Inheritance of Proofs

Martin Hofmann* Wolfgang Naraschewski†
Martin Steffen Terry Stroup

Arbeitsgruppe Allgemeine Algebra
Fachbereich Mathematik, Technische Hochschule Darmstadt
Schloßgartenstraße 7, D-64289 Darmstadt, Germany

and

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

and

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

June 1996

Abstract

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 proof 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.

*`mh@mathematik.th-darmstadt.de`
†`{wgnarasc|steffen|terry}@immd7.informatik.uni-erlangen.de`

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 allow divide-and-conquer strategies to break down large programs into abstract data types, modules, objects, or similar. Since the resulting components ideally mirror the decomposition of the problem in conceptually self-contained units, it is natural to organise verification along the structure of the programs.

The most successful and mathematically well-founded approaches are algebraic specifications, wide-spectrum languages, and type theory. In the field of *algebraic specification* (see e.g. [Wir90] for an overview), a rich body of refinement notions has been developed, supporting behavioural abstractions and horizontal and vertical refinement steps. These allow to design large programs by stepwise refinement and to decompose their correctness proofs along the refinement and module structure. A number of approaches aim at enhancing algebraic design languages with object-oriented constructs, among these Foops [GM87] (an extension of OBJ), GSBL [CO88], Object-Z [DDRS89], Spectral [KS91], and OOZE [AG91].

Wide-spectrum languages provide a common framework for programs and specifications, where the class of programs is regarded as the executable subset of specifications. Extended ML [ST86] [ST91], as prominent example, allows to combine the functional programming language ML and its module mechanism with logical specifications, structuring the specifications along the structure of the modules. The verification, though, does not take place in Extended ML itself, but has to be carried out externally.

An integration of programs and constructive logic into a single formal system is provided by *type theory*, making it appealing for specification and verification of programs. A fundamental property, shared by many typed λ-calculi, is the correspondence between logical propositions and types, or proofs and programs respectively, captured by the Curry-Howard isomorphism [CF58, How80]. This analogy is also known as propositions-as-types or programs-as-proofs. Based on different type theories, a number of tools for machine-assisted reasoning have been developed, e.g. AUTOMATH [dB80], Coq [DFH$^+$93], NuPRL [Jac94], Alf [AGNvS94], and Lego [LP92]. By exploiting Curry-Howard's isomorphism, they allow interactive development of mechanically verified programs.

The typed λ-calculus $F^{\omega}_{\leq}$ has been proposed [Pie92] as core calculus for object-oriented programming languages. It is sufficiently expressive to capture the essential mechanisms of class-based object-oriented languages such as Smalltalk, namely subtyping, encapsulation, class inheritance, and late binding, but for specifying the operations dependent types are indispensable.

In this paper, we extend the object model of [PT94], enriching objects with a proof component. To this end, we represent its object model in Lego and extend the objects' interfaces by specifications of its methods. These specifications encompass in particular the equational axiomatisation of the Pierce-Turner inheritance mechanism given in [HP96]. Objects consequently not only contain a state and an implementation of the methods, but proofs of their correctness relatively to the corresponding specification as well. The pairing of a program together with its correctness proof into one package is called a deliverable [BM93]. Taking the proofs as an integral part of objects allows applying object-oriented mechanisms mentioned above to *proofs* as well. The present work is closely related to [HP96] and can be seen as a further development and elaboration of ideas expressed there. In particular, our approach builds on the equational axiomatisation of coercion and update and on the idea of "inheriting proofs" explained by way of example there. In particular, our approach builds on the equational axiomatisation 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 formalisation the present paper extends [HP96] by providing a more general framework for the specification and verification of late-binding methods.

The remainder of the paper is organised 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, designed and implemented by Randy Pollack, is an interactive, type theoretic proof checker. It realises the extended calculus of constructions [Luo90] and a family of weaker, related type theories: the Edinburgh Logical Framework [HHP93], the calculus of constructions [CH88], and the generalised calculus of constructions [Coq86]. The system comprises a strongly typed functional programming language as well as a higher-order intuitionistic logic. The extended calculus of constructions uses a predicative hierarchy of universes for programming and an impredicative universe for logical propositions. Each term is strongly normalising, forcing all definable programs to terminate. 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, allowing to omit automatically synthesizable function arguments. Furthermore, the user is given the freedom to add arbitrary reductions. For an introduction into Lego, the reader is referred to the Lego-manual [LP92] or the web resources [Lego95].

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 $\star$. We fall back upon definitions provided by the Lego-library [JM94] whenever appropriate. For the inductive strong sigma type of the library `lib_sigma/lib_sigma.l` we write Σ and $\langle \_ , \_ \rangle$ for the respective dependent pair, omitting the type-annotations. 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's 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 that simplifies definitions, synthesising omitted arguments on its own. We shall use the $|$-symbol to indicate implicit arguments as in Lego.

In most definitions, we do not give the whole expression as a λ-term, but put some leading abstractions into the text. Finally, we elide conjunctions between displayed equations. Apart from these conventions, though, all definitions are complete and can be 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 [CW85] for the typed object-oriented toy language Fun based on $F_{\leq}$, an extension of the second order polymorphic λ-calculus [Gir72, Rey74] 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 [FM94], a collection of relevant papers in [GM94].

For our purpose of integrating an object calculus into a logical framework, one particular formal system, the system $F^{\omega}_{\leq}$ [Pie92] is a suitable basis, since it avoids the complexity of calculi with recursive types [Bru92, Mit90]. Moreover, introducing a general fixed point constructor into a logical system such as Lego does not simply complicate the presentation, but makes any logical proposition provable, rendering the system inconsistent. $F^{\omega}_{\leq}$, the ω-order extension of $F_{\leq}$, has been proposed by Pierce and Turner [PT92, PT94, HP92] as a core calculus for object-oriented languages in the style of Smalltalk [GR83]. 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 [Pie92, PT94].

An *object* is a collection of operations, working on an internal *state*. Both state and

[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`

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 [MP88], 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*. The resulting type of objects with signature $Sig : \star \to \star$ is

$$Object = \exists\, Rep : \star .\ Rep \times Sig\ Rep\,.$$

With existential quantification as top-level constructor of the type of objects, the introduction and elimination rules for the existential quantifier will be used to create new objects (existential introduction) or to gain access to the internal operations by method invocation (existential elimination).

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*. Secondly, 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 added to unchanged, inherited ones.

An important intricacy are the so-called *self-methods*. This concept, popular since Smalltalk, allows 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, the *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 in this model, there is a connection between them: 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 behaviour 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 behaviour. 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 [Bar92] the ω-order λ-calculus F^{ω} [Gir72] is a subcalculus of the calculus of constructions. Subtyping, though an integral part of $F^{\omega}_{\leq}$, is neither present in the 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 [HP96], 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 *positive* variant of $F_{\leq}$ in [HP96]. The interpretation of *get* and *put* as extraction and update functions is captured by the following three equations.

Definition 2.1 (Laws for get and put [HP96]) 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 three equations:

$$\forall s : S, t : T \quad get\,(put\; s\; t) =_L t \qquad (1)$$
$$\forall s : S \quad put\; s\,(get\; s) =_L s \qquad (2)$$
$$\forall s : S, t_1, t_2 : T \quad put\,(put\; s\; t_1)\; t_2 =_L put\; s\; t_2 \qquad (3)$$

The term *GetPutLaws* of type $\forall S \,|\, \star.\ \forall T \,|\, \star.\ (S \to T) \to (S \to T \to S) \to \star$ is the Lego-representation of the above definition, used to define the subtype relation.[2]

Definition 2.2 (Subtype relation) Assume the types $S, T : \star$. The *subtype relation* is then defined as:

$$S \leq T \stackrel{\text{def}}{=} \Sigma\, get : S \to T . \Sigma\, put : S \to T \to S .\ GetPutLaws\ get\ put$$

[2] Recall that Lego's |-syntax for implicit arguments allows us to omit mentioning the first two arguments of *GetPutLaws*.

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

Lemma 2.1 (Pre-order) For all S, T, and U of type $\star$, $S \leq S$, and if $S \leq T$ and $T \leq U$, then $S \leq U$.

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 $get(gp_{S \leq T})$ with $get(gp_{T \leq U})$. The update function is composed as $\lambda s : S.\ \lambda u : U.\ put\, gp_{S \leq T}\ s(put(gp_{T \leq U}\ (get(gp_{S \leq T}\ s))\ u))$. Proving the respective laws is straightforward. □

We shall refer to the corresponding Lego-proofs by the terms $refl_{\leq} : \forall S : \star.\ S \leq S$ and $trans_{\leq} : \forall S, T, U \,|\, \star.\ (S \leq T) \to (T \leq U) \to (S \leq U)$.

