Inheritance of Proofs

Martin Hofmann
Arbeitsgruppe Logik und mathem. Grundl. der Informatik, Fachbereich Mathematik, Technische Hochschule Darmstadt, Schloßgartenstraße 7, D-64289 Darmstadt, Germany

Wolfgang Naraschewski
Institut für Informatik, Technische Universität München, Arcisstraße 21, D-80290 München, Germany

Martin Steffen
Institut für Informatik, Christian-Albrechts-Universität Kiel, Preusserstraße 1-9, D-24105 Kiel, Germany

Terry Stroup
Lehrstuhl für Informatik VII, Friedrich-Alexander-Universität Erlangen-Nürnberg, Martensstraße 3, D-91058 Erlangen, Germany

The Curry-Howard isomorphism, a fundamental property shared by many type theories, establishes a direct correspondence between programs and proofs. This suggests that the same structuring principles that ease programming be used to simplify proving as well. To exploit object-oriented structuring mechanisms for verification, we extend the object-model of Pierce and Turner, based on the higher-order typed λ-calculus $F^{\omega}_{\leq}$, with a logical component.

By enriching the (functional) signature of objects with a specification, the methods and their correctness proofs are packed together in the objects. The uniform treatment of methods and proofs gives rise in a natural way to object-oriented proving principles — including inheritance of proofs, late binding of proofs, and encapsulation of proofs — as analogues to object-oriented programming principles. We have used Lego, a type-theoretic proof checker, to explore the feasibility of this approach. In particular, we have verified a small hierarchy of classes.

1. Introduction

Many programming languages have been developed to ease modular and structured design of programs. The popularity of powerful structuring techniques, including object-oriented ones, is a convincing argument that those mechanisms support the programming task. Depending on the programming style, they cater to divide-and-conquer strategies for breaking down large programs into abstract data types, modules, objects, or similar. Since the resulting components ideally mirror the decomposition of the problem into conceptually self-contained units, it is natural to organize verification along the structure of the programs.

In this paper we apply this idea to object-orientation by showing that specification and verification can be organized along the class hierarchy of object-oriented programs and in particular that inheritance applies to proofs as well.

We achieve this goal by embedding Pierce and Turner's object model [47] into the Extended Calculus of Constructions (ECC) [33] — a type theory in which programs and proofs, as well as types and specifications coexist and can be interleaved. A particular feature of type theories like ECC is that logical propositions are treated as (particular) types having their proofs as elements; the so-called *Curry-Howard isomorphism*.

This means that in principle structuring mechanisms that apply to programs and types scale up to proofs and specifications. Although the Curry-Howard isomorphism serves as a guideline for generalizing the programming mechanisms to proofs, several nontrivial novel problems arise in the context of verification, e.g. the treatment of self proofs in Section 3.

Beyond the uniform treatment of programs and proofs, a further advantage of type theory is that implementations are available which equip type theory with a front-end for the interactive goal-directed development of proofs and programs. We have used the Lego implementation of ECC to carry out the proofs of all the necessary meta-properties and for the verification of a simple class hierarchy.

Our work is closely related to [23] and can be seen as a further development and elaboration of ideas expressed there. In particular, our approach builds upon the equational axiomatization of coercion and update and on the idea of "inheriting proofs" explained by way of example there. Apart from providing a formal type-theoretic underpinning for object-oriented verification, including a Lego formalization, the present paper extends [23] by providing a more general framework for the specification and verification of late-binding methods, which avoids including implementation details in specifications as was done in loc. cit.; see Section 3 for details. Moreover, we explain how β-reduction on proof terms can be used to eliminate the consistency assumptions on specifications which were needed in [23]. Finally, the treatment of generic proof methods is new.

The remainder of the paper is organized as follows: after a brief review of Lego, we present in Section 2 an encoding of object-oriented programs in Lego based on $F^{\omega}_{\leq}$, and enriched by verification. The straightforward extension, though, fails to capture all subtleties involved in the verification of late binding methods. Hence in Section 3 we modify the encoding to overcome the limitations of the first attempt. In the concluding Section 4 we discuss related and further work.

2. Verification of object-oriented programs with Lego

2.1. The Lego proof assistant

The Lego proof assistant [35] is an interactive, type-theoretic proof checker. It implements the Extended Calculus of Constructions (ECC) and a family of weaker, related type theories. The system comprises a strongly typed functional programming language as well as a higher-order intuitionistic logic. The ECC uses a predicative hierarchy of universes for programming and an impredicative universe for logical propositions. In this work, the higher universes are *not* used and instead we employ the impredicative universe $\star$ both for propositions and datatypes. Since we don't make use of the hierarchy of universes, our development could be realized in the Calculus of Constructions with inductive types as implemented in the Coq system [13], too.

We wish to stress that the use of a single universe for both propositions and types is — although pragmatically advantageous — not crucial for our approach. It would work equally well if we would employ two impredicative universes *Set* and $\star$ like in the Coq system. It seems plausible that the program extraction mechanism of Coq which strips off all terms of kind $\star$ from a type-theoretic development could then be extended to object-oriented programs. The drawback of having two separated universes is that we would have to duplicate various definitions and rules and also that the Lego implementation does not provide *Set* and $\star$.

Lego offers inductive definitions of data types together with induction principles. By means of its refinement mechanism, based on first-order unification, it supports interactive, goal-directed proof development in a natural-deduction style. Working with Lego is supported by local and global definitions, typical ambiguity, and implicit arguments, which allow the user to omit automatically synthesizable function arguments. Lego also offers arbitrary user-defined reduction rules, the soundness of which has to be verified externally, of course. We will make use of this feature in the definition of existential types with dependent elimination rule.

A detailed introduction to Lego can be found in the manual [35] or in the web resources [32].

For the purpose of understanding this paper it should be enough to know that Lego supports an extension of F^{ω} by higher-order logic with explicit proof objects. That is to say, if A is a type of F^{ω}, we can form in Lego the type $A \to \star$ of types depending on A, in particular of *predicates* over A; if $P : A \to \star$ and $a : A$ then $P(a)$ is a type; the type of proofs of $P(a)$. Elements of such proof types can be constructed interactively in a goal-directed fashion. Types and predicates can be packed together using Σ-types; for example $\Sigma x : A.P(x)$ has as elements pairs $\langle x, p\rangle$ where $x : A$ and $p : P(x)$. If $z : \Sigma x : A.P(x)$ then $z.1 : A$ and $z.2 : P(z.1)$. To ease readability, we use dependent records as syntactic sugar for (nested) Σ-types. In the above example we could write $[c_1 : A, c_2 : P(c_1)]$ for the dependent record type. The corresponding inhabitants are of the form $[c_1 = a, c_2 = p]$ where $a : A$ and $p : P(a)$, i.e. p is a proof of $P(a)$.

At the moment, the Lego type theory does not provide subtyping; we simulate a subtyping judgement $A \leq B$ by a certain type whose members are triples consisting of a coercion from A to B, an update function and a proof that these two functions satisfy certain equations. See Section 2.3.1 for details.

Conventions All definitions and proofs of this paper have been machine-checked by Lego.[1] To ease human reading, though, we do not employ Lego-syntax; instead we write terms in a more conventional notation, using λ for abstraction and $\forall$ for Lego's dependent Π-type. For the impredicative universe `Prop` of logical propositions we write $\star$ and dependent records are noted as indicated above. We fall back upon definitions provided by the Lego-library [26] whenever appropriate. If not immediately obvious, these definitions will be informally explained in the text so that no knowledge of the Lego library is required. We further use the keyword *let* for local definitions, denote unary applications $(f\,a)$ some place by $a.f$ and write $(\_,\_)$ for the non-

dependent pairing function with surjective pairing from the library `lib_prop/lib_prod.l`. We denote Leibniz equality from `lib_eq.l` by $=_L$, allowing ourselves infix notation. The inductive natural numbers of the Lego-library `lib_prop/lib_nat/lib_nat.l` are written as *nat* and we assume tacitly the usual operators to be available and standard properties to hold.

Lego supports implicit syntax to simplify definitions, synthesizing omitted arguments on its own. As in Lego, we replace a declaration $x : A$ in a $\forall$-type or in a λ-abstraction by $x \mid A$ indicating that x is an implicit argument.

In most definitions, we do not give the whole expression as a λ-term, but put some leading abstractions into the text. Free variables in types are meant to be universally quantified. Finally, we elide conjunctions between displayed equations. Apart from these conventions, though, all definitions are complete and can be directly translated into Lego.

2.2. The $F^{\omega}_{\leq}$ object model

In recent years, a number of typed λ-calculi have been investigated as foundation of typed object-oriented languages. The line of research started with Cardelli and Wegner's proposal [10] for the typed object-oriented toy language Fun based on $F_{\leq}$, an extension of the second order polymorphic λ-calculus [18, 49] by subtypes. Cardelli and Wegner proposed to model objects as records of their methods. The language Fun has spawned quite a number of different calculi of varying complexity. An overview can be found in [17], a collection of relevant papers in [20].

For our purpose of integrating an object calculus into a logical framework, one particular formal system, the system $F^{\omega}_{\leq}$ [45] is a suitable basis, since it avoids the complexity of calculi with recursive types [5, 38].

$F^{\omega}_{\leq}$, the extension of $F_{\leq}$ by type operators of arbitrary order, has been proposed by Pierce and Turner [46, 47, 24] as a core calculus for object-oriented languages in the style of Smalltalk [19]. In the following, we informally recapitulate the representation of object-oriented programming concepts in this framework. A more detailed account of representing object-oriented programs in $F^{\omega}_{\leq}$ can be found in [45, 47].

According to [47] an *object* is a collection of operations, working on an internal *state*. Both state and operations are *encapsulated* or hidden inside the object, and access is controlled by the interface. In the object model we use, encapsulation is represented by existential quantification; encapsulation by existential quantification was first proposed by [39], though for abstract data types rather than objects.

We call the type of the internal state the *representation type* of the object. The type of the operations, abstracted over the representation type, is called the object's *signature* $Sig : \star \to \star$. See the definition of *SigPoint* in the example on Page 5 for an example of such a signature. Usually, for $Rep : \star$ the type $Sig\ Rep$ is a record of functions with argument of type Rep. The type of objects with signature $Sig : \star \to \star$ is defined as

$$Object(Sig) = \exists\, Rep : \star .\ [state : Rep, ops : Sig\ Rep]$$

Using the introduction rule for existential types we can construct an element of type $Object(Sig)$ from some representation type Rep, a state $state : Rep$ and an implementation $ops : Sig\ Rep$ of the operations. Existential elimination on the other hand allows one (under a covariance condition on the form of Sig explained in detail in [24]) to define a *generic method* of type $Sig\ Object$ which when applied to an object applies its internal methods to its state and returns the packaged result.

In class-based languages, a *class* serves as a blueprint for objects and can be used in two ways: First, to create new objects sharing the representation and implementation common to the class: the classes *instances*. Second, to define new subclasses incrementally by *inheritance*, where (parts of) the definitions of the old superclass may be used. By inheritance, some methods may be re-implemented and overridden or, by enriching the signature, new methods may be added to unchanged, inherited ones.

An important intricacy are the so-called *self-methods*. This concept, popular since Smalltalk, permits methods to be defined in terms of other methods of the same class. What makes it difficult to model is that *self* does not refer statically to the methods implemented by the class. If a method refers via *self* to another method and gets inherited by a subclass, then *self* no longer refers to the operations of the superclass, from which it was inherited, but dynamically to the ones of the new class; in case one of the methods is re-implemented, all others referring to it via *self* are modified as well. This is known as dynamic binding of methods or *late binding*.

The last ingredient we mention is *subtyping*. Subtyping constitutes an order relation on types, where $S \leq T$ means that an element of type S can be regarded as an element of T and thus safely be used when an inhabitant of T is expected. This is known as *substitutability* or *subsumption*. Subtyping must not be confused with inheritance: Inheritance is the *construction* of a new subclass, whereas subtyping is concerned with the *use* of objects — or terms in general. Although inheritance and subtyping are different, there is a connection between them in this model: the type of any instance of a subclass is a subtype of the type of any instance of the superclass. Subclasses and superclasses themselves, however, are not related by subtyping.

