, it is not acceptable for Σ, Γ and Φ because values of these state components will be lost and reintroduced arbitrarily during encodings and decodings, which would lead to chaotic behavior of the resulting class.

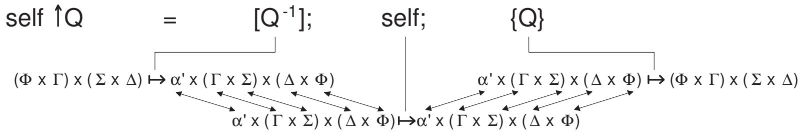

Figure 4: Illustration of adjusting state spaces by decoding

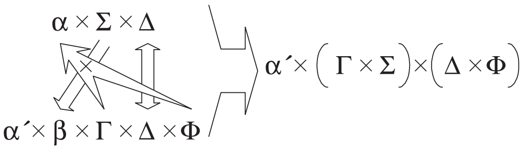

Figure 5: Illustration of type variables instantiation

4 Refinement on Classes

We can express class refinement in terms of a slight modification of the refinement on abstract data types [16, 4, 12]. An abstract data type T can be represented in the form

$$T \;=\; (t_0, tp),$$

where $t_0 : \Sigma$ is an initial value of the internal state, and tp is a tuple of procedures modifying this internal state. The tuple of procedures is of type $\Pi_n(\Sigma \times \Delta)$, where Δ is the state space component representing all parameters of all procedures.

We say that the abstract data type $T = (t_0, tp)$ is data refined by an abstract data type $T' = (t'_0 : \Sigma', tp' : \Pi_n(\Sigma' \times \Delta \times \Phi))$ via a relation $R : \Sigma' \leftrightarrow \Sigma$, if the initialization establishes R and all procedures preserve $R \times \pi_1$:

$$\begin{array}{l} T \sqsubseteq_R T' \;\widehat{=}\; (R.\, t'_0.\, t_0) \;\wedge\; (tp \sqsubseteq_{R \times \pi_1} tp'), \text{ where} \\ \pi_1.\, (x' : \alpha, y' : \beta).\, (x : \alpha) \;=\; (x' = x) \end{array}$$

In this definition we take into account the fact that state space components Δ and Φ model method parameters. Values in these state space components cannot be related to each other or to the internal states of classes C and C' by any invariant. Therefore, we can connect the state components representing procedure parameters of T' and T using π_1, which discards the component Φ of T' and equates the components Δ of T' and T.

Now class refinement can be defined as follows. Let $\mathcal{P}$ be a projection function
$\mathcal{P} \,\widehat{=}\, \lambda(y, (x_1, ..., x_n, x'_1, ..., x'_k)) \cdot (y, (x_1, ..., x_n))$ discarding extra methods of an object. Let C be as defined in the previous section and $C' = (c'_0, cm')$, with $cm'.\ \mathit{self} = (c'_1, ..., c'_n, c''_1, ..., c''_k)$, then

$$C \sqsubseteq C' \;\widehat{=}\; \exists R : \Sigma' \leftrightarrow \Sigma \cdot (\mathit{create}\ C) \sqsubseteq_R \mathcal{P}(\mathit{create}\ C')$$

This definition states that the class C' refines the class C if there exists a relation R, coercing their corresponding internal state, such that the objects these classes generate are in the data refinement relation with respect R. As objects are effectively abstract data types, we reduce the definition of class refinement to the definition of the refinement on abstract data types which was shown to be sound and complete [13]. Our definition of class refinement ensures that all objects of the refining class are safely substitutable for objects of the refined class. This notion of class refinement allows the instance variables of C' extend those of C or to be completely different. The refinement relation can be also applied to a pair of abstract or concrete classes.

Verifying substitutability of objects generated by a base class, and a derived class having additional methods, is usually complicated by the presence of the subsumption mechanism. Due to subsumption, different clients can simultaneously work with an object through different interfaces. This can lead to a havoc, because a client can find the object in a state which is unreachable through the interface this client have used. As in this paper we are not modeling the subsumption mechanism, this problem does not apply. In fact, new methods in the derived class can even be aborting, under the condition that they are not self-called by the overridden methods. As in our definition of class refinement new methods are projected away after calculating an object of the class, the behavior of the new methods cannot influence object substitutability.

5 Example

In this section we consider an example illustrating all concepts presented above. In the example we make use of the standard notation for specifying sequences. Let us briefly review operators defined on sequences. Given a sequence, the operator **hd** returns its first element; the operator **tl** returns its tail subsequence; the operator $\#$ returns the cardinality of the given sequence; the operator $\in$, given an element and a sequence, returns *true* if this element can be found in the sequence and *false* otherwise; the operator $\frown$ concatenates the second given sequence to the end of the first one. Sequence elements can be accessed by their index.