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* of type $\forall\, Rep : \star.\ (Sig\ Rep) \to \star$ which specifies properties of the object in terms of its operations. Given a representation type *Rep*, the interface is thus written as *dependent product* $\Sigma\, ops : Sig\ Rep\ .\ Spec\ Rep\ ops$ of the functional signature and the specification. Hence the body of an object has type $Rep \times \Sigma\, ops : Sig\ Rep\ .\ Spec\ Rep\ ops$ and consists of a state together with a dependent pair of 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 the existential quantifier of $F^{\omega}_{\leq}$ to hide the internal state and the operations. No existential quantifier is built into the calculus of constructions or Lego; there are, however, different ways to encode existential quantifiers or weak dependent sums. A first attempt could be just to use "the same" existential quantifier as in $F^{\omega}_{\leq}$, i.e. the standard *impredicative encoding* of the weak sum:

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

This encoding is expressive enough as long as the interface of objects includes a solely functional signature. We could use this impredicative encoding even with specifications in the interface, if the only goal were to *introduce* objects. But, as already mentioned, we also need a mechanism to access the internal operations and the proofs. For arbitrary operations *ops* of type *Sig Rep*, this is achieved by the external counterparts of these operations, the so-called generic methods *meths* of type *Sig Object*. Here, *Object* stands for the type of objects of the given signature and specification, i.e. an existentially quantified type. In the same way as the generic methods represent the outside view of the operations, we need an externalised version of the proofs, turning *Spec Rep ops* into *Spec Object meths*.

Transforming the internal operations and proofs into their external, generic analogues means in short: existential *elimination*. For the above existential quantifier, as for the impredicative encodings of other data types [BB85, Wra89], the elimination function of type

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

is reflected by the encoding $\lambda C:\star.\,\lambda f:(\forall R:\star.\,(P\,R)\to C).\,\lambda o:(\exists R:\star.\,P\,R).\,o\,C\,f$ itself, where $P:\star\to\star$ is an arbitrary predicate.[3]

For the proof methods, the result type C of the elimination has to speak about objects; after all, we are interested in proving properties of objects. Hence to be useful for generic proof methods the elimination function for all predicates $P:\star\to\star$ has to be of type:

$$\forall C:(\exists R:\star.\,P\,R)\to\star.\,(\forall R:\star.\,\forall x:P\,R.(C\;(\mathit{pack}\;P\;R\;x)))\to\forall o:(\exists R:\star.\,P\,R).\,C\;o$$

where $\mathit{pack}:\forall P:\star\to\star.\,\forall R:\star.\,(P\,R)\to\exists R:\star.\,P\,R$ is the function for introducing existential quantifiers. It is, however, not possible to give a term of this type, i.e. an impredicative encoding of the elimination rule expressed in this type. To do so would require the induction principle: "If C holds for all elements built by the type constructor *pack*, then C holds for all p of the existential type." It is a well-known weakness of impredicative encodings that they do not provide such induction principles. In other words: there is *no* impredicative encoding of data types where the result type of the elimination rule depends on the elements of the type to be eliminated.

A solution is to add the formation, the introduction, and the elimination rule of the existential quantifier to the context by *declaration* and determine the computational meaning by Lego's *reduction rules*. This, of course, means that we are leaving the setting of the extended calculus of constructions.

Definition 2.3 (Existential quantification) The formation, the introduction, and the elimination rule for the type constructor $\exists$ are declared as follows:

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

Assume a predicate $P:\star\to\star$, a predicate $C:(\exists R:\star.\,P\,R)\to\star$ and a function f of type $\forall R:\star.\forall x:P\,R.C\;(\mathit{pack}\,P\;R\;x)$. Assume further $R:\star$ and $x:P\,R$. The reduction rule is then defined as:

$$\mathit{open}\,P\,C\;f\;(\mathit{pack}\,P\,R\,x)\Rightarrow f\;R\;x$$

[3] We use the more familiar notation $\exists R:\star.\,P\,R$ instead of $\exists(\lambda R:\star.\,P\,R)$.

This existential quantifier can be soundly interpreted in the PER/ω-set model of the extended calculus of constructions [Luo90] as follows. If F is a function mapping PER's to PER's, define $\exists(F)$ as the symmetric, transitive closure of the union of the $F(R)$ as R ranges over the set of PER's. This is the least upper bound of the $F(R)$ in the complete lattice of the PER's 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 PER's 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 yields the desired interpretation of *open*. This argument shows that — as far as equational soundness is concerned — we can even replace *pack* and *open* by subtyping rules of the form

$$\frac{? \vdash F : \star \to \star}{?\,, X : \star \vdash F(X) \leq \exists(F)}$$

$$\frac{? \vdash F : \star \to \star \qquad ? \vdash C : \exists(F) \to \star}{? \vdash \forall X : \star .\, \forall f : F(X).\, C(f) \leq \forall g : \exists(F).\, C(g)}$$

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 [DFH+93] and restrict the dependent elimination rule for existentials to predicates $C : \exists(F) \to \star$ while keeping the traditional non-dependent eliminator for C of type *Set*. 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 have to duplicate various definitions and rules and also that the Lego implementation does not provide *Set* and $\star$.

With this type constructor we can now define the type of objects.

Definition 2.4 (Type of objects) Assuming a signature $Sig : \star \to \star$ and a specification $Spec : \forall\, Rep : \star .\ (Sig\ Rep) \to \star$, the *type of objects* is given as:

$$Object \ \stackrel{\text{def}}{=}\ \exists\, Rep : \star .\ Rep \times \Sigma\, ops : Sig\ Rep\ .\ Spec\ Rep\ ops$$

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 2.5 (Object introduction) Assuming implicitly a representation type *Rep*, a signature *Sig*, and a specification *Spec*, the term for *object introduction* is defined as:

$$
\begin{array}{lcl}
\mathit{ObjectIntro} & \stackrel{\text{def}}{=} & \lambda\, \mathit{state} : \mathit{Rep}\,.\ \lambda\, \mathit{ops} : \mathit{Sig}\ \mathit{Rep}\,.\ \lambda\, \mathit{prfs} : \mathit{Spec}\ \mathit{Rep}\ \mathit{ops}\,. \\
& & \quad \mathit{pack}\ \ (\lambda\, \mathit{Rep} : \star.\ \ \mathit{Rep} \times \Sigma\, \mathit{ops} : \mathit{Sig}\ \mathit{Rep}\,.\ \mathit{Spec}\ \mathit{Rep}\ \mathit{ops}) \\
& & \qquad\quad \mathit{Rep} \\
& & \qquad\quad (\mathit{state}, \langle \mathit{ops}, \mathit{prfs} \rangle) \\
& : & \mathit{Rep} \to \forall\, \mathit{ops} : \mathit{Sig}\ \mathit{Rep}\,.\ (\mathit{Spec}\ \mathit{Rep}\ \mathit{ops}) \to \mathit{Object}\ \mathit{Sig}\ \mathit{Spec}
\end{array}
$$

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 2.6 (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*:

$$
\mathit{SigPoint} \stackrel{\text{def}}{=} \lambda\, \mathit{Rep} : \star.\ \underbrace{(\mathit{Rep} \to \mathit{nat})}_{\mathit{getX}} \times \underbrace{(\mathit{Rep} \to \mathit{nat} \to \mathit{Rep})}_{\mathit{setX}} \times \underbrace{(\mathit{Rep} \to \mathit{Rep})}_{\mathit{inc1}}
$$

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:

$$
\begin{array}{lll}
\mathit{SpecPoint} \stackrel{\text{def}}{=} \forall r : \mathit{Rep}\,.\ \forall n : \mathit{nat}\,. & \mathit{ops}\,.\,\mathit{getX}(\mathit{ops}\,.\,\mathit{setX}\ \ r\ n) =_L n \\
\qquad\qquad\quad\ \forall r : \mathit{Rep}\,. & \quad \mathit{ops}\,.\,\mathit{getX}(\mathit{ops}\,.\,\mathit{inc1}\ \ r) =_L (\mathit{ops}\,.\,\mathit{getX}\ \ r) + 1
\end{array}
$$

Let the terms *getX*, *setX* and *inc1* abbreviate the respective projection functions from triples of operations. The type of points *Point* is defined with the type constructor *Object*.

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

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

$$
\mathit{opsPoint} \stackrel{\text{def}}{=} (\lambda n : \mathit{nat}\,.\ n, \lambda n : \mathit{nat}\,.\ \lambda m : \mathit{nat}\,.\ m, \lambda n : \mathit{nat}\,.\ n + 1) : \mathit{SigPoint}\ \mathit{nat}\,.
$$

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*:

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

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 and proofs to perform the requested manipulations. If the operations *ops* of an object have type *Sig Rep*, the type of the generic functional methods *meths* is *Sig* (*Object Sig Spec*). The generic version of proofs of *Spec Rep ops* has type *Spec* (*Object Sig Spec*) *meths*. In the point example, the generic methods *methsPoint* have type $SigPoint\ Point = (Point \to nat) \times (Point \to nat \to Point) \times (Point \to Point)$ and the generic version of the first equation is $\forall p\!:\!Point\,.\ \forall n\!:\!nat\,.\ methsPoint\,.\,getX(methsPoint\,.\,setX\ \ p\ n) =_L n$. 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 [Pie92] 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. In contrast to the model of Pierce und Turner we have to deal with the proof-part as well, assigning the extraction function the type $\forall\, Rep\!:\!\star\,.\ (\Sigma\, ops\!:\!Sig\ Rep\,.\ Spec\ Rep\ ops) \to \Sigma\, ops\!:\!SigPoint\ Rep\,.\ SpecPoint\ Rep\ ops$.