2.3. Encoding of object-oriented programs

The system $F^{\omega}_{\leq}$ is sufficiently expressive to model object-oriented programs but, lacking dependent types, neither to specify their behavior nor to reason about them internally. We transfer the $F^{\omega}_{\leq}$ object-model to Lego and extend it in such a way that the types of the objects will not only include the functional types of the operations, but also a specification of their behavior. The objects then contain correctness proofs in addition to the implementation of the operations.

Apart from subtyping, transferring $F^{\omega}_{\leq}$'s object-oriented programming model to Lego is trivial, since in the λ-cube [4] the ω-order λ-calculus F^{ω} [18] is a subcalculus of the Extended Calculus of Constructions (more precisely of the pure Calculus of Constructions). Subtyping, though an integral part of $F^{\omega}_{\leq}$, is neither present in the Extended Calculus of Constructions nor in Lego, so we have to find an adequate representation.

2.3.1. Subtyping A type S being a subtype of T, written $S \leq T$, means that it is safe to use terms of the smaller type in all cases where a term of the bigger type is expected. This is expressed by *subsumption*. Conventionally, the subtype relation can be captured by so-called *coercion functions*, where the statement $S \leq T$ is represented as a function $f : S \to T$. If we view the type S as a more refined version of T, the coercion function extracts the T-part of elements of S. As shown in [23], this simple representation is not enough to model update together with subtype polymorphism in a functional setting. To account for updating, $S \leq T$ is represented as a *pair* of functions, say *get* and *put*, with $get : S \to T$ and $put : S \to T \to S$. The function *get* plays the role of the coercion function, extracting the T-part of elements of S, and *put* takes as first argument a value of type S and overwrites its T-part with the second argument, without altering the rest. For a restricted set of types the functions, *get* and *put* can be generated automatically. A model where subtyping is interpreted in this way has been developed for a certain restriction of $F_{\leq}$ in [23]. The interpretation of *get* and *put* as extraction and update functions is captured by the three equations of the following definition.

Definition. *[Laws for get and put [23]] Assume implicitly two types S and T and assume further two functions $get : S \to T$ and $put : S \to T \to S$. The* laws for *get* and *put are defined as the following equations:*

$$get\,(put\,s\,t) =_L t \quad (1)$$

$$put\,s\,(get\,s) =_L s \quad (2)$$

$$put\,(put\,s\,t_1)\,t_2 =_L put\,s\,t_2 \quad (3)$$

The Lego-representation *GetPutLaws* of these equations abstracted over S, T, *get*, and *put* has type $\forall S\,|\star.\ \forall T\,|\star.\ (S \to T) \to (S \to T \to S) \to \star$. We use it to define the subtype relation, where Lego's |-syntax for implicit arguments allows us to omit mentioning the first two arguments of *GetPutLaws*:

Definition. *[Subtype relation] Assume the types $S : \star$ and $T : \star$. The* subtype relation *is then defined as:*

$$S \leq T \stackrel{\text{def}}{=} [\, get : S \to T,\ put : S \to T \to S,\ gpOK : GetPutLaws\ get\ put\,]$$

The elements *gp* of a type $S \leq T$ are triples, consisting of two functions *get* and *put*, and a proof *gpOK* that they satisfy the required laws. Reflexivity and transitivity of the subtype relation are easily established.

Lemma 1. *The subtyping relation is a pre-order in the sense that there are terms $refl_{\leq} : \forall S : \star.\ S \leq S$ and $trans_{\leq} : \forall S, T, U\,|\star.\ (S \leq T) \to (T \leq U) \to (S \leq U)$.*

Proof. For reflexivity, define the two functions as the identity and the second projection. The *GetPutLaws* are immediate by reflexivity of Leibniz's equality.

For transitivity, let $gp_{S \leq T}$ be a proof for $S \leq T$ and $gp_{T \leq U}$ for $T \leq U$. Define the extraction function from S to U as the composition of $gp_{S \leq T}.get$ with $gp_{T \leq U}.get$. The update function is composed as

$$\lambda s : S.\ \lambda u : U.\ gp_{S \leq T}.put\ s\ (gp_{T \leq U}.put\ (gp_{S \leq T}.get\ s)\ u).$$

Proving the respective laws is straightforward. ∎

We do not address coherence of this notion of subtyping here, so — strictly speaking — we model $F^{\omega}_{\leq}$ *derivations* not judgements. In principle, Curien and Ghelli's coherence proof for $F_{\leq}$ with explicit coercions [14] should be applicable to the present situation.

2.3.2. Objects Intuitively, the inclusion of specifications in the interface of objects is straightforward. In addition to the functional signature $Sig : \star \to \star$, the interface needs a component *Spec* which specifies properties of the object in terms of its operations and thus has type $\forall\, Rep : \star.\ (Sig\ Rep) \to \star$. Recall that $\star$ is the kind of datatypes *and* propositions. Given a representation type *Rep*, the body of an object has type $[state : Rep, ops : Sig\ Rep, prfs : Spec\ Rep\ ops]$ and consists of a state, the operations, and a proof that they satisfy the specification.

But how to achieve encapsulation? In the informal explanation in Section 2.2, we used $F^{\omega}_{\leq}$'s existential quantifier to hide the internal state and the operations. Using the standard *impredicative encoding*

$$\exists = \lambda P : \star \to \star.\ \forall C : \star.\ (\forall R : \star.\ (P\ R) \to C) \to C$$

of the weak sum, then, as explained in [47] and [24], we can define a generic method *meths* of type *Sig Object*. However, although every object embodies a proof that its methods satisfy the specification *Spec* we cannot obtain an externalized version of the proofs, i.e. prove

Spec Object meths. The reason is that the (definable) elimination function associated with the impredicative encoding is too weak as can be seen from its type

$$\forall C : \star . (\forall R : \star . (P\ R) \to C) \to (\exists R : \star . P\ R) \to C,$$

where $P : \star \to \star$ is an arbitrary type operator. (We use the more familiar notation $\exists R : \star .\ P\ R$ instead of $\exists(\lambda R : \star .\ P\ R)$.) Roughly speaking, this elimination rule allows us to define a *function* of type $(\exists R : \star .\ P\ R) \to C$ provided we specify it on canonical elements. It does not allow us to prove anything about elements of type $\exists R : \star .\ P\ R$, i.e. to define a *dependent function* of type $\forall o : (\exists R : \star .\ P\ R) .\ C\ o$ where $C : (\exists R : \star .\ P\ R) \to \star$.

The solution we propose is to axiomatically assume an existential quantifier together with the usual introduction rule and a dependent elimination rule which overcomes the shortcoming of the impredicative encoding. Of course, the soundness of such an extension has to be validated externally as we will do below.

Definition. *[Existential quantification] The formation, the introduction, and the elimination rule for the type constructor* $\exists$ *are declared as follows:*

$$\begin{array}{lcl}
\exists & : & (\star \to \star) \to \star \\
pack & : & \forall P : \star \to \star . \forall R : \star . (P\ R) \to \exists R : \star . P\ R \\
open & : & \forall P : \star \to \star . \forall C : (\exists R : \star . P\ R) \to \star . \\
 & & \quad (\forall R : \star . \forall x : P\ R . (C\ (pack\ P\ R\ x))) \to \\
 & & \quad\quad \forall o : (\exists R : \star . P\ R) . C\ o
\end{array}$$

Assume predicates $P : \star \to \star$ *and* $C : (\exists R : \star . P\ R) \to \star$ *and a function* $f : \forall R : \star .\ \forall x : P\ R .\ C\ (pack\ P\ R\ x)$. *Assume further* $R : \star$ *and* $x : P\ R$. *The reduction rule is then defined as:*

$$open\ P\ C\ f\ (pack\ P\ R\ x) \Rightarrow f\ R\ x$$

This existential quantifier can be soundly interpreted in the PER/ω-set model of the ECC [33] as follows. If F is a function mapping partial equivalence relations on $\mathbf{N}$ (PERs) to PERs. Define $\exists(F)$ as the symmetric, transitive closure of the union of the $F(R)$ as R ranges over the set of PERs. This is the least upper bound of the $F(R)$ in the complete lattice of the PERs ordered by set-theoretic inclusion. The pack-construct can then be modelled as an inclusion map, i.e. we have $F(R) \subseteq \exists(F)$ for each R. To interpret *open* we assume a family of PERs indexed over the quotient of $\exists(F)$ or equivalently a PER $C(n)$ for each n in the domain of $\exists(F)$ and satisfying $C(n) = C(n')$ whenever n and n' are related by $\exists(F)$. The premise to *open* corresponds in the PER model to an algorithm e such that for each PER R, whenever n and n' are related in $F(R)$ then $e(n)$ and $e(n')$ are defined and related in $C(n)(= C(n'))$. Now, if n and n' are related in $\exists(R)$ it follows by induction on the length of a path relating n and n' that $e(n)$ and $e(n')$ are both defined and related in $C(n)$. So e itself yields the interpretation of *open*.

Now we can define the type of objects, using the declared existential quantifier.

Definition. *[Type of objects] Assuming a signature Sig of type* $\star \to \star$ *and a specification Spec of type* $\forall\, Rep : \star .\ (Sig\ Rep) \to \star$, *the* type of objects *is given as:*

$$\begin{array}{l}
Object \stackrel{\text{def}}{=} \exists\, Rep : \star . \\
\quad [state : Rep, ops : Sig\ Rep, prfs : Spec\ Rep\ ops]
\end{array}$$

With the existential quantifier as top-level constructor, objects are built by the existential introduction rule. To ease the presentation, we define a term for constructing objects with the help of the existential introduction operator *pack*.

Definition. *[Object introduction] Assuming implicitly a representation type Rep, a signature Sig, and a specification Spec, the function ObjectIntro for* object introduction *is defined as:*

$$\begin{array}{l}
ObjectIntro \stackrel{\text{def}}{=} \\
\quad \lambda\, mystate : Rep . \\
\quad\quad \lambda\, myops : Sig\ Rep . \\
\quad\quad\quad \lambda\, myprfs : Spec\ Rep\ myops . \\
\quad\quad\quad\quad pack\ \ (\lambda\, Rep : \star . [\ state : Rep, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad ops : Sig\ Rep, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad prfs : Spec\ Rep\ ops\ ]) \\
\quad\quad\quad\quad\quad\quad Rep \\
\quad\quad\quad\quad\quad\quad [\ state = mystate, \\
\quad\quad\quad\quad\quad\quad\ \ ops = myops, \\
\quad\quad\quad\quad\quad\quad\ \ prfs = myprfs\ ] \\
\quad : Rep \to \forall\, myops : Sig\ Rep\ .\ (Spec\ Rep\ myops) \to \\
\quad\quad Object\ Sig\ Spec
\end{array}$$

Let's illustrate these definitions of objects with the standard example of points. For the sake of discussion, our points have one coordinate in *nat* admitting examination by *getX*, overwriting by *setX*, and augmentation by *inc1*. A natural choice, though not the only possible one, for the internal representation type is the type of natural numbers itself.