A class *String* represents a string by a sequence of characters s, which is initialized to the empty sequence. The class *String* supports the following operations: *getS* accesses the sequence representation of the string, which is otherwise encapsulated within the class; *append* appends a character to the end of the string; *appendAll* concatenates the sequence with an incoming one; *cardinality* returns the number of characters in the sequence. Note how *appendAll* attaches all elements of the incoming sequence by self-calling the *append* method.

$$
\begin{array}{l}
\textit{String} \;=\; \textbf{class} \\
\quad s : \textit{seq of char} := \langle\rangle, \\
\quad \textit{getS}(\textbf{res}\ rs : \textit{seq of char}) \mathrel{\widehat{=}} rs := s, \\
\quad \textit{append}(\textbf{val}\ x : \textit{char}) \mathrel{\widehat{=}} s := s \frown \langle x \rangle, \\
\quad \textit{appendAll}(\textbf{val}\ ss : \textit{seq of char}) \mathrel{\widehat{=}} \\
\qquad \textbf{while}\ ss \neq \langle\ \rangle\ \textbf{do} \\
\qquad\quad \textit{self.append}(\textbf{hd}\ ss); \\
\qquad\quad ss := \textbf{tl}\ ss \\
\qquad \textbf{od}, \\
\quad \textit{cardinality}(\textbf{res}\ r : \textit{int}) \mathrel{\widehat{=}} r := \#s, \\
\quad \textit{isInString}(\textbf{val}\ wh : \textit{char}, \textbf{res}\ b : \textit{bool}) \mathrel{\widehat{=}} \\
\qquad b := wh \in ss \\
\textbf{end}
\end{array}
$$

A modifier *Counting* introduces a new instance variable $n : int$, initialized to zero, to contain the cardinality of the sequence. It overrides the *append* method to increment n and then makes a super-call to its base class implementation. It also overrides the *cardinality* method to return the value of n. The method *search* which given a character returns its index in the sequence if this character is in it, otherwise it returns zero.

$$
\begin{array}{l}
\textit{Counting} \;=\; \textbf{modifier} \\
\quad n : \textit{int} := 0, \\
\quad \textit{append}(\textbf{val}\ x : \textit{char}) \mathrel{\widehat{=}} n := n + 1; \textit{super.append}(x), \\
\quad \textit{cardinality}(\textbf{res}\ r : \textit{int}) \mathrel{\widehat{=}} r := n, \\
\quad \textit{isInString}(\textbf{val}\ wh : \textit{char}, \textbf{res}\ b : \textit{bool}) \mathrel{\widehat{=}} \\
\qquad \textbf{begin}\ i : \textit{int} \cdot \\
\qquad\quad \textit{self.search}(wh, i); \\
\qquad\quad b := \neg(i = 0) \\
\qquad \textbf{end}, \\
\quad \textit{search}(\textbf{val}\ w : \textit{char}, \textbf{res}\ ri : \textit{int}) \mathrel{\widehat{=}} \\
\qquad \textbf{begin}\ ss : \textit{seq of char} \cdot \\
\qquad\quad \textit{super.getS}(ss); \\
\qquad\quad [ri := ri' \mid \;\; w \in ss \;\wedge\; ss[ri'] = w \;\vee \\
\qquad\qquad\qquad\quad\;\; w \notin ss \;\wedge\; ri' = 0] \\
\qquad \textbf{end}, \\
\textbf{end}
\end{array}
$$

By applying the modifier *Counting* to the base class *String* we construct a class *CountingString*. Note that the resulting class maintains an invariant $\#s = n$, saying that the number of characters in the string is equal to n, initialization establishes this invariant and all methods preserve it.