Example 2.7 (Generic methods for points) Assume implicitly a signature $Sig\!:\!\star \to \star$ and a specification $Spec\!:\!\forall\, Rep\!:\!\star\,.\ (Sig\ Rep) \to \star$. Assume further a function *coercion* of type $\forall\, Rep\!:\!\star\,.\ (\Sigma\, ops\!:\!Sig\ Rep\,.\ Spec\ Rep\ ops) \to \Sigma\, ops\!:\!SigPoint\ Rep\,.\ SpecPoint\ Rep\ ops$. To ease readability we abbreviate the first and second projection function of the Σ type by *ops* and *prfs* respectively. The generic method $Point'setX$ is defined as follows:

$$
\begin{array}{lcl}
Point'setX & \stackrel{\text{def}}{=} & \lambda\, o : Object\ Sig\ Spec\ .\\
& & \lambda\, n : nat\ .\\
& & open\ (\lambda\, Rep : \star .\ \ Rep \times \Sigma\ ops : Sig\ Rep\ .\ Spec\ Rep\ ops)\\
& & \qquad (\lambda\_ : Object\ Sig\ Spec\ .\ \ Object\ Sig\ Spec)\\
& & \qquad (\lambda\, Rep : \star .\\
& & \qquad\ \lambda\, stateopsprfs : Rep \times \Sigma\ ops : Sig\ Rep\ .\ Spec\ Rep\ ops\ .\\
& & \qquad\quad let\ \ state \quad = \ stateopsprfs\ .1\\
& & \qquad\qquad\quad opsprfs \ = \ stateopsprfs\ .2\\
& & \qquad\quad in\ ObjectIntro\ ((coercion\ Rep\ opsprfs).\ ops\ .\ setX\ state\ n)\\
& & \qquad\qquad\qquad\qquad\qquad opsprfs\ .\ ops\\
& & \qquad\qquad\qquad\qquad\qquad opsprfs\ .\ prfs)\\
& & \qquad o\\
& : & (Object\ Sig\ Spec) \to nat \to (Object\ Sig\ Spec)
\end{array}
$$

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.

Example 2.8 (Generic proof methods for points) As in the previous example, assume implicitly a signature *Sig*, a specification *Spec*, and a coercion function *coercion*. The generic proof method for the first equation $Point'\,prf_1$ is defined as follows:

$$
\begin{array}{lcl}
Point'\,prf_1 & \stackrel{\text{def}}{=} & \lambda\, o : Object\ Sig\ Spec\ .\\
& & \lambda\, n : nat\ .\\
& & open\ (\lambda\, Rep : \star .\ \ Rep \times \Sigma\ ops : Sig\ Rep\ .\ Spec\ Rep\ ops)\\
& & \qquad (\lambda o' : Object\ Sig\ Spec\ .\\
& & \qquad\quad Point'\,getX\ coercion(Point'setX\ coercion\ o'\ n) =_L n)\\
& & \qquad (\lambda\, Rep : \star .\\
& & \qquad\ \lambda\, stateopsprfs : Rep \times \Sigma\ ops : Sig\ Rep\ .\ Spec\ Rep\ ops\ .\\
& & \qquad\quad let\ \ state \qquad\quad = \ stateopsprfs\ .1\\
& & \qquad\qquad\quad opsprfs \qquad = \ stateopsprfs\ .2\\
& & \qquad\qquad\quad opsprfsPoint \ = \ coercion\ Rep\ opsprfs\\
& & \qquad\quad in\ opsprfsPoint\ .\ prfs\ .1\ \ state\ \ n)\\
& & \qquad o\\
& : & \forall o : Object\ Sig\ Spec\ .\ \forall n : nat\ .\\
& & \quad Point'\,getX\ coercion\ (Point'setX\ coercion\ o\ n) =_L n
\end{array}
$$

The generic proof $Point'prf_2$ of the second equation has type $\forall o : Object\ Sig\ Spec\ .\ \ \forall n : nat\ .\ Point'\,getX\ coercion\ \ (Point'\,inc1\ coercion\ r) =_L (Point'\,getX\ coercion\ r) + 1$ 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 [HP92], namely for signatures of the form $\lambda\, Rep\!:\!\star.\ Rep \to (T\ \ Rep)$, where T is *positive* in its argument Rep.

The restriction to positive 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. The price for using weak existential quantification for hiding has been discussed already for abstract data types in [MP88] and [Mac86]. (Cf. [BCC+95] for a detailed discussion of problems related with binary methods in typed object-oriented programming languages.)

In the example of points, we were able to define the generic functional methods, since the signature is in principle of the above form. Instead of $SigPoint$, we could have used $\lambda\, Rep\!:\!\star.\ Rep \to (nat \times (nat \to Rep) \times Rep)$ as well; for presentational purposes, we have chosen the form of signature from Definition 2.6.

Objects without proof components In the previous sections we have emphasised the advantage 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 given in [PT94].

Definition 2.9 (Type of objects without proof component) Assuming a signature $Sig\!:\!\star \to \star$ the *type of objects without proof component* is given as:

$$Object\_eff \ \stackrel{\text{def}}{=}\ \exists\, Rep\!:\!\star.\ Rep \times (Sig\ Rep)$$

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 2.10 (Objects without proof component) Assuming implicitly a representation type Rep, a signature Sig, and a specification $Spec$, the term for *forgetting the* *proof component* is defined as:

$$
\begin{array}{lll}
\mathit{forget\_prfs} & \stackrel{\text{def}}{=} & \lambda\, o : \mathit{Object\ Sig\ Spec}\,.\\
 & & \quad \mathit{open}\ (\lambda\, \mathit{Rep} : \star .\ \ \mathit{Rep} \times \Sigma\, \mathit{ops} : \mathit{Sig\ Rep}\,.\, \mathit{Spec\ Rep\ ops})\\
 & & \qquad (\lambda \_ : \mathit{Object\ Sig\ Spec}\,.\ \ \mathit{Object\_eff\ Sig})\\
 & & \qquad (\lambda\, \mathit{Rep} : \star .\\
 & & \qquad\ \ \lambda\, \mathit{stateopsprfs} : \mathit{Rep} \times \Sigma\, \mathit{ops} : \mathit{Sig\ Rep}\,.\, \mathit{Spec\ Rep\ ops}\,.\\
 & & \qquad\qquad \mathit{let}\ \ \mathit{state} = \mathit{stateopsprfs}\,.1\\
 & & \qquad\qquad\qquad \mathit{operations} = \mathit{stateopsprfs}\,.2.\mathit{ops}\\
 & & \qquad\qquad \mathit{in}\ \ \mathit{pack}\ \ (\lambda\, \mathit{Rep} : \star .\ \ \mathit{Rep} \times (\mathit{Sig\ Rep}))\\
 & & \qquad\qquad\qquad\qquad \mathit{Rep}\\
 & & \qquad\qquad\qquad\qquad (\mathit{state}, \mathit{operations}))\\
 & & \qquad o\\
 & : & (\mathit{Object\ Sig\ Spec}) \to (\mathit{Object\_eff\ Sig})
\end{array}
$$

2.3.3 Classes

As informally explained 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 not only to provide 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 non-trivial 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 Definition 2.1 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 $\forall\, \mathit{Rep} : \star .\ (\mathit{Rep} \leq \mathit{ClassR}) \to \Sigma\, \mathit{ops} : \mathit{Sig\ Rep}\,.\, \mathit{Spec\ Rep\ ops}$.

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 $\Sigma\, \mathit{ops} : \mathit{Sig\ Rep}\,.\, \mathit{Spec\ Rep\ ops}$, i.e. the implementation is abstracted over this variable, giving classes the following type.

Definition 2.11 (Type of classes) Assume a representation type $\mathit{ClassR} : \star$, a signature $\mathit{Sig} : \star \to \star$, and a specification $\mathit{Spec} : \forall\, \mathit{Rep} : \star .\ (\mathit{Sig\ Rep}) \to \star$. The *type of classes* is given as:

$$
\begin{array}{lll}
\mathit{Class} & \stackrel{\text{def}}{=} & \forall\, \mathit{Rep} : \star .\ (\mathit{Rep} \leq \mathit{ClassR}) \to\\
 & & \qquad \Sigma\, \mathit{ops} : \mathit{Sig\ Rep}\,.\, \mathit{Spec\ Rep\ ops} \to \Sigma\, \mathit{ops} : \mathit{Sig\ Rep}\,.\, \mathit{Spec\ Rep\ ops}
\end{array}
$$

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 defintion by our running example.