Example 1. [Points] The signature SigPoint of points is the product of the types of the operations getX, setX, and inc1, abstracted over the representation type Rep:

$$\begin{array}{l}
SigPoint \stackrel{\text{def}}{=} \lambda\, Rep : \star . [\ getX : Rep \to nat, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad setX : Rep \to nat \to Rep, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad inc1 : Rep \to Rep\ ]
\end{array}$$

For the specification of points, assume a representation type Rep and operations ops conforming to the signature of type SigPoint Rep. To simplify the presentation, the specification SpecPoint consists of only two equations:

$$\begin{array}{l}
SpecPoint \stackrel{\text{def}}{=} \\
\quad ops.getX(ops.setX\ r\ n) =_L n \\
\quad\quad ops.getX(ops.inc1\ r) =_L (ops.getX\ r) + 1
\end{array}$$

The type of points Point is defined using the type constructor Object.

$$Point \stackrel{\text{def}}{=} Object\ \ SigPoint\ \ SpecPoint$$

Let us define a concrete object $MyPoint : Point$ *with representation type nat and initial value 3 by the object introduction rule ObjectIntro. The operations are implemented as:*

$$\begin{aligned} opsPoint \stackrel{\text{def}}{=} & [\, getX = \lambda n : nat\,.\, n, \\ & \quad setX = \lambda n : nat\,.\, \lambda m : nat\,.\, m, \\ & \quad inc1 = \lambda n : nat\,.\, n + 1\,] \\ & : SigPoint\ nat\,. \end{aligned}$$

The pair $prfsPoint : SpecPoint\ nat\ opsPoint$ *of correctness proofs for the two equations is immediate by reflexivity of Leibniz's equality. Putting it all together by object introduction yields a concrete point of type Point:*

$$MyPoint \stackrel{\text{def}}{=} ObjectIntro\ \ 3\ \ opsPoint\ \ prfsPoint$$

2.3.3. Generic methods So far, we have means to encapsulate the state of objects by existential quantification. As mentioned before, we also need a mechanism to gain disciplined access to the objects, using the operations and the proofs mentioned in the interface. The *generic methods* are functions that open the objects and use the internal operations to perform the requested manipulations. If the operations *ops* of an object have type $Sig\ Rep$ for some $Rep : \star$, the type of the generic functional methods *meths* is $Sig\ (Object\ Sig\ Spec)$. The generic version of proofs of $Spec\ Rep\ ops$ then has type $Spec\ (Object\ Sig\ Spec)\ meths$. In the point example, the generic methods *methsPoint* have the type $SigPoint\ Point = [\, getX : Point \to nat,\ setX : Point \to nat \to Point,\ inc1 : Point \to Point\,]$ and the generic version of the first equation of the specification is for instance $methsPoint.getX\,(methsPoint.setX\ p\ n) =_L n$. As can be seen from their types, the generic methods are to be defined generically for all objects, i.e. independently of any internal implementation.

The generic methods discussed above invoke the internal operations and proofs of objects with a specific interface. Subtyping should facilitate the use of generic methods for more refined objects, e.g. the application of the points' methods to colored points, providing additional operations and proofs dealing with the color. It is not enough, however, to be able to *apply* the generic methods to more refined objects, as the state-modifying methods have to *return* objects of the subtype, too. For example, the type of the method overwriting the x-coordinate of points should be $\forall P \leq Point\,.\, P \to nat \to P$. It is well known [45] that only trivial functions inhabit this type. The solution proposed for $F^{\omega}_{\leq}$ is to use the subtype polymorphism not on the type of objects, but on their signature, resulting in $\forall\, Sig \leq SigPoint\,.\, (Object\ Sig) \to nat \to (Object\ Sig)$ as the type for the *setX* method. In Section 2.3.1 we have encoded the subtype relation as pairs of extraction and update functions. Since for the above subtype relation on the signatures, the update part is not needed, we represent the relation simply by an extraction function.

Example 2. [Generic methods for points] Assume implicitly a signature $Sig : \star \to \star$ *and a specification* $Spec : \forall\, Rep : \star\,.\, (Sig\ Rep) \to \star$ *together with a coercion* co_sig *of type* $\forall\, Rep\,|\star\,.\, (Sig\ Rep) \to (SigPoint\ Rep)$. *The generic method* $Point'setX$ *is defined as follows:*

$$\begin{aligned} & Point'setX \stackrel{\text{def}}{=} \\ & \quad \lambda\, o : Object\ Sig\ Spec\,. \\ & \quad \lambda\, n : nat\,. \\ & \quad open\ (\lambda\, Rep : \star\,.\, [\, state : Rep, \\ & \qquad\qquad ops : Sig\ Rep, \\ & \qquad\qquad prfs : Spec\ Rep\ ops\,]) \\ & \qquad (\lambda\_ : Object\ Sig\ Spec\,.\ Object\ Sig\ Spec) \\ & \qquad (\lambda\, Rep : \star\,. \\ & \qquad\ \lambda\, stateopsprfs : [\, state : Rep, \\ & \qquad\qquad\qquad ops : Sig\ Rep, \\ & \qquad\qquad\qquad prfs : Spec\ Rep\ ops\,]. \\ & \qquad\ ObjectIntro \\ & \qquad\quad [state \ = \ (co\_sig\ stateopsprfs.ops).\, setX \\ & \qquad\qquad\qquad\qquad stateopsprfs.state\ n, \\ & \qquad\qquad ops \ = \ stateopsprfs.ops, \\ & \qquad\qquad prfs \ = \ stateopsprfs.prfs\,]\,) \\ & \qquad o \\ & : (Object\ Sig\ Spec) \to nat \to (Object\ Sig\ Spec) \end{aligned}$$

The methods $Point'getX : (Object\ Sig\ Spec) \to nat$ *and* $Point'inc1 : (Object\ Sig\ Spec) \to (Object\ Sig\ Spec)$ *can be defined analogously.*

In a similar way, the generic proof methods for points are obtained by opening the point and accessing the corresponding internal proof. In contrast to the model of Pierce and Turner we have to deal with the proof-part as well, which implies that in addition to the signature coercion $co\_sig : \forall\, Rep\,|\star\,.\, (Sig\ Rep) \to (SigPoint\ Rep)$ we require a specification coercion

$$\begin{aligned} co\_spec : & \ \forall\, Rep\,|\star\,.\, \forall\, ops\,|\, Sig\ Rep\,. \\ & \quad (Spec\ Rep\ ops) \to \\ & \qquad (SpecPoint\ Rep\ (co\_sig\ ops))\ . \end{aligned}$$

With these extraction functions we can define generic proof methods for the equations of the specification of points.

Example 3. [Generic proof methods for points] Assume implicitly a signature Sig, a specification Spec, and coercion functions co_sig and co_spec. The generic proof method for the first equation $Point'prf_1$ *is defined*

as follows:

$$
\begin{array}{l}
Point'prf_1 \stackrel{\text{def}}{=} \\
\quad \lambda\, o : Object\ Sig\ Spec\ . \\
\quad \lambda\, n : nat\ . \\
\quad open\ (\lambda\, Rep :\star\ .\ [\ state : Rep, \\
\qquad\qquad\qquad\qquad ops : Sig\ Rep, \\
\qquad\qquad\qquad\qquad prfs : Spec\ Rep\ ops\ ]) \\
\qquad\quad (\lambda o : Object\ Sig\ Spec\ .\ \ Point'getX\ co\_sig \\
\qquad\qquad\quad (Point'setX\ co\_sig\ \ o\ n) =_L\ n) \\
\qquad\quad (\lambda\, Rep :\star\ . \\
\qquad\quad \lambda\, stateopsprfs : [\ state : Rep, \\
\qquad\qquad\qquad\qquad\qquad ops : Sig\ Rep, \\
\qquad\qquad\qquad\qquad\qquad prfs : Spec\ Rep\ ops\ ]\ . \\
\qquad\quad let\ \ mystate\ \ =\ \ stateopsprfs.state \\
\qquad\qquad\ \ myprfs\ \ =\ \ co\_spec\ (stateopsprfs.prfs) \\
\qquad\quad in\ myprfs.1\ \ mystate\ \ n) \\
\qquad\quad o \\
\quad : \forall o : Object\ Sig\ Spec\ .\ \forall n : nat\ . \\
\qquad\quad Point'getX\ co\_sig\ (Point'setX\ co\_sig\ o\ n) =_L\ n
\end{array}
$$

In the same way, the generic proof method $Point'prf_2$ *of the second equation has type* $\forall o : Object\ Sig\ Spec\ .\ \forall n : nat\ .\ Point'getX\ \ co\_sig\ \ (Point'inc1\ \ co\_sig\ \ r) =_L (Point'getX\ co\_sig\ r)+1$ *and can be defined analogously.*

We have illustrated the generic methods on the specific example of points. For a restricted set of signatures it is possible to define the generic methods uniformly [24], namely for signatures of the form $\lambda\, Rep :\star\ .\ Rep \to (T\ \ Rep)$, where T is *covariant* in its argument Rep.

The restriction to covariant signatures excludes the definition of *binary* generic methods such as $Point \to Point \to bool$ since they would need to compare the state of two points of arbitrary representation types; but these are hidden by the existential quantifier. This phenomenon has been discussed already in the context of abstract data types in [39] and [36]. (Cf. [6] for a detailed discussion of problems related with binary methods in typed object-oriented programming languages.)

In this example we were able to define the generic functional methods, as the signature is basically of the above form. Instead of *SigPoint*, we could have used $\lambda\, Rep :\star\ .\ \ Rep \to [getX : nat, setX : nat \to Rep, inc1 : Rep\ ]$ as well; for presentational purposes, we have chosen the form of signature on Page 5.

Similarly, a generic proof method with type $Spec\ (Object\ Sig\ Spec)\ meths$ can only be defined if the specification has the form of universally quantified clauses each of which contains the type Rep in covariant position only. For example, if *SpecPoint* would contain a third clause

$$
\begin{array}{l}
\forall r : Rep\ . \\
\quad \forall r' : Rep\ .\ (ops.getX\ \ r =_L\ ops.getX\ \ r') \to (r =_L\ r')
\end{array}
$$

then a generic proof method would state that two point objects with equal x-coordinate are equal. Note that the type of r' constitutes a contravariant occurrence of type Rep. Even if this property is locally satisfied by concrete representations it is unsound in its general form (think of points with a color attribute which is blue in one representation and red in another one). We believe that the required covariance condition on *Spec* can be formalised in such a way that a uniform definition of generic proof methods can be given along the lines of [24].

For the moment we circumvent this problem by giving an explicit definition of generic (proof) methods in each case.

2.3.4. Objects without logical components In the previous sections we have emphasized the benefit of packing programs and proofs together in the objects. In the context of formal verification the given arguments are justified, but they don't apply if the objects are to be executed. For this purpose the proofs are ballast; worse still they are big. As programs and proofs form a pair, we can jettison the proofs simply by projecting out the programs. To take care of encapsulation, we open the objects first, then extract the programs, and finally repack the objects without the proofs. The type of the resulting trim objects coincides with the one in [47].

Definition. *[Type of objects without logical component] Assuming a signature* $Sig :\star \to \star$ *the* type of objects without proof component *is given as:*

$$Object\_eff \stackrel{\text{def}}{=} \exists\, Rep :\star\ .\ [state : Rep, ops : Sig\ Rep]$$

Defining the function $forget\_prfs$ of type $\forall\, Sig :\star \to \star\ .\ \forall\, Spec : (\forall\, Rep :\star\ .\ (Sig\ Rep) \to \star)\ .\ (Object\ Sig\ Spec) \to (Object\_eff\ Sig)$ which forgets the proof-part of objects, is analogous to defining generic methods.

Definition. *[Objects without logical component] Assuming implicitly a representation type Rep, a signature Sig, and a specification Spec, the term for* forgetting the proof component *is defined as:*

$$
\begin{array}{l}
forget\_prfs \stackrel{\text{def}}{=} \\
\quad \lambda\, o : Object\ Sig\ Spec\ . \\
\quad open\ (\lambda\, Rep :\star\ .\ [\ state : Rep, \\
\qquad\qquad\qquad\qquad ops : Sig\ Rep, \\
\qquad\qquad\qquad\qquad prfs : Spec\ Rep\ ops\ ]) \\
\qquad\quad (\lambda\_ : Object\ Sig\ Spec\ .\ \ Object\_eff\ Sig) \\
\qquad\quad (\lambda\, Rep :\star\ . \\
\qquad\quad \lambda\, stateopsprfs : [\ state : Rep, \\
\qquad\qquad\qquad\qquad\qquad ops : Sig\ Rep, \\
\qquad\qquad\qquad\qquad\qquad prfs : Spec\ Rep\ ops\ ]\ . \\
\qquad\quad pack\ \ (\lambda\, Rep :\star\ .\ [state : Rep, ops : Sig\ Rep]) \\
\qquad\qquad\ \ Rep \\
\qquad\qquad\ \ [\ state\ \ =\ \ stateopsprfs.state \\
\qquad\qquad\ \ \ ops\ \ =\ \ stateopsprfs.ops\ ]) \\
\qquad\quad o \\
\quad : (Object\ Sig\ Spec) \to (Object\_eff\ Sig)
\end{array}
$$

Alternatively, we can view "inheritance of proofs" merely as a methodology for structuring informal proofs on paper.