$$
\begin{array}{l}
\mathit{CountingString} \;=\; \textbf{class} \\
\quad s : \mathit{seq\ of\ char} := \langle\rangle, \\
\quad n : \mathit{int} := 0, \\
\quad \mathit{getS}(\textbf{res}\ rs : \mathit{seq\ of\ char}) \mathrel{\widehat{=}} rs := s, \\
\quad \mathit{append}(\textbf{val}\ x : \mathit{char}) \mathrel{\widehat{=}} n := n + 1; s := s \frown \langle x \rangle, \\
\quad \mathit{appendAll}(\textbf{val}\ ss : \mathit{seq\ of\ char}) \mathrel{\widehat{=}} \\
\qquad \textbf{while}\ ss \neq \langle\ \rangle\ \textbf{do} \\
\qquad\quad \mathit{self.append}(\textbf{hd}\ ss); \\
\qquad\quad ss := \textbf{tl}\ ss \\
\qquad \textbf{od}, \\
\quad \mathit{cardinality}(\textbf{res}\ r : \mathit{int}) \mathrel{\widehat{=}} r := n, \\
\quad \mathit{isInString}(\textbf{val}\ wh : \mathit{char}, \textbf{res}\ b : \mathit{bool}) \mathrel{\widehat{=}} \\
\qquad \textbf{begin}\ i : \mathit{int}\ \cdot \\
\qquad\quad \mathit{self.search}(wh, i); \\
\qquad\quad b := \neg(i = 0) \\
\qquad \textbf{end}, \\
\quad \mathit{search}(\textbf{val}\ w : \mathit{char}, \textbf{res}\ ri : \mathit{int}) \mathrel{\widehat{=}} \\
\qquad \textbf{begin}\ ss : \mathit{seq\ of\ char}\ \cdot \\
\qquad\quad ss := s; \\
\qquad\quad [ri := ri' \mid \; w \in ss \;\wedge\; ss[ri'] = w \;\vee \\
\qquad\qquad\qquad\qquad w \notin ss \;\wedge\; ri' = 0] \\
\qquad \textbf{end} \\
\textbf{end}
\end{array}
$$

We can create instances of *String* and *CountingString* using the operation *create*. Let us compute the definitions of the resulting objects. For brevity we leave out the definitions of attributes that are not affected by the *create* operation.

$$
\begin{array}{l}
\mathit{String} \;=\; \textbf{object} \\
\quad \ldots \\
\quad \mathit{appendAll}(\textbf{val}\ ss : \mathit{seq\ of\ char}) \mathrel{\widehat{=}} \\
\qquad \textbf{while}\ ss \neq \langle\ \rangle\ \textbf{do} \\
\qquad\quad s := s \frown \langle \textbf{hd}\ ss \rangle; \\
\qquad\quad ss := \textbf{tl}\ ss \\
\qquad \textbf{od}, \\
\quad \ldots \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathit{CountingString} \;=\; \textbf{object} \\
\quad \ldots \\
\quad \mathit{appendAll}(\textbf{val}\ ss : \mathit{seq\ of\ char}) \mathrel{\widehat{=}} \\
\qquad \textbf{while}\ ss \neq \langle\ \rangle\ \textbf{do} \\
\qquad\quad n := n+1;\ s := s \frown \langle \textbf{hd}\ ss\rangle; \\
\qquad\quad ss := \textbf{tl}\ ss \\
\qquad \textbf{od}, \\
\quad \ldots \\
\quad \mathit{isInString}(\textbf{val}\ wh : \mathit{char}, \textbf{res}\ b : \mathit{bool}) \mathrel{\widehat{=}} \\
\qquad \textbf{begin}\ i : \mathit{int} \cdot \\
\qquad\quad \textbf{begin}\ ss : \mathit{seq\ of\ char} \cdot \\
\qquad\qquad ss := s; \\
\qquad\qquad [i := i' \mid \quad wh \in ss \ \wedge\ ss[i'] = wh \ \vee \\
\qquad\qquad\qquad\qquad wh \notin ss \ \wedge\ i' = 0] \\
\qquad\quad \textbf{end} \\
\qquad \textbf{end} \\
\quad \ldots \\
\textbf{end}
\end{array}
$$

Now let us demonstrate that class *CountingString* is in fact a refinement of the class *String*. To show that $\mathit{String} \sqsubseteq \mathit{CountingString}$ we prove a stronger goal that

$$(\mathit{create\ String}) \sqsubseteq_R \mathcal{P}(\mathit{create\ CountingString}),$$

where $R\ (s', n')\ s \;=\; s' = s \ \wedge\ n' = \#s$. It is easy to see that the initial values of the objects are connected by R. Thus we are left to show that corresponding methods are in the data refinement relation.

As it was mentioned before, we model method parameters as corresponding global variables that methods can access along with the class instance variables. Thus we have that all methods of *String* operate on the state partitioned by the variables s, rs, x, ss, wh, b, r, and methods of *CountingString* operate on the state partitioned by the variables s', n', rs', x', ss', wh', b', r', w', ri'. We use prime notation for the variables of *CountingString* to differentiate them from the corresponding variables of *String*.

Due to space limitations we only show refinement between the *cardinality* methods, as, in our opinion, these methods are both concise and illustrative. As some of the variables are not accessed by the *cardinality* methods we abbreviate them using lists of variables u, u', v', such that

$$
\begin{array}{lcl}
u & = & rs, x, ss, wh, b \\
u' & = & rs', x', ss', wh', b' \\
v' & = & w', ri'
\end{array}
$$

We also abbreviate tuples of variables (s, u, r) as y and $((s', n'), ((u', r'), v'))$ as x.