Example 2.12 (Class of points) The type *PointClass* of classes of points with representation type *nat*, signature *SigPoint*, and specification *SpecPoint* (cf. Example 2.6) is constructed by means of the type constructor *Class*:

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

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

$$\begin{array}{lcl} \mathit{MyPointClass} & \stackrel{\text{def}}{=} & \lambda\, \mathit{Rep} : \star \\ & & \lambda\, \mathit{gp} : \mathit{Rep} \leq \mathit{nat}\,. \\ & & \lambda\, \mathit{self} : (\Sigma\, \mathit{ops} : \mathit{SigPoint\ Rep}\,.\, \mathit{SpecPoint\ Rep\ ops}). \\ & & \quad \langle \mathit{opsPointClass}, \mathit{prfsPointClass} \rangle \end{array}$$

The operations of the class are implemented as the following triple:[4]

$$\begin{array}{lcll} \mathit{opsPointClass} & = & (\ \lambda r : \mathit{Rep}\,.\,(\mathit{get\ gp})\ r, & (\mathit{getX}) \\ & & \ \lambda r : \mathit{Rep}\,.\,\lambda n : \mathit{nat}\,.\ (\mathit{put\ gp})\ r\ n, & (\mathit{setX}) \\ & & \ \lambda r : \mathit{Rep}\,.\ \mathit{self}\,.\,\mathit{ops}\,.\,\mathit{setX}\ \ r\ (\mathit{self}\,.\,\mathit{ops}\,.\,\mathit{getX}\ \ r) + 1\ ) & (\mathit{inc1}) \end{array}$$

Finally, we have to prove the correctness of the three operations just defined, i.e. give an element *prfsPointClass* of type *SpecPoint Rep opsPointClass*.

The first equation of the specification

$$\forall r : \mathit{Rep}\,.\,\forall n : \mathit{nat}\,.\ \ \mathit{opsPointClass}\,.\,\mathit{getX}(\mathit{opsPointClass}\,.\,\mathit{setX}\ \ r\ n) =_L n$$

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

$$\forall r : \mathit{Rep}\,.\,\forall n : \mathit{nat}\,.\,(\mathit{get\ gp})\ ((\mathit{put\ gp})\ r\ n) =_L n$$

The equation coincides with the first law for get and put, accessible by $(\mathit{gpOK\ gp}).1$.

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

$$\forall r : \mathit{Rep}\,.\ \ \mathit{opsPointClass}\,.\,\mathit{getX}(\mathit{opsPointClass}\,.\,\mathit{inc1}\ \ r) =_L (\mathit{opsPointClass}\,.\,\mathit{getX}\ \ r) + 1$$

which β-reduces to

$$\forall r : \mathit{Rep}\,.\,(\mathit{get\ gp})(\mathit{self}\,.\,\mathit{ops}\,.\,\mathit{setX}\ \ r\ (\mathit{self}\,.\,\mathit{ops}\,.\,\mathit{getX}\ \ r) + 1) =_L ((\mathit{get\ gp})\ r) + 1$$

[4]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*. Conversely the simple example was taken to illustrate the two crucial manipulations of state: reading and updating.

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.

2.3.4 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 [PT92], this dependency was resolved by using a fixed point operator. In strongly normalising calculi such as the calculus of construction, of course, the general fixed point operator of type $\forall A : Type\,.\,(A \to A) \to A$ is not definable and assuming a term *fix* of this type makes the system inconsistent.[5]

Approximating the fixed point operator *fix* by a *sequence* $fix_0, fix_1, \ldots$ of fixed point operators, where $fix_i = f\ \ fix_{i+1}$, as proposed by Martin-Löf in [ML90], does preserve the consistency of the CC, but destroys strong normalisation and thus makes equality undecidable. Fixed points do their damage by permitting unlimited iterations of functions. But we are not interested in classes whose instantiation requires unlimited iterations, because we do not regard self-methods a means to introduce general recursion to the programming language. We therefore restrict ourselves to those classes for which bounded iterations are enough to resolve the self-methods. Thus, consistency and normalisation are preserved at the price of having to give a number n of function iterations and an iteration basis *basis* for every resolution of self-methods. To this end, the iteration operator $nat\_iter : \forall A \,|\, Type\,.\, A \to (A \to A) \to nat \to A$ is used as defined in the Lego-library.[6]

Definition 2.13 (Instantiation) Assuming implicitly a representation type *ClassR*, a

[5] The term *fix false* $(\lambda f : false.\ f)$ proves the absurd proposition *false*.

[6] In [Aud93], an extension of the calculus of constructions with fixed points for the universe of programs is proposed.

signature *Sig*, and a specification *Spec*, the instantiation operator *new* is defined as:

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

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 the function $class\ \ ClassR\ (refl_{\leq}\ \ ClassR)$. A definition assuring the fixed point property of the iteration will be given in Section 3. Generating an appropriate number of function-iterations could be automated by a partially decidable algorithm.

Example 2.14 (Instance of points) An 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.

$$
\begin{array}{rcl}
basis & : & \Sigma\ ops : SigPoint\ nat\ .\ SpecPoint\ nat\ ops\\
MyPointInstance & \stackrel{\text{def}}{=} & new\ \ MyPointClass\ \ 3\ \ basis\ \ 2
\end{array}
$$

In this example, the iteration basis *basis* is merely assumed. This is dangerous in general, as there is no guarantee that the specification is satisfiable at all. To ensure consistency, an inhabitant of $\Sigma\ ops : SigPoint\ nat\ .\ SpecPoint\ nat\ ops$ is needed as basis for the iteration instead of just assuming it. This would amount to a whole implementation including the correctness proofs, but without the use of *self*. A partially decidable algorithm could be used to provide such a basis by iterating the implementation given by the programmer until the variable *self* is resolved completely.

2.3.5 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 must not refer to the operations and proofs of the superclass, but to the ones of the present class instead. Therefore, *self* of the superclass cannot be resolved by a fixed point operator, but the *self* of the subclass is supplied to the superclass — after an appropriate transformation with *coercion*.

Definition 2.15 (Inheritance) Assume implicitly a representation type $SuperR : \star$, a signature $SuperSig : \star \to \star$, and a specification $SuperSpec : \forall\, Rep : \star .\ (SuperSig\ Rep) \to \star$ of the superclass. In addition, for the subclass a representation type $SubR$, a signature $SubSig$, and a specification $SubSpec$ correspondingly. Finally, assume a proof $gp_{SubR \leq SuperR} : SubR \leq SuperR$ and a function $coercion : \forall\, Rep : \star .\ (\Sigma\, ops : SubSig\ Rep\ .\ SubSpec\ Rep\ ops) \to \Sigma\, ops : SuperSig\ Rep\ .\ SuperSpec\ Rep\ ops$. The *inheritance operator* is defined as follows:

$$
\begin{array}{lll}
inherit & \stackrel{\text{def}}{=} & \lambda\, SuperClass : Class\ SuperR\ SuperSig\ SuperSpec\ . \\
& & \lambda\, build : \ \forall\, Rep : \star .\ (Rep \leq SubR) \to \qquad\qquad\qquad\qquad (gp_{Rep \leq SubR}) \\
& & \qquad\quad (\Sigma\, ops : SuperSig\ Rep\ .\ SuperSpec\ Rep\ ops) \to \quad (super) \\
& & \qquad\quad (\Sigma\, ops : SubSig\ Rep\ .\ SubSpec\ Rep\ ops) \to \qquad (self) \\
& & \qquad\quad\ \Sigma\, ops : SubSig\ Rep\ .\ SubSpec\ Rep\ ops\ . \\
& & (\ \lambda\, Rep : \star . \\
& & \quad \lambda\, gp_{Rep \leq SubR} : Rep \leq SubR\ . \\
& & \quad \lambda\, self : \Sigma\, ops : SubSig\ Rep\ .\ SubSpec\ Rep\ ops\ . \\
& & \quad build\ \ Rep \\
& & \qquad\quad gp_{Rep \leq SubR} \\
& & \qquad\quad (SuperClass\ \ Rep \\
& & \qquad\qquad\qquad (trans_{\leq}\ gp_{Rep \leq SubR}\ \ gp_{SubR \leq SuperR}) \\
& & \qquad\qquad\qquad (coercion\ Rep\ self)) \\
& & \qquad\quad self) \\
& : & (Class\ SuperR\ SuperSig\ SuperSpec) \qquad\qquad \to \\
& & \ (\forall\, Rep : \star .\ (Rep \leq SubR) \to \\
& & \ \ (\Sigma\, ops : SuperSig\ Rep\ .\ SuperSpec\ Rep\ ops) \to \\
& & \ \ (\Sigma\, ops : SubSig\ Rep\ .\ SubSpec\ Rep\ ops) \to \\
& & \ \ \ \Sigma\, ops : SubSig\ Rep\ .\ SubSpec\ Rep\ ops\ ) \qquad \to \\
& & (Class\ SubR\ SubSig\ SubSpec)
\end{array}
$$

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 2.16 (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*:

$$SigCPoint \stackrel{\text{def}}{=} \lambda\, Rep : \star\, .\ (SigPoint\ \ Rep) \times (\underbrace{(Rep \to Rep)}_{inc2} \times \underbrace{(Rep \to Color)}_{getC})$$

For the specification *SpecCPoint*, assume an arbitrary representation type *Rep* and operations $ops : SigCPoint\, Rep$. Let *opspoint* stand for $ops\,.1$ and *opsnew* for $ops\,.2$. The specification extends the specification *SpecPoint* of points by three equations.

$$\begin{array}{rcl} SpecCPoint & \stackrel{\text{def}}{=} & (SpecPoint\ \ Rep\ \ opspoint) \times \\ & & (\forall r : Rep\, .\ \ opspoint\, .\, getX\ \ (opsnew\, .\, inc2\ \ r) =_L \\ & & \qquad\qquad (opspoint\, .\, getX\ \ r) + 2 \\ & & \ \forall r : Rep\, .\ \ opsnew\, .\, getC\ \ (opsnew\, .\, inc2\ \ r) =_L blue \\ & & \ \forall r : Rep\, .\, \forall n : nat\, .\ \ opsnew\, .\, getC\ \ (opspoint\, .\, setX\ \ r\ n) =_L blue) \end{array}$$