2.3.5. Classes As mentioned in Section 2.2, a class determines the implementation of its instances. Since we have extended the interface of objects with a specification, a class has to provide not only the code of the operations, but a proof of its correctness as well.

We cannot yet implement the class for a fixed representation type, say *ClassR*, since the mechanism of *inheritance* may extend and change the representation type. So the signature and the specification both have to refer to a representation type *Rep*, as yet indeterminate. Of course we cannot expect to program nontrivial operations and proofs for an arbitrary representation type *Rep*. Constraining the possible representation types to subtypes of the fixed *ClassR* gives the necessary connection between the two types in terms of the extraction and update function: the laws of *get* and *put* on Page 4 guarantee that the operations will behave correctly on the *ClassR* part of its subtype *Rep* without compromising the rest. The representation type *Rep* remains provisional as long as we create subclasses by inheritance. It will be fixed, i.e. identified with the representation type of the corresponding class, only when an instance of the class is generated. Hence we could write the type of a class with fixed representation type *ClassR*, signature *Sig*, and specification *Spec* as $\forall\, Rep : \star.\ (Rep \leq ClassR) \to [ops : Sig\ Rep, prfs : Spec\ Rep\ ops]$.

So far, though, we have not said a word about self-methods and self-proofs. The possibility of self-reference to operations and proofs in classes is the key to the flexibility of inheritance. In this functional setting, self-reference is simply achieved by assuming *self* as a variable of type $[ops : Sig\ Rep, prfs : Spec\ Rep\ ops]$, i.e. the implementation is abstracted over this variable, giving classes the following type.

Definition. *[Type of classes] Assume a representation type* $ClassR : \star$*, a signature* $Sig : \star \to \star$*, and a specification* $Spec : \forall\, Rep : \star.\ (Sig\ Rep) \to \star$*. The* type of classes *is given as:*

$$
\begin{array}{l}
Class \stackrel{\text{def}}{=} \\
\quad \forall\, Rep : \star.\ (Rep \leq ClassR) \to \\
\qquad [ops : Sig\ Rep, prfs : Spec\ Rep\ ops] \to \qquad (self) \\
\qquad\quad [ops : Sig\ Rep, prfs : Spec\ Rep\ ops]
\end{array}
$$

A fixed point operator will be used to resolve the functional abstraction on *self* at instantiation time; this will be discussed in the following section. Again we illustrate the definition by our running example.

Example 4. [Class of points] The type PointClass of classes of points with representation type nat, signature SigPoint, and specification SpecPoint (cf. the example on Page 5) is built using the type constructor Class:

$$PointClass \stackrel{\text{def}}{=} Class\ nat\ SigPoint\ SpecPoint$$

We define a concrete class MyPointClass of type PointClass as pair of the operations and of their correctness proofs. The abstraction over self enables reference to the self-methods and self-proofs.

$$
\begin{array}{l}
MyPointClass \stackrel{\text{def}}{=} \\
\quad \lambda\, Rep : \star.\ \lambda\, gp : Rep \leq nat\ . \\
\quad \lambda\, self : [ops : SigPoint\ Rep, prfs : SpecPoint\ Rep\ ops]. \\
\qquad [ops = opsPointClass, prfs = prfsPointClass]
\end{array}
$$

The operations of the class are implemented as the following triple:

$$
\begin{array}{l}
opsPointClass = \\
\quad [\, getX \ = \ \lambda r : Rep\ .\ gp.get\ r, \\
\qquad setX \ = \ \lambda r : Rep\ .\ \lambda n : nat\ .\ gp.put\ r\ n, \\
\qquad inc1 \ = \ \lambda r : Rep\ .\ self.ops.setX\ r \\
\qquad\qquad\qquad\qquad (self.ops.getX\ r) + 1\,]
\end{array}
$$

Although the example may suggest that the two functions get and put for the subtype relation were tailored to encode the two methods getX and setX the contrary is true. As shown in [23] the functions get and put are canonical in the sense that they make the basic manipulations of a state available: reading and updating it.

Finally, we have to prove the correctness of the operations just defined, i.e. give an element prfsPointClass of type SpecPoint Rep opsPointClass. The first equation of the specification

$$opsPointClass.getX\ (opsPointClass.setX\ r\ n) =_L n$$

only contains operations not depending on self. Using their implementation it reduces to:

$$gp.get\ (gp.put\ r\ n) =_L n$$

The equation coincides with the first law for get and put, accessible by gp.gpOK.1.

The specification's second equation postulates the correct behaviour of the increment operation, which is defined in terms of self:

$$
\begin{array}{l}
opsPointClass.getX\ (opsPointClass.inc1\ r) =_L \\
\quad (opsPointClass.getX\ r) + 1
\end{array}
$$

which β*-reduces to*

$$
\begin{array}{l}
gp.get\ (self.ops.setX\ r\ (self.ops.getX\ r) + 1) =_L \\
\quad (gp.get\ r) + 1
\end{array}
$$

This equation, though, is not provable in the present situation. The reason is that there is no way to relate the implementation of the methods — in the above equation the function get as implementation of the getX method — with the operations referred to by self . In the current encoding of classes, a richer specification

would not help, since the necessary connection cannot even be specified. This does not imply that proofs about self methods are impossible at this stage. It is possible to prove equations involving only self methods, but not, as in the above equation, those involving both self and other methods. Section 3 will discuss this problem and propose solutions.

2.3.6. Instantiation The instantiation operator *new* is a function that generates a new object when applied to a class and an initial value. As explained in the previous section, a class does not provide an implementation of objects for a fixed representation type $ClassR$, but for any representation type $Rep \leq ClassR$. At instantiation, the representation type becomes fixed, i.e. identified with $ClassR$. In addition, classes are abstracted over the variable *self*. This dependency has to be resolved, ensuring that *self* now refers to the class being instantiated.

In [46], this dependency was resolved using a fixed point operator. Although it is in principle possible to extend Lego by general recursion, see e.g. [48, 2], we have opted for the simpler alternative of using the bounded fixpoints from [23]. This means that we replace an instance $fix\,f$ where $f : A \to A$ by the n-fold iteration $f^n(basis)$ of f on a start value $basis : A$. This expression yields the unique fixpoint of f provided f^n is a constant function. Again following [23] we think that it is a reasonable restriction that dependencies of methods on *self* be resolved after a fixed, input-independent number of recursive unfoldings. For a more thorough discussion of bounded fixpoints we refer to op. cit. In Lego the n-fold iteration of a function is encoded using the iteration operator *nat_iter* of type $\forall A \mid Type\,.\; A \to (A \to A) \to nat \to A$ as defined in the Lego-library.

Definition. *[Instantiation] Assuming implicitly a representation type ClassR, a signature Sig, and a specification Spec, the instantiation operator new is defined as:*

$$
\begin{array}{l}
new \stackrel{\text{def}}{=} \\
\quad \lambda\, class : Class\ ClassR\ Sig\ Spec\,. \\
\quad \lambda\, state : ClassR\,. \\
\quad \lambda\, basis : [ops : Sig\ ClassR, prfs : Spec\ ClassR\ ops] \\
\quad \lambda n : nat\,. \\
\qquad let\ \ opsprfs = \\
\qquad\qquad nat\_iter\ \ basis \\
\qquad\qquad\qquad (class\ \ ClassR\ (refl_{\leq}\ \ ClassR)) \\
\qquad\qquad\qquad n \\
\qquad in\ \ (ObjectIntro\ state\ opsprfs.ops\ \ opsprfs.prfs) \\
\quad : (Class\ ClassR\ Sig\ Spec) \to ClassR \to \\
\quad\ [ops : Sig\ ClassR, prfs : Spec\ ClassR\ ops] \to nat \to \\
\qquad Object\ Sig\ Spec
\end{array}
$$

This instantiation operator neither guarantees that after the given number of function iterations the self-methods and self-proofs are resolved, nor that they are resolvable at all. To ensure this, the definition can easily be modified so that the programmer has to prove that $nat\_iter\ \ basis\ \ (class\ \ ClassR\ \ (refl_{\leq}\ \ ClassR))\ \ n$ is indeed a fixed point of $class\ \ ClassR\ \ (refl_{\leq}\ \ ClassR)$. In Section 3 we give a more refined definition of instantiation which requires the programmer to provide a proof that the iteration is indeed a fixed point of the class. Also notice that instantiation takes a correct implementation, namely *basis* as an argument. In other words, we have to provide an implementation plus a correctness proof in the first place.

In a strongly normalising system like Lego the *rule of strengthening* is admissible. This means that if a term — be it a program or a proof — does not literally contain a variable (or an assumption) then it can by type-checked without this variable (or assumption). This allows us to use the following semi-algorithm to compute an appropriate iteration index n *and* to discharge the consistency assumption *basis*.

- Given $class\ \ ClassR\ \ (refl_{\leq}\ \ ClassR)$ successively compute the β-normal forms of $class^i\ basis$ where $basis : [ops : Sig\ ClassR, prfs : Spec\ ClassR\ ops]$ is a fresh variable.
- Stop when an index n is found such that the β normal form of $class^n\ basis$ does not contain *basis* literally.

Now, $selfresolved \stackrel{\text{def}}{=} class^n\ basis$ is a fixed point of *class* and moreover, by strengthening, constitutes a provably correct implementation of the specification *Spec* not depending on consistency of the latter. This semi-algorithm will always terminate if the function *class* does not contain circular dependencies on self either in the code or in the proofs. We believe that this situation can in many cases also be recognized by a static analysis of the structure of possible calls to self.

Example 5. [Instance of points] A concrete object MyPointInstance with x-coordinate 3 is instantiated from the class PointClass by means of the instantiation operator new. Only two iterations are needed to resolve the self-methods; thereafter the variable self has disappeared. We can therefore apply the instantiation operator to a variable basis and nevertheless obtain (by strengthening) a closed and correct implementation:

$$
\begin{array}{l}
basis : [ops : SigPoint\ nat, prfs : SpecPoint\ nat\ ops] \\
MyPointInstance \stackrel{\text{def}}{=} new\ \ MyPointClass\ \ 3\ \ basis\ \ 2
\end{array}
$$

We want to stress that this line of reasoning hinges on the fact that we have explicit proof objects and β-reduction on these. Without β-reduction the variable *basis* could never literally disappear.

2.3.7. Inheritance Inheritance allows to define new classes by means of already defined ones. As in the object model of $F^{\omega}_{\leq}$, inheritance is represented by a function *inherit* which generates the subclass when applied

to a superclass and to a function *build*. The argument *build* serves as an instruction how to construct the subclass from the implementation of the superclass.

Like any class, the subclass has to be implemented for an arbitrary subtype *Rep* of its representation type *SubR*. To use the implementation of the superclass in the subclass, we have to ensure that the operations of the superclass work on *Rep* as well. A proof of $SubR \leq SuperR$ together with transitivity of the subtype relation suffices.

Late binding requires that the variable *self* in the inherited operations and proofs be bound to the *self*-parameter of the present class. This is achieved by transforming the *self*-parameter using appropriate coercion functions between the signatures and the specifications.