Therefore we need to prove the following goal:

$$\textbf{var}\ y \cdot r := \#s \sqsubseteq_{R\times\pi_1} \textbf{var}\ x \cdot r' := n'$$

In the proof we make use of the following *removed state component rule*:

$$\textbf{var}\ r, u \cdot r, u := r', u'; \langle sw\rangle; \textbf{end} \sqsubseteq \textbf{var}\ r, u \cdot r, u := r', u''; \langle sw\rangle; \textbf{end}$$

We prove our goal as follows:

$$
\begin{array}{l}
\quad \textbf{var}\ y \cdot r := \#s \sqsubseteq_{R\times\pi_1} \textbf{var}\ x \cdot r' := n' \\
= \{\textit{definition of the data refinement}\ \} \\
\quad \{R\times\pi_1\}; (\textbf{var}\ y \cdot r := \#s) \sqsubseteq (\textbf{var}\ x \cdot r' := n'); \{R\times\pi_1\} \\
= \{\textit{definition of the coercing angelic update}\} \\
\quad \textbf{begin}\, y \cdot \\
\quad\ \{\textbf{var}\ y, x \cdot s, u, r := \bar{s}, \bar{u}, \bar{r} \,|\, (R\times\pi_1).\,((s', n'), ((u', r'), v')).\,(\bar{s}, \bar{u}, \bar{r})\}; \langle sw\rangle; \\
\quad \textbf{end}; (\textbf{var}\ y \cdot r := \#s) \sqsubseteq \\
\quad\quad (\textbf{var}\ x \cdot r' := n'); \\
\quad\quad \textbf{begin}\, y \cdot \\
\quad\quad\ \{\textbf{var}\ y, x \cdot s, u, r := \bar{s}, \bar{u}, \bar{r} \,|\, (R\times\pi_1).\,((s', n'), ((u', r'), v')).\,(\bar{s}, \bar{u}, \bar{r})\}; \langle sw\rangle; \\
\quad\quad \textbf{end} \\
= \{\textit{moving assignments inside the blocks, definitions of relations}\} \\
\quad \textbf{begin}\, y \cdot \\
\quad\ \{\textbf{var}\ y, x \cdot s, u, r := \bar{s}, \bar{u}, \bar{r} \,|\, \bar{s} = s' \ \wedge\ n' = \#s' \ \wedge\ \bar{u} = u' \ \wedge\ \bar{r} = r'\}; \\
\quad\ (\textbf{var}\ y, x \cdot r := \#s); \langle sw\rangle; \\
\quad \textbf{end} \sqsubseteq \\
\quad\quad \textbf{begin}\, y \cdot (\textbf{var}\ y, x \cdot r' := n'); \\
\quad\quad\ \{\textbf{var}\ y, x \cdot s, u, r := \bar{s}, \bar{u}, \bar{r} \,|\, \bar{s} = s' \ \wedge\ n' = \#s' \ \wedge\ \bar{u} = u' \ \wedge\ \bar{r} = r'\}; \langle sw\rangle; \\
\quad\quad \textbf{end} \\
\Leftarrow \{\textit{lifting functions to relations, sequantial composition of relations, logic}\ \} \\
\quad \textbf{begin}\, y \cdot \\
\quad\ \{\textbf{var}\ y, x \cdot s, u, r, r' := \bar{s}, \bar{u}, \bar{r}, \bar{r'} \,|\, \bar{s} = s' \ \wedge\ n' = \#\bar{s} \ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \bar{u} = u' \ \wedge\ \bar{r} = n' \ \wedge\ \bar{r'} = r'\}; \langle sw\rangle; \\
\quad \textbf{end} \sqsubseteq \\
\quad\quad \textbf{begin}\, y \cdot \\
\quad\quad\ \{\textbf{var}\ y, x \cdot s, u, r, r' := \bar{s}, \bar{u}, \bar{r}, \bar{r'} \,|\, \bar{s} = s' \ \wedge\ n' = \#\bar{s} \ \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \bar{u} = u' \ \wedge\ \bar{r} = n' \ \wedge\ \bar{r'} = n'\}; \langle sw\rangle; \\
\quad\quad \textbf{end} \\
= \{\textit{removed state component rule}\} \\
\quad T
\end{array}
$$

6 Conclusion and Related Work

The research presented in this paper extends and elaborates the work on the formalization of classes and class refinement described in [21]. The models of classes, inheritance and class refinement have proved their worth in studying the inheritance reuse mechanism and, in particular, the fragile base class problem which occurs while maintaining object-oriented software systems relying on code inheritance [20]. As compared with [21], in this work we allow modifiers to introduce additional state, so that the resulting class works on the inherited and the added states. In addition, modifiers can not only have overriding methods, but also introduce extra ones.