The functions *inc2* and *getC* thereby denote the projection functions of pairs. 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 projection functions, when the meaning is clear from the context. The same convention shall apply to the proofs. Now, we define a class *MyCPointClass* with representation type $(nat \times Color)$ by means of the inheritance operator *inherit*.

$$\begin{array}{rcl} MyCPointClass & \stackrel{\text{def}}{=} & inherit\, (nat \times Color)\ SigCPoint\ SpecCPoint \\ & & \qquad gp \\ & & \qquad coercion \\ & & \qquad MyPointClass \\ & & \qquad (\lambda\, Rep : \star\, . \\ & & \qquad\ \lambda\, gp_{Rep \leq (nat\, \times\, Color)} : Rep \leq (nat \times Color) \\ & & \qquad\ \lambda\, super : (\Sigma\, ops : SigPoint\ Rep\, .\ SpecPoint\ Rep\ ops) \\ & & \qquad\ \lambda\, self : (\Sigma\, ops : SigCPoint\ Rep\, .\ SpecCPoint\ Rep\ ops) \\ & & \qquad\ \langle opsCPointClass, prfsCPointClass \rangle) \\ & : & Class\, (nat \times Color)\ SigCPoint\ SpecCPoint \end{array}$$

The term $gp : (nat \times Color) \leq nat$ is a dependent triple consisting of the get and put functions $\lambda nc : (nat \times Color)\, .\ nc.1$ and $\lambda nc : (nat \times Color)\, .\ \lambda n : nat\, .\ (n, nc.2)$ together with the verification of the required laws, which is straightforward using the η-rule for pairs.[7] The coercion function *coercion* simply forgets 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

[7]The η-rule is provided by the inductive definition of the pair from the Lego-library.

variables *self* and *super* allow references to the methods of the colored point class and the point class respectively.

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

Finally, we have to prove the correctness of the five operations just defined, 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. Example 2.12) and will be addressed in the next section.

The second equation[8] on page 19 reduces to $super \, . \, ops \, . \, getX(super \, . \, ops \, . \, inc1 \;\; r) =_L (super \, . \, ops \, . \, getX \;\; r) + 1$, which coincides with the type of $super \, . \, prfs \, .2$. Hence, the inherited proof $super \, . \, prfs \, .2$ shows 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 super class, 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.

This equation β-reduces to $super \, . \, ops \, . \, getX(super \, . \, ops \, . \, inc1(super \, . \, ops \, . \, inc1 \;\; r)) =_L (super \, . \, ops \, . \, getX \;\; r) + 2$ 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.

[8] In the sequel, we elide leading universal quantification over $r : Rep$ and $n : nat$.

3.1 Classes

As seen in the previous section, we can cope with equations about non late binding methods. We have also mentioned in Example 2.12 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 Example 2.12 we cannot expect to prove the second equation in the specification of points, relating the implementations of *getX* and *inc1*:

$$(get\ gp)(self\,.\,ops\,.\,setX\ \ r\ (self\,.\,ops\,.\,getX\ \ r)+1)\ \ =_L\ \ ((get\ gp)\ r)+1 \tag{1}$$

since we do not know how the *setX* method in subclasses will behave together with the current implementation *get gp* 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.

At first sight, a solution could be to include details of the implementation into the specification.[9] In the example, one could think of adding the equation $ops\,.\,getX =_L get\ gp$ to the specification of points. This would give the desired connection between *self* and the present implementation: $self\,.\,getX =_L get\ gp$. Such a specification of internal details is clearly unwanted 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 becomes provable. The operations $self\,.\,getX$ and $self\,.\,setX$ then get replaced by their implementation, yielding:

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

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, then of course by the implementation of the respective subclass. The second equation then takes the form:

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

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.

[9] An extension to the encoding presented so far to specify such details has been presented in [Nar95] using ideas of [HP96].

This suggests providing a uniform proof inside the class of points, one not relying on the concrete implementation of *getX*, but only on the satisfaction of the first equation of *SpecPoint*. Therefore we replace the implementation of *getX* in Equation (1) by the abstract operation *self . ops . getX*:

$$\mathit{self.ops.getX}(\mathit{self.ops.setX}\ \ r\ (\mathit{self.ops.getX}\ \ r) + 1) =_L (\mathit{self.ops.getX}\ r) + 1 \qquad (1')$$

This modification is no real change *for the instances* of the class, since after substituting the *resolved* self operations for *self . ops*, the new specification coincides with the original one. To do so, the instantiation of the operations must be performed *before* the instantiation of the proofs. Hence the single fixed point must be splitted into two: one for the methods and one for the proofs, the second one depending on the first.

Since a generalised specification Spec' will contain the abstract self-versions of some methods together with the unchanged methods, it has to take both of them as parameters, and thus has type $\forall\, \mathit{Rep} : \star.\ (\mathit{Sig\,Rep}) \to (\mathit{Sig\,Rep}) \to \star$. As explained, it should be no stronger than the original specification *Spec* in the sense that, if the fixed point of the operations has been resolved, both specifications must coincide, i.e. for all representation types *Rep* and all operations *ops* we require $\mathit{Spec}'\ \mathit{Rep\ ops\ \ ops} =_L \mathit{Spec\ Rep\ ops}$ and hence can take this equation as definition for the ungeneralised specification *Spec*.

Definition 3.1 (Type of classes) Assume a representation type $\mathit{ClassR} : \star$, a signature $\mathit{Sig} : \star \to \star$, a generalised specification $\mathit{Spec}' : \forall\, \mathit{Rep} : \star.\ (\mathit{Sig\ Rep}) \to (\mathit{Sig\ Rep}) \to \star$. Let the term *Spec* stand for $\lambda\, \mathit{Rep} : \star.\ \lambda\, \mathit{ops} : (\mathit{Sig\ Rep}).\ \ \mathit{Spec}'\ \mathit{Rep\ ops\ ops}$, which denotes the ungeneralised version of the specification Spec'. The *type of classes* is then given as:

$$\begin{aligned} \mathit{Class} \ \stackrel{\text{def}}{=}\ & \forall\, \mathit{Rep} : \star.\ (\mathit{Rep} \leq \mathit{ClassR}) \to \\ & \Sigma f : (\mathit{Sig\ Rep}) \to \mathit{Sig\ Rep}\,. \\ & \qquad \forall\, \mathit{selfops} : \mathit{Sig\ Rep}\,.\ (\mathit{Spec\ Rep\ selfops}) \to \mathit{Spec}'\ \mathit{Rep}\ (f\ \mathit{selfops})\ \ \mathit{selfops} \end{aligned}$$

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. The type of points and their signature remain unchanged. The original specification *SpecPoint* is slightly generalised to $\mathit{SpecPoint}'$, which we shall define now.

Example 3.2 (Class of points) Assuming a representation type *Rep*, concrete operations *ops*, and abstract operations *selfops* of type *SigPoint Rep*, the *generalised specification* of points is given as $\mathit{SpecPoint}' \stackrel{\text{def}}{=}$

$$\begin{array}{lrcl} \forall r : \mathit{Rep}\,. \forall\, n : \mathit{nat}\,. & \mathit{ops}\,.\,\mathit{getX}(\mathit{ops}\,.\,\mathit{setX}\ \ r\ \ n) & =_L & n \\ \forall r : \mathit{Rep}\,. & \mathit{selfops}\,.\,\mathit{getX}(\mathit{ops}\,.\,\mathit{inc1}\ \ r) & =_L & (\mathit{selfops}\,.\,\mathit{getX}\ \ r) + 1 \end{array}$$

The specification $\mathit{SpecPoint} \stackrel{\text{def}}{=} \lambda\, \mathit{Rep} : \star.\ \lambda\, \mathit{ops} : (\mathit{SigPoint\ Rep}).\ \ \mathit{SpecPoint}'\ \mathit{Rep\ ops\ ops}$ is defined by identifying *ops* and *selfops* of the specification $\mathit{SpecPoint}'$.

The type *PointClass* with representation type *nat*, signature *SigPoint* and generalised specification $\mathit{SpecPoint}'$, is constructed by means of the type constructor *Class*:

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

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

$$MyPointClass \stackrel{\text{def}}{=} \lambda\, Rep : \star .\ \lambda\, gp : Rep \leq nat\ .\ \langle opsPointClass, prfsPointClass \rangle$$

As we saw, 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:

$$\begin{aligned} opsPointClass \ = \ & \lambda\, selfops : SigPoint\ Rep\ . \\ & \quad (\lambda r : Rep\ .\ (get\ gp)\ r, && (getX) \\ & \quad\ \lambda r : Rep\ .\ \lambda n : nat\ .\ \ (put\ gp)\ r\ n, && (setX) \\ & \quad\ \lambda r : Rep\ .\ \ selfops\ .\ setX\ \ r\ (selfops\ .\ getX\ \ r) + 1) && (inc1) \end{aligned}$$

Finally, we have to prove the correctness of the three operations just defined, i.e. assuming $selfops : SigPoint\ Rep$ and $selfprfs : SpecPoint\ Rep\ selfops$, give an element *prfsPointClass* of type $SpecPoint'\ Rep\ (opsPointClass\ selfops)\ \ selfops$. The proof of the 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 Equation (1′); the self-proof $selfprfs\ .1 : \forall r : Rep\ .\ \forall n : nat\ .\ selfops\ .\ getX\ \ (selfops\ .\ setX\ r\ n) =_L n$ is used to prove it.