Definition. *[Inheritance] Assume implicitly a representation type* $SuperR : \star$*, a signature* $SuperSig : \star \to \star$*, and a specification SuperSpec of the superclass, which itself has type* $\forall\, Rep : \star .\ (SuperSig\ Rep) \to \star$*. In addition, assume for the subclass a representation type SubR, a signature SubSig, and a specification SubSpec correspondingly. Finally, assume a proof* $gp_{SubR \leq SuperR}$ *of type* $SubR \leq SuperR$ *and two coercion functions* $co\_sig : \forall\, Rep \,|\, \star .\ (SubSig\ Rep) \to (SuperSig\ Rep)$ *and* co_spec *of type* $\forall\, Rep \,|\, \star .\ \forall\, ops \,|\, SubSig\ Rep .\ (SubSpec\ Rep\ ops) \to (SuperSpec\ Rep\ (co\_sig\ ops))$*. The* inheritance operator *is then defined as follows:*

$$
\begin{array}{l}
inherit \stackrel{\text{def}}{=} \\
\quad \lambda\, SuperClass : Class\ SuperR\ SuperSig\ SuperSpec . \\
\quad \lambda\, build : \forall\, Rep : \star .\ (Rep \leq SubR) \to \\
\qquad\qquad [ops : SuperSig\ Rep, \\
\qquad\qquad\ prfs : SuperSpec\ Rep\ ops] \to \qquad (super) \\
\qquad\qquad [ops : SubSig\ Rep, \\
\qquad\qquad\ prfs : SubSpec\ Rep\ ops] \to \qquad (self) \\
\qquad\qquad [ops : SubSig\ Rep, prfs : SubSpec\ Rep\ ops]. \\
\quad (\ \lambda\, Rep : \star .\ \lambda\, gp_{Rep \leq SubR} : Rep \leq SubR . \\
\qquad \lambda\, self : [ops : SubSig\ Rep, prfs : SubSpec\ Rep\ ops]. \\
\qquad build\ \ Rep\ gp_{Rep \leq SubR} \\
\qquad\qquad (SuperClass\ \ Rep \\
\qquad\qquad\qquad\qquad (trans_{\leq}\ \ gp_{Rep \leq SubR} \\
\qquad\qquad\qquad\qquad\qquad\ gp_{SubR \leq SuperR}) \\
\qquad\qquad\qquad\qquad [\ ops\ \ =\ \ co\_sig\ \ self.ops, \\
\qquad\qquad\qquad\qquad\ \ prfs\ \ =\ \ co\_spec\ \ self.prfs\ ]) \\
\qquad\qquad self\ ) \\
: (Class\ SuperR\ SuperSig\ SuperSpec) \to \\
\quad (\forall\, Rep : \star .\ (Rep \leq SubR) \to \\
\qquad [ops : SuperSig\ Rep, prfs : SuperSpec\ Rep\ ops] \to \\
\qquad [ops : SubSig\ Rep, prfs : SubSpec\ Rep\ ops] \to \\
\qquad [ops : SubSig\ Rep, prfs : SubSpec\ Rep\ ops]) \to \\
\quad (Class\ SubR\ SubSig\ SubSpec)
\end{array}
$$

Continuing the example, we use inheritance to construct a class of colored points. Thus assume a type $Color : \star$ together with elements *blue*, *red*, ... In addition to the operations *getX*, *setX* and *inc1* of points, the class of colored points contains the operations *inc2* and *getC*, where the operation *inc2* increments the coordinate by two and *getC* extracts the color.

Example 6. [Colored points] The signature of colored points SigCPoint extends the signature of points SigPoint by the types of the operations inc2 and getC, abstracted over a representation type Rep:

$$
\begin{array}{l}
SigCPoint \stackrel{\text{def}}{=} \lambda\, Rep : \star .\ [\ opspoint : SigPoint\ \ Rep, \\
\qquad\qquad\qquad\qquad\qquad\ inc2 : Rep \to Rep, \\
\qquad\qquad\qquad\qquad\qquad\ getC : Rep \to Color\ ]
\end{array}
$$

To simplify the further exposition, we pretend that the operations form a flat quintuple. We also use names such as getX, setX etc. for the appropriate record selectors, when the meaning is clear from the context.

For the specification SpecCPoint, assume an arbitrary representation type Rep and operations ops of type SigCPoint Rep. The specification SpecCPoint extends the specification SpecPoint by three equations.

$$
\begin{array}{l}
SpecCPoint \stackrel{\text{def}}{=} \\
\quad (SpecPoint\ \ Rep\ \ ops.opspoint) \times \\
\quad (\ ops.getX\ \ (ops.inc2\ \ r) \quad =_L\ \ (ops.getX\ \ r) + 2 \\
\qquad ops.getC\ \ (ops.inc2\ \ r) \quad =_L\ \ blue \\
\qquad ops.getC\ \ (ops.setX\ \ r\ n) \ =_L\ \ blue\ )
\end{array}
$$

As in the case of the operations, we assume that the proofs form a flat quintuple. We refer to the components of prfs which has type SpecCPoint Rep ops by prfs.1 through prfs.5.

Now, we define a class MyCPointClass with representation type $(nat \times Color)$ *by means of the inheritance operator inherit.*

$$
\begin{array}{l}
MyCPointClass \stackrel{\text{def}}{=} \\
\quad inherit\ (nat \times Color)\ SigCPoint\ SpecCPoint \\
\qquad\quad gp\ co\_sig\ co\_spec \\
\qquad\quad MyPointClass \\
\qquad\quad (\ \lambda\, Rep : \star . \\
\qquad\qquad \lambda\, gp_{Rep \leq (nat \times Color)} : Rep \leq (nat \times Color) . \\
\qquad\qquad \lambda\, super : [\ ops : SigPoint\ Rep, \\
\qquad\qquad\qquad\qquad\ prfs : SpecPoint\ Rep\ ops\ ]. \\
\qquad\qquad \lambda\, self : [\ ops : SigCPoint\ Rep, \\
\qquad\qquad\qquad\qquad\ prfs : SpecCPoint\ Rep\ ops\ ]. \\
\qquad\qquad [\ ops = opsCPointClass, \\
\qquad\qquad\ \ prfs = prfsCPointClass\ ] \\
\qquad\quad) : Class\ (nat \times Color)\ SigCPoint\ SpecCPoint
\end{array}
$$

The term $gp : (nat \times Color) \leq nat$ *is a dependent triple consisting of functions* $get = \lambda nc : (nat \times Color) .\ nc.1$ *and* $put = \lambda nc : (nat \times Color) .\ \lambda n : nat .\ (n, nc.2)$ *together with the straightforward verification of the required laws. The proof uses the* η*-rule for pairs wich is provided by their inductive definition from the Lego-library. The coercion* $co\_sig : \forall\, Rep \,|\, \star .\ SigCPoint\ Rep \to SigPoint\ Rep$ *for the signature and co_spec of type*

$\forall\, Rep \mid \star .\, \forall\, ops \mid (SigCPoint\ Rep).\ (SpecCPoint\ Rep\ ops) \rightarrow (SpecPoint\ Rep\ \ (co\_sig\ ops))$ *for the specification part respectively simply forget the new operations and the new proofs.*

To implement the operations opsCPointClass, we inherit getX and inc1 from the superclass of points. To illustrate late binding, the operation setX of the colored point class artificially sets the color to blue. The operation inc2 uses the operation inc1 of the point-class twice and getC simply extracts the color. In the definition of the operations, the variables self and super provide access to the methods of the colored point class and the point class respectively.

$$
\begin{array}{l}
opsCPointClass = \\
\quad [\ getX \ = \ \lambda r : Rep\ .\ super.ops.getX\ \ r, \\
\quad\ setX \ = \ \lambda r : Rep\ .\ \lambda n : nat\ . \\
\qquad\qquad\qquad gp_{Rep \leq (nat \times Color)}.put\ \ r\ (n, blue), \\
\quad\ inc1 \ = \ \lambda r : Rep\ .\ super.ops.inc1\ \ r, \\
\quad\ inc2 \ = \ \lambda r : Rep\ .\ super.ops.inc1 \\
\qquad\qquad\qquad\qquad\qquad (super.ops.inc1\ \ r), \\
\quad\ getC \ = \ \lambda r : Rep\ .\ (gp_{Rep \leq (nat \times Color)}.get\ \ r).2\ )\ ]
\end{array}
$$

Finally, we have to prove the correctness of these five operations, i.e. give an element prfsCPointClass of type SpecCPoint Rep opsCPointClass.

We have to postpone the discussion of the first and the fourth equation since at this stage it is not possible to prove propositions relating the variable super with get and put. The problem is similar to the one for self encountered in the encoding of classes (cf. the example on Page 8) and will be addressed in the next section.

The second equation of the quintuple SpecCPoint on Page 10 reduces to $super.ops.getX\ \ (super.ops.inc1\ \ r) =_L (super.ops.getX\ \ r)+1$, *which coincides with the type of super.prfs.2. Therefore, we can use the inherited proof super.prfs.2 to show the correctness of the current equation. This equation demonstrates that it is possible to* inherit correctness proofs *to verify inherited operations. Note that the situation of the previous equation is not as simple as the proof might suggest. The operation inc1 in the subclass refers, as in the superclass, via self to the setX operation, which we have changed in the subclass. Due to late binding, this also affects the implementation of inc1. Nevertheless, the inheritance of the proof works, since we have not altered the behaviour of setX on the point part.*

This way of reasoning is not restricted to situations where the inherited proof is reused without modification. New equations of a subclass can also be proven by proof inheritance, as can be seen in the third equation.

Expanding the definitions of the operations getX and inc2 in opsCPointClass, the third equation becomes $super.ops.getX\ (super.ops.inc1\ (super.ops.inc1\ r)) =_L (super.ops.getX\ \ r) + 2$ *and can be shown by employing the inherited proof super.prfs.2 twice.*

The last equation can be established easily with the laws for get and put, even though the point part of setX is inherited and in the equation super is mixed with get and put. This is feasible because the definition of the point part is irrelevant for the proof.

3. Proofs over self-methods

In this section we improve the encoding of classes, instantiation, and inheritance, to overcome the difficulties with proofs over methods with late binding. The definitions of objects, generic methods, and generic proof methods remain unchanged.

3.1. Classes

As seen in the previous section, we can cope with equations about non late binding methods. We have also mentioned in the example on Page 8 that some equations with self methods are provable, namely if the specification of the self methods suffices to establish the properties to be shown. In many cases, especially when self methods appear together with non-self methods, we are stuck. The reason is that, by late binding, the self methods may refer to operations of subclasses whereas the non-self methods refer to the special implementation of the present class. For instance, in the example on Page 8 we cannot expect to prove the second equation in the specification of points, relating the implementations of *getX* and *inc1*:

$$
\begin{array}{c}
gp.get\ (self.ops.setX\ \ r\ (self.ops.getX\ \ r) + 1) \\
=_L (gp.get\ \ r) + 1
\end{array}
\qquad (1)
$$

since we do not know how the *setX* method in subclasses will behave together with the current implementation *gp.get* of the *getX* method. The only thing we know about the self-operations is that they satisfy the specification; the verification cannot rely on any details of the implementation.

In [23] the problem was overcome by including implementation details into the specification.[2] In our example, one would then add the equation $ops.getX =_L gp.get$ to the specification of points. This would give the desired connection between *self* and the present implementation: $self.ops.getX =_L gp.get$. However, such a specification of internal details is problematic since it *fixes the implementation* also for the subclasses, which will have to satisfy the extended specification, too. Even worse: including implementation details into the objects' interfaces misses the point of encapsulation, whose purpose is to abstract away from details.

The previous analysis shows that without restriction on the implementation of the subclasses Equation (1) is simply not true in the class *PointClass*. Nevertheless, after solving the self-operations of the point class by a

fixed point, the equation does become provable. The operations *self.ops.getX* and *self.ops.setX* then get replaced by their implementation, yielding:

$$\underbrace{gp.get}_{getX}\ (\underbrace{gp.put}_{setX}\ r\ (\underbrace{gp.get}_{getX}\ r)\ +1) =_L (\underbrace{gp.get}_{getX}\ r)\ +1$$

This equation follows from the first equation in the specification of points. This observation applies not only to the class of points itself, but to all of its subclasses: upon instantiation, the *self* gets replaced by the implementation provided by the respective subclass. The second equation then takes the form:

$$impl.getX\,(impl.setX\ \ r\ (impl.getX\ r) + 1) =_L \quad (1_{sc})$$
$$(impl.getX\ r) + 1$$

where *impl.getX* and *impl.setX* are the concrete implementation of the methods *getX* and *setX*, thus satisfying at least the specification of points. Again we can use the first equation of *SpecPoint* for the proof.

These calculations suggest that the problem with the verification of late binding methods (here *inc1*) is not a genuine one, but rather caused by a limitation of our approach. Simply requiring that a class maps correct implementations to correct implementations (as was done in Definition 2.3.5 on Page 8) is not enough to establish all reasonable and expected properties.

To overcome this problem, we will consider generalized specifications which contain two copies of every method: one ranging over the actual implementation and the other one over the self parameter. More formally, we consider $Spec' : \forall\, Rep\,.\ (Sig\ Rep) \to (Sig\ Rep) \to \star$. It is the task of the programmer or verifier to decide which copy of a method is appropriate for the generalized specification. We will show below that the fixed point of the program part of a class satisfies the intended diagonalized specification $Spec \stackrel{\text{def}}{=} \lambda\, ops : (Sig\ Rep).\ Spec'\ ops\ ops$. In the sequel we will use primes to refer to a generalized specification and use their name without prime for the diagonalized specification.