We have encountered two interconnected research directions which are related to the presented research. The first research direction is in giving semantics to the object–oriented language constructs. Abadi and Leino in [1] develop a logic of object-oriented programs in the style of Hoare, prove its soundness, and discuss completeness issues. As many of the object-oriented languages were developed as an extension of some imperative language it appears natural to build the semantics of the object–oriented constructs by extending the semantics of imperative language constructs, as we do in this work. A similar approach also based on refinement calculus was reported in [22]. However, that paper focuses on different issues, for instance, it concentrates on modeling the subsumption mechanism, without considering self-referencing in the presence of dynamic binding. Naumann [24] defines the semantics of a simple Oberon-like programming language with similar specification constructs as here, also based on predicate transformers.

Hense in [15] gives a denotational semantics of an object-oriented programming language. His model provides for classes with state, self-referencing, and multiple inheritance. The latter is described using the wrappers of Cook and Palsberg [10]. However he does not consider substitutability of objects which is the main focus of this paper.

Substitutability of objects in various embodiments constitutes the second related research direction. The most well-known approach in the object–oriented community is that of "behavioral subtyping" of classes [2, 19]. This approach distinguishes between the specification of a class and its implementation. By having specification constructs as part of the (extended) programming language, this distinction becomes unnecessary, eliminating many proof obligations. Moreover, the well-established theory of data refinement allows for reuse of a large number of theoretical results and proof rules. Class refinement has also been studied in various extensions of the Z specification languages, e.g. [18], but only between class specifications and not implementations. Class refinement within refinement calculus was also defined in [22], but the fact that self-referential methods are not considered results in a different definition of class refinement.

A number of concepts have not been treated, for example, we did not consider subsumption and subtyping polymorphism. These issues constitute the topic of our future work.

Acknowledgments

We would like to thank Emil Sekerinski for valuable comments on earlier versions of this paper.

References

[1] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Theory and Practice of Software Development: Proceedings / TAPSOFT '97*, Lecture Notes in Computer Science 1214, pages 682–696. Springer, April 1997.

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

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

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

[5] R. J. R. Back and J. von Wright. Refinement calculus I: Sequential nondeterministic programs. In W. P. d. J. W. deBakker and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, Lecture Notes in Computer Science, pages 42–66. Springer-Verlag, 1990.

[6] R. J. R. Back and J. von Wright. Predicate transformers and higher order logic. In *REX Workshop on Semantics: Foundations and Applications*, Beekbergen, the Netherlands, 1992.

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

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

[9] R. J. R. Back and J. von Wright. Encoding, decoding and data refinement. Technical Report TUCS-TR-236, Turku Centre for Computer Science, Mar. 1, 1999.

[10] W. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice–Hall International, 1976.

[12] P. H. Gardiner and C. C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1):143–162, 1991.

[13] P. H. B. Gardiner and C. Morgan. A single complete rule for data refinement. Technical Report PRG-TR-7-89, Oxford University Computing Laboratory, Nov. 1989.

[14] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[15] A. V. Hense. Wrapper semantics of an object-oriented programming language with state. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 548–568. Springer-Verlag, Sept. 1991.

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

[17] C. A. R. Hoare, J. He, and A. C. A. Sampaio. Normal form approach to compiler design. *Acta Informatica*, 30(8):701–739, Nov. 1993.

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

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

[20] L. Mikhajlov. *Software Reuse Mechanisms and Techniques: Safety Versus Flexibility*. PhD thesis, Åbo Akademi, Åbo Akademi University, Turku, Finland, December 1999.

[21] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *ECOOP'98*, LNCS 1445, pages 355–382. Springer-Verlag, July 1998.

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

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

[24] D. A. Naumann. Predicate transformer semantics of an Oberon-like language. In *Proceedings of the IFIP WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods, and Calculi*, pages 467–487. Elsevier, June 1994.

[25] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices*, 21(11):38–45, Nov. 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.

[26] A. Tarski. A lattice theoretical fixed point theorem and its applications. *Pacific J. Mathematics*, 5:285–309, 1955.

[27] P. Wegner and S. Zdonik. Inheritance as an incremental modification mechanism. In *European Conference on Object Oriented Programming (ECOOP'88)*, pages 55–77. Springer, 1988. Lecture Notes in Computer Science, Volume 322.

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**