3.2 Instantiation

Next we adapt the instantiation function to deal with the modified definition of classes. The main change is that, first, *new* computes the fixed point *operations* of the function $f : (Sig\ Rep) \rightarrow Sig\ Rep$, which implements the operations abstracted over *self*, and afterwards resolves the correctness proofs of *operations*. Formally though, it is not possible to solve the proof part of classes directly by iteration, since we cannot iterate a function of type $(Spec\ Rep\ operations) \rightarrow Spec'\ Rep\ \ (f\ operations)\ \ operations$. But by the definition of *Spec*, and the fact that *operations* is a fixed point of f we know that the equation $Spec'\ Rep\ (f\ \ operations)\ \ operations =_L Spec\ Rep\ operations$ holds.

Definition 3.3 (Instantiation) Assuming implicitly a representation type *ClassR*, a signature *Sig*, and a generalised specification *Spec′*. Let the term *Spec* stand for the ungeneralised specification $\lambda\, Rep : \star .\ \ \lambda\, ops : (Sig\ Rep).\ \ Spec'\ Rep\ ops\ ops\ .$ The *instantiation operator* is then defined as:

$$
\begin{array}{lll}
\mathit{new} & \stackrel{\text{def}}{=} & \lambda\, \mathit{class} : \mathit{Class\ ClassR\ Sig\ Spec\ Spec'\ Spec'\_ok}\ .\\
& & \lambda\, \mathit{state} : \mathit{ClassR}\ .\\
& & \lambda\, \mathit{opsbasis} : \mathit{Sig\ ClassR}\ .\\
& & \lambda\, n : \mathit{nat}\ .\\
& & \mathit{let}\quad \mathit{opsprfs} \;=\; \mathit{class\ ClassR}\ (\mathit{refl}_{\leq}\ \mathit{ClassR})\\
& & \qquad \mathit{operations} \;=\; \mathit{nat\_iter\ opsbasis\ opsprfs}\,.\,\mathit{ops}\ n\\
& & \mathit{in}\ \ \lambda\, \mathit{selfopsresolved} :\ \mathit{opsprfs}\,.\,\mathit{ops\ operations} =_L \mathit{operations}\ .\\
& & \quad \lambda\, \mathit{prfsbasis} : \mathit{Spec\ ClassR\ operations}\ .\\
& & \quad \lambda\, m : \mathit{nat}\ .\\
& & \quad \mathit{let\ proofs} = \mathit{nat\_iter\ prfsbasis}\ (h\ (\mathit{opsprfs}\,.\,\mathit{prfs\ operations}))\ m\\
& & \quad \mathit{in}\ \ \lambda\, \mathit{selfprfsresolved} : (h\ (\mathit{opsprfs}\,.\,\mathit{prfs\ \ operations}))\ \mathit{proofs} =_L \mathit{proofs}\ .\\
& & \qquad \mathit{ObjectIntro\ ClassR\ operations\ proofs}
\end{array}
$$

The function h of type $((\mathit{Spec\ ClassR\ operations}) \rightarrow (\mathit{Spec'\ ClassR}\ (\mathit{opsprfs}\,.\,\mathit{ops\ operations})\ \mathit{operations})) \rightarrow ((\mathit{Spec\ ClassR\ operations}) \rightarrow (\mathit{Spec\ ClassR\ operations}))$ performs the required adaption of types using *selfopsresolved* and substitutability of equality.

Example 3.4 (Instance of points) The instantiation for points remains basically unchanged. Again, only two iterations are needed to resolve the self-methods. For the proof part, two iterations suffice as well, but, of course, in general the numbers of iterations may be different. After having reached the fixed point, the proofs of stability are 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 with the operations and the proofs separately. We also solve the problem of mixing inherited operations with newly implemented ones, encountered in Example 2.16. 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 super class, 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 super class.

Definition 3.5 (Inheritance) Assume implicitly a representation type *SuperR*, and a signature *SuperSig*, as in Definition 2.15. In addition assume implicitly a generalised specification $\mathit{SuperSpec'} : \forall\, \mathit{Rep} : \star\,.\ (\mathit{SuperSig\ Rep}) \rightarrow (\mathit{SuperSig\ Rep}) \rightarrow \star$. Furthermore assume a representation type *SubR*, a signature *SubSig*, and a generalised specification $\mathit{SubSpec'}$ for the subclass. Let the terms *SuperSpec* and *SubSpec* stand for the ungeneralised specifications as in Definition 3.1. Finally assume proofs $\mathit{gp}_{\mathit{SubR} \leq \mathit{SuperR}} : \mathit{SubR} \leq \mathit{SuperR}$ and two coercion functions $\mathit{co\_sig} : \forall\, \mathit{Rep} : \star\,.\ (\mathit{SubSig\ Rep}) \rightarrow \mathit{SuperSig\ Rep}$ and *co_spec* of type $\forall\, Rep : \star .\ \forall\, selfops : SubSig\ Rep .\ (SubSpec\ Rep\ selfops) \rightarrow SuperSpec\ \ Rep$ $(co\_sig\ Rep\ selfops)$. The *inheritance operator* is defined as follows:

$$
\begin{array}{lll}
inherit & \stackrel{\mathrm{def}}{=} & \lambda\, SuperClass : Class\ SuperR\ SuperSig\ SuperSpec' . \\
& & \lambda\, build : \ \forall\, Rep : \star .\ \forall\, gp_{Rep \leq SubR} : Rep \leq SubR . \\
& & \qquad \forall\, superops : SuperSig\ Rep . \\
& & \qquad \Sigma\ \ f : (SubSig\ Rep) \rightarrow SubSig\ Rep, \\
& & \qquad\quad let\ \ gp_{Rep \leq SuperR} \ = \ trans_{\leq}\ \ gp_{Rep \leq SubR}\ \ gp_{SubR \leq SuperR} \\
& & \qquad\qquad\qquad super \ = \ SuperClass\ Rep\ gp_{Rep \leq SuperR} \\
& & \qquad\quad in\ \ \forall\, selfops : SubSig\ Rep . \\
& & \qquad\qquad (superops =_L super\,.\,ops\ (co\_sig\ Rep\ selfops)) \rightarrow \\
& & \qquad\qquad (SuperSpec'\ Rep\ superops\ (co\_sig\ Rep\ selfops)) \rightarrow \\
& & \qquad\qquad (SubSpec\ Rep\ selfops) \rightarrow SubSpec'\ Rep\ (f\ selfops)\ \ selfops \\
& & (\ \lambda\, Rep : \star .\ \lambda\, gp_{Rep \leq SubR} : Rep \leq SubR . \\
& & \quad let\ \ gp_{Rep \leq SuperR} \ = \ trans_{\leq}\ \ gp_{Rep \leq SubR}\ \ gp_{SubR \leq SuperR} \\
& & \qquad\quad super \ = \ SuperClass\ Rep\ gp_{Rep \leq SuperR} \\
& & \qquad\quad opsprfs \ = \ \lambda\, selfops : SubSig\ Rep . \\
& & \qquad\qquad\qquad build\ \ Rep\ gp_{Rep \leq SubR} \\
& & \qquad\qquad\qquad\quad (super\,.\,ops\ (co\_sig\ Rep\ selfops)) \\
& & \quad in\ \ \langle \lambda\, selfops : SubSig\ Rep\ .\ (opsprfs\ selfops).\,ops\ selfops, \\
& & \qquad \lambda\, selfops : SubSig\ Rep\ . \\
& & \qquad \lambda\, selfprfs : SubSpec\ Rep\ selfops\ . \\
& & \qquad (opsprfs\ selfops).\,prfs\ \ selfops \\
& & \qquad\qquad (refl_{=_L}\ \ super\,.\,ops\ (co\_sig\ Rep\ selfops)) \\
& & \qquad\qquad (super\,.\,prfs\ \ (co\_sig\ Rep\ selfops) \\
& & \qquad\qquad\qquad (co\_spec\ Rep\ selfops\ selfprfs)) \\
& & \qquad\qquad selfprfs \rangle \\
& & \ ) : Class\ SubR\ SubSig\ SubSpec'
\end{array}
$$

Continuing with the example of colored points, we don not need to change the definitions of *CPoint*, *SigCPoint*, and *SpecPoint* of Section 2.6. To complete the example, we define a generalised specification *SpecCPoint'*.