In the specification of points, the separation of abstract and actual implemented operations is done as follows:

Example 7. [Generalized specification of points] Assuming a representation type Rep, concrete operations ops, and abstract operations selfops of type SigPoint Rep, the generalized specification *of points is given as*

$$\begin{array}{l} SpecPoint' \stackrel{\text{def}}{=} \\ \quad ops.getX\ (ops.setX\ r\ n)\ =_L\ n \\ \quad selfops.getX\ (ops.inc1\ r)\ =_L\ (selfops.getX\ r) + 1 \end{array}$$

Notice that $SpecPoint =_L SpecPoint'\ ops\ ops$.

A class will now provide for all $Rep \le ClassR$

- a self-dependent implementation $code : (Sig\ Rep) \to (Sig\ Rep)$ of the operations, and
- a proof that $Spec'\ selfops\ selfops'$ entails the correctness of $Spec'(code\ selfops)\ selfops$.

More formally:

Definition. *[Type of classes] Assume a representation type* $ClassR : \star$*, a signature* $Sig : \star \to \star$*, and a generalized specification* $Spec' : \forall\, Rep : \star.\ (Sig\ Rep) \to (Sig\ Rep) \to \star$*. Let the term Spec stand for the specification* $\lambda\, Rep : \star.\ \lambda\, ops : (Sig\ Rep).\ Spec'\ Rep\ ops\ ops$*, which represents the ungeneralized version of Spec′. The* type of classes *is then given as:*

$$\begin{array}{l} Class \stackrel{\text{def}}{=} \forall\, Rep : \star.\ (Rep \le ClassR) \to \\ \quad [\ code : (Sig\ Rep) \to Sig\ Rep\ . \\ \quad\ prfs : \forall\, selfops, selfops' : Sig\ Rep\ . \\ \quad\quad\quad (Spec'\ Rep\ selfops\ selfops') \to \\ \quad\quad\quad\ Spec'\ Rep\ (code\ selfops)\ \ selfops\ ] \end{array}$$

Notice that we have now separated the program part and the specification part of a class into two components. We could have done so in the case of the simple classes as well, but found the mixing of the two using Σ-types more perspicuous.

The rest of the section is concerned with adapting instantiation and inheritance. Before starting with instantiation, we complete the class of points with the new definition.

Example 8. [Class of points] The type of points and their signature remain unchanged. As shown, the original specification SpecPoint is slightly generalized to SpecPoint′.

The type of point classes with representation type nat, signature SigPoint, and generalized specification SpecPoint′, is constructed by means of the type constructor Class:

$$PointClass \stackrel{\text{def}}{=} Class\ nat\ SigPoint\ SpecPoint'$$

The concrete class MyPointClass is again a pair of operations and correctness proofs:

$$\begin{array}{l} MyPointClass \stackrel{\text{def}}{=} \\ \quad \lambda\, Rep : \star.\ \lambda\, gp : Rep \le nat\ . \\ \quad\quad [code = codePointClass, prfs = prfsPointClass] \\ \quad : PointClass\ . \end{array}$$

As programs and proofs are now separated, the self reference to the operations and the proofs is no longer achieved by the single variable self, but by two distinct variables: selfops and selfprfs. The implementation of the operations can be used without change:

$$\begin{array}{l} codePointClass = \\ \quad \lambda\, selfops : SigPoint\ Rep\ . \\ \quad\quad [\ getX\ \ = \lambda r : Rep\ .\ gp.get\ \ r, \\ \quad\quad\ \ setX\ \ = \lambda r : Rep\ .\ \lambda n : nat\ .\ gp.put\ \ r\ n, \\ \quad\quad\ \ inc1\ \ = \lambda r : Rep\ .\ selfops.setX\ \ r \\ \quad\quad\quad\quad\quad\quad (selfops.getX\ \ r) + 1\ ] \end{array}$$

Finally, we have to prove their correctness, i.e. assuming selfops and selfops′ of type SigPoint Rep and selfprfs of type SpecPoint Rep selfops selfops′, provide a correctness proof prfsPointClass of the specification SpecPoint′ Rep (codePointClass selfops) selfops. The proof of its first equation is identical to the one in the old definition of this class. The second equation, the one we had to modify, now β-reduces to

$$selfops.getX\,(selfops.setX\ \ r\ (selfops.getX\ r) + 1) =_L (selfops.getX\ r) + 1$$

The self-proof selfprfs.1 $: \forall r : Rep\ .\ \forall n : nat\ .\ selfops.getX\ (selfops.setX\ r\ n) =_L n$ *establishes its correctness.*

3.2. Instantiation

Next we adapt the instantiation function to deal with the modified definition of classes. Compared with the definition of *new* in Section 2 the major difference arises from the way the self-dependencies are resolved. In Section 2 the self-dependent operations and proofs of classes are of the following form:

$$[ops : Sig\ Rep, prfs : Spec\ Rep\ ops] \rightarrow [ops : Sig\ Rep, prfs : Spec\ Rep\ ops]$$

In the setting of Section 2, the fixed point reached by iterating a class yielded both the desired final implementation and its correctness proof. In the present situation we will iterate the function $code : (Sig\ Rep) \rightarrow Sig\ Rep$ to obtain the final implementation and afterwards construct its correctness proof using the proof component of the class. More formally, we proceed as follows.

Suppose that we are given a class of type *Class ClassR Sig Spec′*. Applying it to *ClassR* and to $(refl_{\leq}\ ClassR)$ and projecting out the components yields two functions:

$$\begin{aligned} &code : (Sig\ Rep) \rightarrow Sig\ Rep \\ &prfs : \forall\ selfops, selfops' : Sig\ Rep\ . \\ &\qquad (Spec'\ Rep\ selfops\ selfops') \rightarrow \\ &\qquad\quad Spec'\ Rep\ (code\ selfops)\ \ selfops\ ] \end{aligned}$$

As before, we assume that our specification is consistent. That is, we assume $opsbasis : Sig\ ClassR$ and $prfsbasis : Spec'\ ClassR\ opsbasis\ opsbasis$. If, as assumed, there are no circular self-dependencies in the definition of *code* then there exists a natural number n such that — starting with *opsbasis* — after n iterations a fixed-point of the self-dependent operations *code* has been reached i.e. $(code\ (code^n\ opsbasis)) = (code^n\ opsbasis)$. Unlike in the setting of Section 2 we now have to explicitly require a proof *coderesolved* that the fixpoint has been reached after n iterations. Then the $(n+1)$-th iteration of the self-dependent proofs (starting with $prfsbasis : Spec'\ Rep\ opsbasis\ opsbasis$) has type $Spec'\ Rep\ (code^{n+1}\ opsbasis)\ (code^n\ opsbasis)$ which coincides with $Spec'\ Rep\ (code^n\ opsbasis)\ (code^n\ opsbasis)$. Hence the fixed-point of the operations satisfies the specification *Spec′*. If the iteration index n is such that after β-reduction both the n-fold iteration of *code* and the $(n+1)$-fold iteration of *prfs* do not contain the assumptions *opsbasis* and *prfsbasis*, respectively, then, as in Section 2, these are no longer needed.

Again, the task of finding such an index n could be carried out by a semi-algorithm which performs a brute force search or alternatively by a static analysis of the possible calls to *self*.

Definition. *[Instantiation] Assume implicitly a representation type ClassR, a signature Sig, and a generalized specification Spec′. Let the term Spec stand for the corresponding ungeneralized specification* $\lambda\ Rep : \star\ .\ \lambda\ ops : (Sig\ Rep)\ .\ Spec'\ Rep\ ops\ ops$. *The* instantiation operator *is thus defined as:*

$$\begin{aligned} new \stackrel{\text{def}}{=}\ & \lambda\ class : Class\ ClassR\ Sig\ Spec'\ . \\ & \lambda\ state : ClassR\ . \\ & \lambda\ opsbasis : Sig\ ClassR\ . \\ & \lambda\ prfsbasis : Spec'\ ClassR\ opsbasis\ opsbasis\ . \\ & \lambda\ n : nat\ . \\ & let \quad codeprfs\ =\ class\ ClassR\ (refl_{\leq}\ ClassR) \\ & \qquad operations\ =\ nat\_iter\ opsbasis\ codeprfs.code\ n \\ & in\ \ \lambda\ coderesolved :\ codeprfs.code\ operations =_L operations\ . \\ & \quad let\ proofs = new\_aux\ opsbasis\ prfsbasis\ n\ coderesolved \\ & \quad in\ \ (ObjectIntro\ state\ operations\ proofs) \\ & \qquad : Object\ Sig\ Spec \end{aligned}$$

FIG. 1. Instantiation

The function *new_aux* performs the abovementioned iteration of *codeprfs.prfs*, yielding an element of type *Spec′ ClassR operations operations*.

Example 9. [Instance of points] The instantiation for points remains basically unchanged. Again, only two iterations are needed to resolve the self-dependencies. Having reached the fixed point of code, the proof coderesolved is trivial by reflexivity of Leibniz's equality.

3.3. Inheritance

The last definition to align is the one for inheritance. The basic mechanisms remain unchanged, but the encoding now has to deal separately with the operations and the proofs. We also solve the problem of mixing inherited operations with newly implemented ones, encountered in the example on Page 10. The problem resembles the one that led to the redefinition of classes: there is no connection between the variable *superops*, denoting the operations of the superclass, and the newly implemented operations. The solution, though, is simpler than the one for *self*, since *superops* stands for an already existing implementation. In the function *build*, we simply have to make available the fact that *superops* really stands for the operations of the superclass.

Definition. *[Inheritance] Assume implicitly a representation type SuperR, a signature SuperSig, and a generalized specification SuperSpec′ of type* $\forall\, Rep : \star.\ (SuperSig\ Rep) \to (SuperSig\ Rep) \to \star$. *Assume a representation type SubR, a signature SubSig, and a generalized specification SubSpec′ for the subclass. Let the terms SuperSpec and SubSpec denote the ungeneralized specifications as in the definition of classes with generalized specifications on Page 12. Finally assume a proof* $gp_{SubR \le SuperR} : SubR \le SuperR$ *and coercions co_sig of type* $\forall\, Rep \mid \star.\ (SubSig\ Rep) \to (SuperSig\ Rep)$ *and co_spec of type*

$$\forall\, Rep \mid \star.\ \forall\, selfops, selfops' \mid (SubSig\ Rep).\ (SubSpec'\ Rep\ selfops\ selfops') \to (SuperSpec'\ Rep\ (co\_sig\ selfops)\ (co\_sig\ selfops'))$$

for the operations and proofs respectively. The inheritance operator *inherit is defined in Figure 2.*

The reader should not be discouraged by the sheer size of this term. As can be seen from Example 10, the programmer does not have to bother about the concrete definition of *inherit* in a concrete application.

As far as the readability of the programs is concerned, you have to remember that our programming language is rigorously encoded within a theorem prover. To make the language practically more useful, syntactic sugar and a number of special tactics are inevitable. See [44] for suggestions, how to make the encoding look like a generalized object-oriented programming language.

$$
\begin{array}{l}
inherit \stackrel{\text{def}}{=} \lambda\, SuperClass : Class\ SuperR\ SuperSig\ SuperSpec'.\\
\quad \lambda\, build : \forall\, Rep : \star.\ \forall\, gp_{Rep \le SubR} : Rep \le SubR.\\
\qquad let\ gp_{Rep \le SuperR} = trans_{\le}\ gp_{Rep \le SubR}\ gp_{SubR \le SuperR}\\
\qquad\quad super = SuperClass\ Rep\ gp_{Rep \le SuperR}\\
\qquad in\ [\ code : (SuperSig\ Rep) \to (SubSig\ Rep) \to SubSig\ Rep,\\
\qquad\quad prfs : \forall\, selfops, selfops' : SubSig\ Rep.\\
\qquad\qquad let\ superops =_L super.code\ (co\_sig\ selfops))\\
\qquad\qquad in\ (SuperSpec'\ Rep\ superops\ (co\_sig\ selfops)) \to\\
\qquad\qquad\quad (SubSpec'\ Rep\ selfops\ selfops') \to\\
\qquad\qquad\quad SubSpec'\ Rep\ (code\ superops\ selfops)\ selfops\ ]\\
\quad (\lambda\, Rep : \star.\ \lambda\, gp_{Rep \le SubR} : Rep \le SubR.\\
\qquad let\ gp_{Rep \le SuperR} = trans_{\le}\ gp_{Rep \le SubR}\ gp_{SubR \le SuperR}\\
\qquad\quad super = SuperClass\ Rep\ gp_{Rep \le SuperR}\\
\qquad\quad codeprfs = build\ Rep\ gp_{Rep \le SubR}\\
\qquad in\ [\ code = \lambda\, selfops : SubSig\ Rep.\ codeprfs.code\ (super.code\ (co\_sig\ selfops))\ selfops,\\
\qquad\quad prfs = \lambda\, selfops, selfops' : SubSig\ Rep.\\
\qquad\qquad \lambda\, selfprfs : SubSpec'\ Rep\ selfops\ selfops'.\\
\qquad\qquad codeprfs.prfs\ selfops\ selfops'\\
\qquad\qquad\qquad (super.prfs\ (co\_sig\ selfops)\ (co\_sig\ selfops')(co\_spec\ selfprfs))\\
\qquad\qquad\qquad selfprfs\ ]\\
\quad) : Class\ SubR\ SubSig\ SubSpec'
\end{array}
$$

FIG. 2. Inheritance

Continuing with the running example of colored points, the definitions of *CPoint*, *SigCPoint*, and *SpecPoint* from Section 1 go unchanged, we only need a generalized specification *SpecCPoint′*.

Example 10. [Colored points] For the generalized specification SpecCPoint′, assume an arbitrary representation type Rep, concrete operations ops, and abstract operations selfops of type SigCPoint Rep.

$$
\begin{array}{l}
SpecCPoint' \stackrel{\text{def}}{=} \\
(\ SpecPoint'\ Rep\ ops.opspoint\ selfops.opspoint\ ) \times \\
(\ selfops.getX\ (ops.inc2\ r) =_L (selfops.getX\ r) + 2 \\
\quad selfops.getC\ (ops.inc2\ r) =_L blue \\
\quad ops.getC\ (ops.setX\ r\ n) =_L blue\ )
\end{array}
$$

Now we can define a class MyCPointClass with representation type $(nat \times Color)$ *by means of the inheritance operator inherit. For the definition of* $gp : (nat \times Color) \leq nat$ *we refer to the example on Page 10. The two terms co_sig and co_spec denote the natural coercion functions from colored points to points.*

$$
\begin{array}{l}
MyCPointClass \stackrel{\text{def}}{=} \\
\quad inherit\ (nat \times Color)\ SigCPoint\ SpecCPoint' \\
\qquad gp\ co\_sig\ co\_spec \\
\qquad PointClass \\
\qquad (\ \lambda\, Rep : \star\, . \\
\qquad\quad \lambda\, gp_{Rep \leq (nat \times Color)} : Rep \leq (nat \times Color)\, . \\
\qquad\quad [\ code\ =\ codeCPointClass, \\
\qquad\qquad prfs\ =\ prfsCPointClass\ ]\ ) \\
\quad : Class\ (nat \times Color)\ SigCPoint\ SpecCPoint'
\end{array}
$$

The implementation of the operations is given by the quintuple opsCPointClass.

$$
\begin{array}{l}
codeCPointClass = \\
\quad \lambda\, superops :\ SigPoint\ Rep\, . \\
\quad \lambda\, selfops : SigCPoint\ Rep\, . \\
\quad [\ getX\ =\ \lambda r : Rep\, .\ superops.getX\ r, \\
\quad\ setX\ =\ \lambda r : Rep\, .\ \lambda n : nat\, . \\
\qquad\qquad gp_{Rep \leq (nat \times Color)}.put\ r\ (n, blue), \\
\quad\ inc1\ =\ \lambda r : Rep\, .\ superops.inc1\ r, \\
\quad\ inc2\ =\ \lambda r : Rep\, .\ superops.inc1\ (selfops.inc1\ r), \\
\quad\ getC\ =\ \lambda r : Rep\, .\ (gp_{Rep \leq (nat \times Color)}.get\ r).2\ ]
\end{array}
$$

With the modified encoding, all equations of the colored point class become provable. Assuming operations selfops, selfops′ : SigCPoint Rep of the subclass, the proofs superprfs of the superclass and the self proofs of colored points selfprfs, we have to give a correctness proof for the specification relatively to the operations just defined. In the following, we abbreviate the operations of colored points as Cops.

Since the setX operation has been reimplemented for colored points, the inherited proof of the first equation of the point class is of no use for proving the respective equation $Cops.getX(Cops.setX\ r\ n) =_L Cops.getX\ r$ *of the colored points. This equation β-reduces to:*

$$
\begin{array}{l}
(gp_{Rep \leq (nat \times Color)}.get \\
\quad (gp_{Rep \leq (nat \times Color)}.put\ \ r\ (n, blue))).1 =_L n
\end{array}
$$

This is immediate by the laws for get and put.

The new encoding with the generalized specifications still admits inheriting proofs for equations containing only inherited methods. So the proof for the second equation can instantly be obtained by superprfs.2, as in the example on Page 10.

The third equation can be proved by applying superprfs.1 twice. This example demonstrates that several references to the proofs of the superclass might be needed to establish one single equation in the subclass.

The fourth equation $selfops.getC\ \ (Cops.inc2\ r) =_L blue$ *expands into*

$$
\begin{array}{l}
selfops.getC \\
\quad (selfops.setX\ \ (selfops.inc1\ r) \\
\qquad\qquad ((selfops.getX\ \ (selfops.inc1\ r)) + 1)) \\
=_L blue
\end{array}
$$

Specialising the proof selfprfs.5 to $r = selfops.inc1\ r$ *and* $n = (selfops.getX\ \ (selfops.inc1\ r)) + 1$ *shows that the left hand side equals blue.*

The last equation $Cops.getC(Cops.setX\ r\ n) =_L blue$ *finally, containing only new methods or reimplemented ones, can be proven directly using the implementation of the colored point class.*

3.4. A more flexible definition of classes

We have also experimented with the following weaker, i.e. easier to implement, definition of classes:

$$
\begin{array}{l}
Class' \stackrel{\text{def}}{=} \forall\, Rep : \star\, .\ (Rep \leq ClassR) \rightarrow \\
\qquad [\ code : (Sig\ Rep) \rightarrow Sig\ Rep\, . \\
\qquad\ prfs : \forall\, selfops : Sig\ Rep\, . \\
\qquad\qquad (Spec'\ Rep\ selfops\ selfops) \rightarrow \\
\qquad\qquad\ Spec'\ Rep\ (code\ selfops)\ \ selfops\ ]
\end{array}
$$

As before, let *operations* stand for the fixpoint of *code* obtained by an appropriate number of iterations. Now, in order to obtain an element of *Spec′ ClassR operations operations* by iterating *prfs* we need to start with variable *prfsbasis* of type *Spec′ ClassR operations operations*, so it seems as if nothing has been gained. If, however, it so happens that the assumed variable drops out after a certain number of iterations then, again, we have shown the correctness of *operations* without assumptions.

So, in this case, β-reduction on proofs is really required for the soundness of the formalism.

Our experience is that the definition *Class′* is more flexible in case references to both self and super are made in the definition of one and the same method. For example, if we implement *inc2* in the example of colored points as

$$inc2 = superops.inc1\ (selfops.inc1\ \ r),$$

then in the formerly presented formalism a verification is possible only if we would include certain implementation details into the specification, whereas in the "alternative" one it goes through immediately without changes.

We find this essential use of β-reduction on proofs interesting and worth exploring but need more experience to gauge its possible merits and drawbacks.

4. Conclusion

Building upon the object-model of [47] and [23], this paper has presented a way to integrate formal verification into an object-oriented programming language. By augmenting the interface of objects by a specification of its behavior, we have demonstrated that object-oriented structuring techniques can be employed in organizing the proofs as well. Our experience so far has been that these object-oriented structuring techniques allow for flexible reuse of proofs. However, we have to concede that more substantial examples are required to justify this. A first step in this direction is the case study concerned with Smalltalk-style collections contained in [43].

In this case study we noticed that *abstract classes*, i.e. classes not meant to be instantiated into objects, are a natural mechanism to reduce the verification effort. In our setting, abstract classes are classes that contain methods or proofs referring to themselves via the variable *self* and which we call *abstract methods* or *abstract proofs*. Note that abstract classes do not denote a new concept, but refer to a special usage of the already encountered mechanism of *self-reference*.

To understand how abstract classes allow flexible reuse of methods, consider a class *Root* as root of the hierarchy of collection classes, which provides methods common to all collections. Let us just focus on four operations, common to all collections: *length*, *add*, *empty*, and *fold*, with the usual meaning. In the class *Root*, the three methods *fold*, *add*, and *empty* are abstract i.e. $fold = self.fold$, $add = self.add$, $empty = self.empty$, and the fourth method *length* is defined straightforwardly in terms of *fold*. No matter which implementation of *fold* we will eventually choose in a subclass, the corresponding *length* will always be defined.

An analogous mechanism — *abstract proofs* — supports flexible reuse of proofs. Consider the following induction principle:

$$(P\ empty) \to (\forall a, c.(P\ c) \to (P\ (add\ \ a\ c))) \to \forall c.(P\ c)$$

In the root class *Root* we cannot provide a proof of such an induction principle yet, since *empty* and *add* are merely abstract. Instead of proving this rule directly, we introduce an *abstract proof* defined as $induction = self.induction$. Using this not yet proven induction principle we can prove properties about the class *Root* which can be inherited by concrete subclasses. The mechanism of proof inheritance takes care that the proofs relying on it are adapted automatically in subclasses, depending on the specific implementation of the representation type, on the chosen operations, and on their correctness proofs (including the deferred proof of the induction principle for the concrete representations of non-abstract subclasses).

Comparison with other work

In Lego much work has been done in formalizing mathematical theories and also in the field of program specification and verification [34] [7] [50] [22] [50] [53], to mention several. Whilst there is an increasing body of work about the semantic foundations of object-oriented programming, notably in the area of typed functional calculi (see [20]), there are still only a few investigations about the verification of specifically object-oriented programs.

Leavens and Weihl in a series of papers [27, 28, 29, 31, 30] investigate modular specification and verification of object-oriented programs featuring subtype polymorphism and late-binding. Modular verification in their setting means: adding a new type to a program must not call for recoding, respecification, or reverification of old modules. In the presence of subtyping, the aim is to use the proofs for objects of the supertype also for objects of all subtypes without change. The problem with late binding methods for verification is that on the one hand one wishes a "static" verification of properties for objects of a given class, but on the other hand inheritance and late-binding of methods can lead to a different semantics in subsequent subclasses. The solution presented is to separate the implementation from its abstract representation, to assign a static type to the objects as upper bound (its *nominal* type), and use the abstract specification to reason about objects of all of its subtypes. Thus objects of a smaller type must not only accept messages meant for objects of a larger type without "message not understood" run-time error, but in addition they have to exhibit the same behavior, as given in the interface specification. Since structural subtyping — employed e.g. in $F^{\omega}_{\leq}$'s (sub-)type system — is too weak to account for compliance with specifications, the notion of subtyping needs a refinement; this stronger notion of behavior-preserving subtyping is known as *behavioral subtyping* [1]. To obtain a convenient mathematical model of the abstractly specified objects, they restrict their attention to objects with im-

mutable state which can be modelled as abstract data types. LOAL can handle *multiple dispatch* of methods, similar to the mechanisms in CLOS. Hoare style specification is used to specify the behavior of the objects via so-called *traits* in the Larch interface specification language as pre- and post-condition of the object's methods. An extension to types with mutable state and aliasing, in an algebraic framework, is presented in [15]. Sticking to an algebraic framework, though, in the presence of a mutable state seems to complicate the model considerably.

In contrast to the work of Leavens and Weihl, Utting [51] [52] handles objects with *mutable* state, but at the expense of data refinement, i.e. in the refinement process, inheritance may not change the internal representation of objects. A methodological difference is that he favors program development by a series of transformations. To this end an extension of *refinement calculus* of [3] [41] [40], being itself an extension of Dijkstra's guarded command language [16], is presented, a wide spectrum language, where executable code and specifications can be freely mixed.