Example 3.6 (Colored points) For the generalised specification *SpecCPoint'*, assume an arbitrary representation type *Rep*, concrete operations *ops*, and abstract operations *selfops* of type *SigCPoint Rep*.

$$
\begin{array}{lll}
SpecCPoint' & \stackrel{\mathrm{def}}{=} & (SpecPoint'\ \ Rep\ \ ops\ selfops) \times \\
& & (\forall r : Rep .\ \ selfops\,.\,getX\ (ops\,.\,inc2\ \ r) =_L (selfops\,.\,getX\ \ r) + 2 \\
& & \ \forall r : Rep .\ \ selfops\,.\,getC\ (ops\,.\,inc2\ \ r) =_L blue \\
& & \ \forall r : Rep\,.\,\forall n : nat .\ \ ops\,.\,getC\ (ops\,.\,setX\ \ r\ n) =_L blue)
\end{array}
$$

Now we 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 Example 2.16. The two terms *co_sig* and *co_spec* denote the natural coercion functions from colored points to points.

$$
\begin{array}{lcl}
\mathit{MyCPointClass} & \stackrel{\text{def}}{=} & \mathit{inherit}\ \ (\mathit{nat} \times \mathit{Color})\ \mathit{SigCPoint}\ \mathit{SpecCPoint}' \\
 & & \qquad\qquad \mathit{gp}\ \mathit{co\_sig}\ \mathit{co\_spec} \\
 & & \qquad\qquad \mathit{PointClass} \\
 & & \qquad\qquad (\ \lambda\, \mathit{Rep} : \star\, . \\
 & & \qquad\qquad\quad \lambda\, \mathit{gp}_{\mathit{Rep} \leq (\mathit{nat} \times \mathit{Color})} : \mathit{Rep} \leq (\mathit{nat} \times \mathit{Color}) \\
 & & \qquad\qquad\quad \lambda\, \mathit{superops} :\ \mathit{SigPoint}\ \mathit{Rep} \\
 & & \qquad\qquad\quad \langle \mathit{opsCPointClass}, \mathit{prfsCPointClass} \rangle) \\
 & : & \mathit{Class}\, (\mathit{nat} \times \mathit{Color})\ \mathit{SigCPoint}\ \mathit{SpecCPoint}'
\end{array}
$$

The implementation of the operations is given by the quintuple *opsCPointClass* abstracted over the self operations.

$$
\begin{array}{lcll}
\mathit{opsCPointClass} & = & \lambda\, \mathit{selfops} : \mathit{SigCPoint}\ \mathit{Rep}\, . & \\
 & & (\ \lambda r : \mathit{Rep}\, .\ \ \mathit{superops}\, .\, \mathit{getX}\ \ r, & (\mathit{getX}) \\
 & & \quad \lambda r : \mathit{Rep}\, .\ \lambda n : \mathit{nat}\, . & \\
 & & \qquad (\mathit{put}\ \mathit{gp}_{\mathit{Rep} \leq (\mathit{nat} \times \mathit{Color})})\ r\ (n, \mathit{blue}), & (\mathit{setX}) \\
 & & \quad \lambda r : \mathit{Rep}\, .\ \ \mathit{superops}\, .\, \mathit{inc1}\ \ r, & (\mathit{inc1}) \\
 & & \quad \lambda r : \mathit{Rep}\, .\ \ \mathit{superops}\, .\, \mathit{inc1}\ \ (\mathit{selfops}\, .\, \mathit{inc1}\ \ r), & (\mathit{inc2}) \\
 & & \quad \lambda r : \mathit{Rep}\, .\ ((\mathit{get}\ \mathit{gp}_{\mathit{Rep} \leq (\mathit{nat} \times \mathit{Color})})\ r).2\ ) & (\mathit{getC})
\end{array}
$$

The reader may have noticed that, compared with the corresponding definition on page 20, we have slightly complicated the implementation of the *inc2* method. In the verification we shall see how the new encoding of classes and objects can also deal with a mixture of *self* and *super*, as employed here in the implementation of *inc2*.

With the new encoding all equations of the colored point class are provable. Assuming $\mathit{selfops} : \mathit{SigCPoint}\ \mathit{Rep}$, a proof $\mathit{superops\_ok} :(\mathit{superops} =_L \mathit{super}\, .\, \mathit{ops}(\mathit{co\_sig}\ \mathit{Rep}\ \mathit{selfops}))$, the reference to the proofs of the super class of points $\mathit{superprfs} : \mathit{SpecPoint}'\ \mathit{Rep}\ \mathit{superops}$ $(\mathit{co\_sig}\ \mathit{Rep}\ \mathit{selfops})$, and the self proofs $\mathit{selfprfs} :(\mathit{SpecCPoint}\ \mathit{Rep}\ \mathit{selfops})$, we have to prove the specification $\mathit{SpecCPoint}'\ \mathit{Rep}\ (\mathit{opsCPoint}\ \mathit{selfops})\ \ \mathit{selfops}$. In the following, we abbreviate the implementation *opsCPoint selfops* 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 $\mathit{Cops}\, .\, \mathit{getX}(\mathit{Cops}\, .\, \mathit{setX}\ r\ n) =_L\ \mathit{Cops}\, .\, \mathit{getX}\ r$ of the colored points. This equation β-reduces to:

$$\mathit{superops}\, .\, \mathit{getX}((\mathit{put}\ \mathit{gp}_{\mathit{Rep} \leq (\mathit{nat} \times \mathit{Color})})\ \ r\ (n, \mathit{blue})) =_L n$$

Using *superops_ok*, the variable *superops* can be replaced by the implementation of the point class, yielding:

$$((\mathit{get}\ \mathit{gp}_{\mathit{Rep} \leq (\mathit{nat} \times \mathit{Color})})\ ((\mathit{put}\ \mathit{gp}_{\mathit{Rep} \leq (\mathit{nat} \times \mathit{Color})})\ \ r\ (n, \mathit{blue}))).1 =_L n$$

This is immediate by the laws for get and put.

The new encoding with the generalised 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 Example 2.16.

The proof of the third equation $\mathit{selfops}.\mathit{getX}(\mathit{Cops}.\mathit{inc2}\ r) =_L (\mathit{selfops}.\mathit{getX}\ r) + 2$ shows, that it is also possible to prove equations, where methods, referenced by *super* are mixed with self methods. The equation β-reduces to

$$\mathit{selfops}.\mathit{getX}(\mathit{superops}.\mathit{inc1}(\mathit{selfops}.\mathit{inc1}\ r)) =_L (\mathit{selfops}.\mathit{getX}\ r) + 2$$

The knowledge about the implementation of the superclass of points (*superops_ok*) allows to infer

$$\mathit{selfops}.\mathit{getX}(\mathit{selfops}.\mathit{setX}\ r((\mathit{selfops}.\mathit{getX}(\mathit{selfops}.\mathit{inc1}\ r)) + 1))$$

for the left hand side of the equation. The first proof *selfprfs*.1 is used to replace this by $\mathit{selfops}.\mathit{getX}(\mathit{selfops}.\mathit{inc1}\ r) + 1$ and *selfprfs*.2 to equate it with $\mathit{selfops}.\mathit{getX}\ r + 2$ as desired.

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

$$\mathit{selfops}.\mathit{getC}(\mathit{superops}.\mathit{inc1}(\mathit{selfops}.\mathit{inc1}\ r) =_L \mathit{blue}$$

whose left hand side can further be developed to

$$\mathit{selfops}.\mathit{getC}\ (\mathit{selfops}.\mathit{setX}\ (\mathit{selfops}.\mathit{inc1}\ r)\ (\mathit{selfops}.\mathit{getX}\ (\mathit{selfops}.\mathit{inc1}\ r)) + 1)$$

with the help of *superops_ok*. Specialising the fifth proof *selfprfs*.5 to $r = \mathit{selfops}.\mathit{inc1}\ r$ and $n = \mathit{selfops}.\mathit{getX}\ (\mathit{selfops}.\mathit{inc1}\ r)) + 1$ shows the equality of this expression with *blue*.

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

4 Conclusion

Building upon the object-model of [PT94] and [HP96], this paper presented a formalization of the semantics of object-oriented features in sufficient detail to support program verification. By augmenting the interface of objects by a specification of its behaviour, we demonstrated how object-oriented structuring techniques can be usefully employed in organising the proofs as well. The complete formalization also enforces disciplined arrangementss to deal with the mass of detail. We see it as confirmation of the utility of computer-aided formalization in general, and Lego in particular; without computer support would not have been possible.

Comparison with other work

In Lego much work has been done in formalising mathematical theories and also in the field of program specification and verification [Luo92] [BM93] [Sch93] [Hof92] [Sch93] [Wan92] to mention several. While there is an increasing body of work about the semantic foundations of object-oriented programming, notably in the area of typed functional calculi (see [GM94]), there are still only a few investigations about verification of specific object-oriented programs.

Leavens and Wheil in a series of papers [Lea88, Lea90, Lea91, LW94, Lea93] 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 the 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 behaviour, 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 behaviour-preserving subtyping is known as *behavioural subtyping* [Ame89]. To obtain a convenient mathematical model of the abstractly specified objects, they restrict their attention to objects with immutable 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 behaviour 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 [DL92]. 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 Wheil, Utting [Utt92] [UR93] 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 [Bac78] [Mor87] [Mor90], being itself an extension of Dijkstra's guarded command language [Dij76], is presented, a wide spectrum language, where executable code and specifications can be freely mixed.

[Mai93] 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 [CM89], 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 [CCHM89] or subtyping, and allows to investigate metamathematical properties such as soundness, consistency, and coherence of different encoded idioms. In contrast to our work, encapsulation, inheritance, or late-binding are not treated. Like in this paper finite unwindings are employed to resolve the fixed points in the encodings of objects. To represent record types for objects, [Hic96] 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 formalised in the NuPRL proof development system [C$^+$86].

Further Work This paper addressed the the generalisation of a specific object model, namely $F^\omega_\leq$'s, to transfer object-oriented programming methodology to the process of verification. One direction of further work could be extending or changing 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 [CP93]. Another easy generalisation could include parametrised classes [PT94]. One could add syntactic sugar or fancier notions of specification, e.g. splitting the specification into an visible, external part, and an internal, hidden one, or to include matching [Bru92] as weaker notion of subtyping, which seem to have advantages in treating binary methods.

A pragmatic path might be, to care for greater ease of the verification process. This could include the automatic generation of *get* and *put* functions proposed in [HP96] for positive signatures or the automatic calculation of the number of fixpoint unwindings. Besides verifying properties of single programs, the transfer into Lego could also be used to prove properties about the encoding itself, such as properties of the inheritance or instantiation operator and the like.

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 2.3.2, it is also problematic to place the test of equality on objects as an equality method inside the objects. In the chosen model, the 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. For the problem of equality with regard to abstract data types, in [Luo92] each representation of an abstract data type comes equipped with an equality, which allows at least to compare data types with the same internal representation. With their interior hidden, it is natural to regard objects as equal if they are *observationally* indistinguishable. Objects can be seen as co-algebras [HP92] resulting in proof methods being a coinduction or bisimulation.

Whether finally the proposed object-oriented structuring mechanisms for verification scale up remains to be investigated in more realistic examples and case studies. A step in this direction is the verification of a small hierarchy of classes resembling Smalltalk-collections in [Nar96].

Acknowledgements

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. Thanks also to the anonymous referees for their helpful comments.

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”.

References

[AG91] Antonio Alencar and Joseph Goguen. OOZE: An object-oriented Z environment. In P. America, editor, *ECOOP '91*, volume 512 of *Lecture Notes in Computer Science*, pages 180–199. Springer, 1991.

[AGNvS94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. *A User's Guide to ALF.* Chalmers University of Technology, Sweden, May 1994.

[Ame89] Pierre America. A behavioural approach to subtyping in object-oriented programming languages. 443, Phillips Research Laboratories, January/April 1989.

[Aud93] 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.

[Bac78] Ralph-Johan R. Back. *On the Correctness of Refinement in Program Development.* PhD thesis, Department of computer Science, University of Helsinki, 1978.

[Bar92] 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.

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[BCC$^+$95] K. Bruce, L. Cardelli, G. Castagna, the Hopkins Objects Group (J. Eifrig, S. Smith, V. Trifonov), G. Leavens, and B. Pierce. On binary methods. Technical report, May 1995. Submitted for publication. Available electronically and as a technical report from LIENS, DEC SRC, and Iowa State.

[BM93] 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.

[Bru92] Kim Bruce. A paradigmatic object-oriented language: Design, static typing and semantics. Technical Report CS-92-01, Williams College, January 1992.

[C$^+$86] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice Hall, 1986.

[CCHM89] 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*, LNCS, pages 273–280. ACM, Springer, September 1989.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic.* North-Holland, 1958.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[CM89] Luca Cardelli and John Mitchell. Operation on records (extended abstract). In Pitt et al. [PRD+89], pages 75–81.

[CO88] S. Clerici and Fernando Orejas. GSBL: an algebraic specification language based on inheritance. In S. Gjessing and K. Nygaard, editors, *ECOOP '88*, volume 322 of *Lecture Notes in Computer Science*, pages 78–92. Springer, 1988.

[Coq86] Thierry Coquand. An analysis of Girard's paradox. In *First Annual Symposium on Logic in Computer Science (LICS) (Cambridge, MA)*, pages 227–236. IEEE, Computer Society Press, June 1986.

[CP93] Adriana B. Compagnoni and Benjamin Pierce. Multiple inheritance via intersection types. Technical Report ECS-LFCS-93-275, Laboratory for Foundations of Computer Science, University of Edinburgh, 1993. also published as Catholoc University Nijmegen computer science technical report 93-18.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. [SH80], pages 589–606.

[DDRS89] David Duke, Roger Duke, Gordon Rose, and Graeme Smith. Object-Z: An object-oriented extension to Z. In Kenneth J. Turner, editor, *First International Conference on Formal Description Techniques FORTE '88 (Stirling, Scotland)*, pages xxx–yyy. North-Holland, 1989.

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq Proof Assistant User's Guide. Rapports Techniques 154, INRIA Rocquencourt, Projet Formel, May 1993. Version 5.8.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[DL92] 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.

[FM94] 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.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupure dans l'arithmetique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[GM87] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. Technical Report SRI-CSL-87-7, SRI International, 1987.

[GM94] 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.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implemementation*. Addison-Wesley, 1983.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993. Preliminary version appeared in Proc. 2nd IEEE Symposium on Logic in Computer Science, 1987, 194–204.

[Hic96] Jason J. Hickey. Formal objects in type theory using very dependent types. Cornell University, Department of Computer Science, On the Net, August 1996. Extended abstract.

[Hof92] 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.

[How80] W. A. Howard. The formulae-as-types-notion of construction. In Seldin and Hindley [SH80], pages 479–490.

[HP92] Martin Hofmann and Benjamin Pierce. An abstract view of objects and subtyping. Technical Report ECS-LFCS-92-225, Laboratory for Foundations of Computer Science, University of Edinburgh, August 1992.

[HP96] Martin Hofmann and Benjamin Pierce. Positive subtyping. *Information and Computation*, 1996. To appear. Previous versions appeared in the Proceedings of Twenty-Second Annual ACM Symposium on Principles of Programming Languages, 1995, ACM, and as University of Edinburgh technical report ECS-LFCS-94-303, September 1994.

[Jac94] Paul Jackson. *NuPRL Manual Version 4.1*. Cornell University, April 1994.

[JM94] Claire Jones and Savi Maharaj. The LEGO library. Available on the World Wide Web [Lego95], February 1994.

[KS91] Bernd Krieg-Brückner and Donald Sannella. Structuring specifications in-the-large and in-the-small: Higher-order functions, dependent types and inheritance in SPECTRAL. Technical Report ECS-LFCS-91-135, Laboratory for Foundations of Computer Science, University of Edinburgh, January 1991.

[Lea88] Gary T. Leavens. *Verifying Object-Oriented Programs that use Subtypes*. PhD thesis, Massachusetts Institute of Technology, 1988.

[Lea90] Gary T. Leavens. Modular verification of object-oriented programs with subtypes. Technical Report 90-09, Iowa State University, Department of Computer Science, July 1990.

[Lea91] 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.

[Lea93] Gary T. Leavens. Inheritance of interface specifications. Technical Report TR 93-23, Iowa State University, Department of Computer Science, September 1993. (Extended Abstract).

[Lego95] Lego. The Lego proof assistant. `http://www.dcs.ed.ac.uk/packages/lego` on the World Wide Web, 1995.

[LP92] 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.

[Luo90] Zhaohui Luo. An extended Calculus of Constructions. Thesis ECS-LFCS-90-118, Laboratory for Foundations of Computer Science, University of Edinburgh, July 1990.

[Luo92] 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.

[LW94] 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.

[Mac86] 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.

[Mai93] 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.

[Mit90] 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.

[ML90] Per Martin-Löf. Mathematics of infinity. In P. Martin-Löf and G. Mints, editors, *COLOG-88. International Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 146–197. Springer, 1990.

[Mor87] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

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

[MP88] 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.

[Nar95] Wolfgang Naraschewski. Object-oriented proving. `http://www7.informatik.uni-erlangen.de /tree /IMMD-VII /Research /Projects /SFB-C2 /lego-summerschool/` on the World Wide Web, 14 - 18 August 1995.

[Nar96] Wolfgang Naraschewski. *Object-Oriented Proof Principles using the Proof-Assistant Lego.* Diplomarbeit, Universität Erlangen, 1996.

[Pie92] Benjamin Pierce. *F-Omega-Sub User's Manual*, version 1.0 edition, October 1992.

[PRD⁺89] David Pitt, David Rydeheard, Peter Dybjer, Andrew Pitts, and Axel Poigné, editors. *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*. Springer, September 1989.

[PT92] 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)*.

[PT94] 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".

[Rey74] 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.

[Sch93] Thomas Schreiber. *Verifikation von imperativen Programmen mit dem Beweisprüfer* LEGO. Diplomarbeit, Universität Erlangen, 1993.

[SH80] J. P. Seldin and J. R. Hindley, editors. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[ST86] Donald Sannella and Andrzej Tarlecki. Extended ML: an institution-independent framework for formal program development. Technical Report ECS-LFCS-86-16, Laboratory for Foundations of Computer Science, University of Edinburgh, December 1986.

[ST91] Donald Sannella and Andrzej Tarlecki. Extended ML: Past, present and future. Technical Report ECS-LFCS-91-138, Laboratory for Foundations of Computer Science, University of Edinburgh, February 1991.

[UR93] 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.

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

[Wan92] P. Wand. Functional programming and verification with Lego. Master's thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1992.

[Wir90] Martin Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13, pages 675–788. Elsevier, 1990.

[Wra89] Gavin C. Wraith. A note on categorical datatypes. In Pitt et al. [PRD+89], pages 118–127.