Mairson [37] presents different object-oriented mechanisms encoded in the calculus of constructions. The emphasis there is not on program verification and its methodology, but on the analysis of languages of typed (record) calculi itself. Following the program extraction methodology, a couple of typed record and object calculi, notably Cardelli and Mitchell's record calculus [9], are represented equationally in the internal higher-order logic of the calculus of constructions. So, for example, extracting the computational content from the encoding of subtyping gives rise to the usual coercion functions. The encodings provide a logical justification for record calculi and object-oriented features like F-bounded polymorphism [8] or subtyping, and allow to investigate metamathematical properties such as soundness, consistency, and coherence of different encoded idioms. In contrast to our work, neither encapsulation, nor inheritance, nor late-binding are treated. Like in this paper, finite unwindings are employed to resolve the fixed points in the object encodings. To represent record types for objects, Hickey [21] introduces a new type constructor, which he calls "very dependent function type", which is "almost" a recursive type but he imposes well-foundedness conditions to avoid circularity. The approach is formalized in the NuPRL proof development system [12].

Further Work

Apart from the need for developing more extensive case studies, several directions for further work suggest themselves. Our approach is based on an encoding of one specific, albeit powerful and well-studied, object model. It would be worthwhile to the extend or change the encoding to comprise other object-oriented features or idioms, such as multiple inheritance, which can be modelled in an extension of $F^{\omega}_{\leq}$ with intersection types [11]. One could add syntactic sugar or fancier notions of specification, e.g. splitting the specification into a visible, external part, and an internal, hidden one, or to include matching [5] as a weaker relation than subtyping which seems to have advantages in inheriting binary methods.

A pragmatic path might be, to spare the user from perfoming every minute step of the required proofs. This could include the automatic generation of the *get* and *put* functions proposed in [23] for positive signatures, or the automatic calculation of the number of fixpoint unwindings. To perform larger case studies, a high-level syntax is inevitable that withholds Lego–specific notations and commands.

Besides verifying properties of actual programs, the transfer into Lego could also be used to prove general properties about the encoding itself, such as properties of the inheritance or instantiation operator.

A deeper question concerns the equality of objects. An intensional equality such as Leibniz's equality is inadequate for the comparison of objects, since it would distinguish between objects of different implementations, which contradicts the idea of encapsulation. As pointed out in Section 2.3.3, it is also problematic to place the test of equality on objects as an equality method inside the objects. In the chosen model, generic methods cannot be defined for signatures containing binary generic methods like a method comparing two objects whose internal representation is hidden by weak existential quantification. We believe that the correct equality for objects would be given by observational equivalence with respect to method invocations as advocated e.g. by Jacobs [25].

Acknowledgments

Thanks to Luis Dominguez, Michael Mendler, and Uwe Nestmann for giving useful suggestions on earlier versions which helped to improve the paper. We are grateful to the members of the Lego-club at the LFCS whose comments influenced the entire work. In particular we want to thank Rod Burstall, James McKinna, and Thomas Schreiber for their discussions. Finally we thank the two anonymous referees for their helpful remarks.

This research was supported by the *Spezifikation und Verifikation verteilter Systeme* project, funded by the *Deutsche Forschungsgemeinschaft*, Sonderforschungsbereich 182, and by the *Britsh Council* and the *Deutscher Akademischer Austauschdienst* within the ARC-programme "Ko-Entwicklung objektorientierter Programme in Lego".

Notes

1. The Lego-sources can be accessed by anonymous ftp at `ftp.informatik.uni-erlangen.de` in the directory `/local/inf7/vs/sfbc2/lego/oo-verification.l`
2. An extension to the encoding presented so far to specify such details has been presented in [42] using ideas of [23].

References

[1] Pierre America. A behavioural approach to subtyping in object-oriented programming languages. 443, Phillips Research Laboratories, January/April 1989.

[2] Philippe Audebaud. CC+: An extension of the Calculus of Constructions with fixpoints. Research Report 93-12, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon Unité de recherche associée au CNRS, July 1993.

[3] Ralph-Johan R. Back. *On the Correctness of Refinement in Program Development.* PhD thesis, Department of computer Science, University of Helsinki, 1978.

[4] Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov Gabbay, and Thomas Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1: Mathematical Structures, pages 117–309. Oxford University Press, 1992.

[5] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2), April 1994. A preliminary version appeared in POPL 1993 under the title "Safe Type Checking in a Statically Typed Object-Oriented Programming Language", and as Williams College Technical Report CS-92-01.

[6] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[7] Rod Burstall and James McKinna. Deliverables: a categorical approach to program development in type theory. In A. M. Borzyszkowski and S. Sokołowski, editors, *Eighteenth Mathematical Foundations of Computer Science (Gdansk, Poland)*, volume 711 of *Lecture Notes in Computer Science*, pages 32–67. Springer, September 1993.

[8] Peter Canning, William Cook, Walt Hill, and Walter Olthoff John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Fourth ACM Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, pages 273–280. ACM, Springer, September 1989.

[9] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in the collection [20]; available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[10] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[11] Adriana B. Compagnoni and Benjamin C. Pierce. Intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, October 1996. Preliminary version available as University of Edinburgh technical report ECS-LFCS-93-275 and Catholic University Nijmegen computer science technical report 93-18, Aug. 1993, under the title *Multiple Inheritance via Intersection Types.*

[12] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice Hall, 1986.

[13] C. Cornes, J. Courant, J-C. Filiatre, G. Huet, P. Manoury, C. Mounoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant User's Guide. Rapports Techniques 0177, INRIA Rocquencourt, Projet Formel, 1995. Version 5.10.

[14] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption. Technical Report LIENS 90-10, Laboratoire d'Informatique de l'Ecole Normale Supérieure, February 1990.

[15] Krishna Kishore Dhara and Gary T. Leavens. Subtyping for mutable types in object-oriented programming languages. Technical report, Iowa State University, Department of Computer Science, November 1992. TR 92-36, submitted to ECOOP '93.

[16] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[17] Kathleen Fisher and John Mitchell. Notes on typed object-oriented programming. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 844–885. Springer, 1994.

[18] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupure dans l'arithmetique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[19] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implemementation.* Addison-Wesley, Reading, MA, 1983.

[20] Carl A. Gunter and John C. Mitchell. *Theoretical Aspects of Object-Oriented Programming, Types, Semantics, and Language Design.* Foundations of Computing Series. MIT Press, 1994.

[21] Jason J. Hickey. Formal objects in type theory using very dependent types. In Kim Bruce, editor, *Informal Proceedings of the Third International Workshop on Foundations of Object-Oriented Languages (FOOL'96).* Cornell University, Department of Computer Science, August 1996. Available electronically through `http://www.cs.williams.edu/~kim/FOOL/Abstracts.html`.

[22] Martin Hofmann. Formal development of functional programs in type theory — a case study. Report ECS-LFCS-92-228, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

[23] Martin Hofmann and Benjamin Pierce. Positive subtyping. In *22nd Annual Symposium on Principles of Programming Languages (POPL) (San Francisco, California)*, pages 186–197. ACM, ACM Press, January 1995. Full version in *Information and Computation*, volume 126, number 1, April 1996. Also available as University of Edinburgh technical report ECS-LFCS-94-303, September 1994.

[24] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995. Previous versions appeared in the Symposium on Theoretical Aspects of Computer Science, 1994, (pages 251–262) and, under the title "An Abstract View of Objects and Subtyping (Preliminary Report)," as University of Edinburgh, LFCS technical report ECS-LFCS-92-226, 1992.

[25] Bart Jacobs. Objects and classes, coalgebraically. In C. B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-orientation with parallelism and persistence.* Kluwer Acad. Publ., 1996.

[26] Claire Jones and Savi Maharaj. The LEGO library. Available on the World Wide Web [32], February 1994.

[27] Gary T. Leavens. *Verifying Object-Oriented Programs that use Subtypes.* PhD thesis, Massachusetts Institute of Technology, 1988.

[28] Gary T. Leavens. Modular verification of object-oriented programs with subtypes. Technical Report 90-09, Iowa State University, Department of Computer Science, July 1990.

[29] Gary T. Leavens. Specifying and verifying object-oriented programs: an overview of the problems and a solution. Technical report, Iowa State University, Department of Computer Science, February 1991.

[30] Gary T. Leavens. Inheritance of interface specifications. Technical Report TR 93-23, Iowa State University, Department of Computer Science, September 1993. (Extended Abstract).

[31] Gary T. Leavens and William E. Wheil. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 1994. An expanded version appeared as Iowa State Unversity Report, 92-28d.

[32] The Lego World Wide Web page, 1997. Accessible through `http://www.dcs.ed.ac.uk/home/lego`.

[33] Zhaohui Luo. An extended Calculus of Constructions. Thesis ECS-LFCS-90-118, Laboratory for Foundations of Computer Science, University of Edinburgh, July 1990.

[34] Zhaohui Luo. A unifying theory of dependent types: the schematic approach. Technical Report ECS-LFCS-92-202, Laboratory for Foundations of Computer Science, University of Edinburgh, March 1992.

[35] Zhaohui Luo and Randy Pollack. Lego proof development system: User's manual. Technical Report ECS-LFCS-92-211, Laboratory for Foundations of Computer Science, University of Edinburgh, May 1992.

[36] David MacQueen. Using dependent types to express modular structure. In *Thirteenth Annual Symposium on Principles of Programming Languages (POPL) (St. Peterburg Beach, FL)*, pages 277–286. ACM, ACM Press, January 1986.

[37] Harry G. Mairson. A Constructive Logic of Multiple Inheritance. In *Twentieth Annual Symposium on Principles of Programming Languages (POPL) (Charleston, SC)*, pages 313–324. ACM, ACM Press, January 1993.

[38] John C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Seventeenth Annual Symposium on Principles of Programming Languages (POPL) (San Fancisco, CA)*, pages 109–124. ACM, ACM Press, January 1990. Also in the collection [20].

[39] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[40] Carrol C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.

[41] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[42] Wolfgang Naraschewski. Object-oriented proving. In *Programs & Proofs: Working in Type Theory, Hetzelsdorf*, 14–18 August 1995.

[43] Wolfgang Naraschewski. *Object-Oriented Proof Principles using the Proof-Assistant Lego*. Diplomarbeit, Universität Erlangen, 1996.

[44] Wolfgang Naraschewski. Towards an object-oriented progification language. In *Proceedings of the 1997 International Conference in Theorem Proving in Higher Order Logics, Bell Labs, Murray Hill, NJ, USA*, 1997. To appear as Springer Lecture Notes in Computer Science. Available through `http://www4.informatik.tu-muenchen.de/~narasche/TPHOL/`.

[45] Benjamin Pierce. *F-Omega-Sub User's Manual, Version 1.4*, February 1993. Available by FTP as part of the `fomega` implementation.

[46] Benjamin Pierce and David Turner. Object-oriented programming without recursive types. Technical Report ECS-LFCS-92-225, Laboratory for Foundations of Computer Science, University of Edinburgh, August 1992. See also *Principles of Programming Languages (POPL '93)*.

[47] Benjamin Pierce and David Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994. A preliminary version appeared in Principles of Programming Languages, 1993, and as University of Edinburgh technical report ECS-LFCS-92-225, under the title "Object-Oriented Programming Without Recursive Types".

[48] Berhard Reus. *Program verification in Synthetic Domain Theory*. PhD thesis, LMU, München, 1995.

[49] John Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Colloque sur la programmation (Paris, France)*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.

[50] Thomas Schreiber. *Verifikation von imperativen Programmen mit dem Beweisprüfer* Lego. Diplomarbeit, Universität Erlangen, 1993.

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

[52] Mark Utting and Ken Robinson. Modular reasoning in an object-oriented refinement calculus. In R. S. Bird, C. C. Morgan, and J. P. C. Woodcock, editors, *Mathematics of Program Construction 1992*, volume 669 of *Lecture Notes in Computer Science*, pages 344–367. Springer, 1993.

[53] P. Wand. Functional programming and verification with Lego. Master's thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.