Object-Oriented Verification based on Record Subtyping in Higher-Order Logic

Wolfgang Naraschewski and Markus Wenzel

Technische Universität München
Institut für Informatik, Arcisstraße 21, 80290 München, Germany
http://www4.informatik.tu-muenchen.de/~narasche/
http://www4.informatik.tu-muenchen.de/~wenzelm/

8th June 1998

Abstract

We show how extensible records with structural subtyping can be represented directly in Higher-Order Logic (HOL). Exploiting some specific properties of HOL, this encoding turns out to be extremely simple. In particular, structural subtyping is subsumed by naive parametric polymorphism, while overridable generic functions may be based on overloading. Taking HOL plus extensible records as a starting point, we then set out to build an environment for object-oriented specification and verification (HOOL). This framework offers several well-known concepts like classes, objects, methods and late-binding. All of this is achieved by very simple means within HOL.

1 Introduction

Higher-order Logic (HOL) [2, 1, 3] is a rather simplistic typed system, Church originally even called it "Simple Theory of Types". At first sight, it might seem futile attempting to use HOL to represent extensible records with structural subtyping, or even object-oriented concepts. One might expect that this requires more advanced concepts at the level of types. The more surprising that HOL is perfectly capable of providing extensible record types. The encoding even turns out to be very simple and natural.

Extensible records in HOL give rise to applications in general mathematical modeling. We will hint at these by an example of simple abstract algebra.

Taking HOL plus extensible records as a basis we develop an object-oriented specification and verification environment (HOOL). This provides several well-known object-oriented concepts like classes, objects, methods and late-binding. On top of these basic concepts we could even achieve abstract classes and encapsulation (hiding), just by employing some mechanism of abstract theories like axiomatic type classes [14].

While this work has originated in the context of Isabelle/HOL [11], in principle its results carry over to other HOL implementations as well. Subsequently we will always refer to "HOL" in a generic sense.

A note on implementation: the latest official Isabelle release (Isabelle98) includes a prototypical package for extensible records. While demonstrating the basic ideas, it is not quite suited for real applications. You should get a more recent (probably unofficial) release for your own experiments.

This paper is structured as follows. Section 2 gives some impression on how to use extensible records in general mathematical modeling. We present a simple example of abstract algebra. Section 3 is foundational: after introducing the HOL logic to some extent, we present our particular encoding of extensible records. Section 4 introduces an environment for object-oriented specification and verification (HOOL). We demonstrate its main features by the running example of coloured points and rectangles. Section 5 explains how the HOOL concepts can be represented in HOL. Section 6 discusses object-oriented verification within the HOOL environment.

2 Basic use of extensible records

2.1 What are extensible records anyway?

2.1.1 Tuples and records

We briefly review some basic notions and notations.

Ordinary tuples and tuple types, which are taken for granted, are written as usual in mathematics, e. g. a triple (a, b, c) of type $A \times B \times C$.

Records are a minor generalization of tuples, where components may be addressed by arbitrary labels (strings, identifiers, etc.) instead of just position. Our concrete record syntax is borrowed from ML: e. g. $\{x = a, y = b, z = c\}$ denotes an individual record of labels x, y, z and values a, b, c, respectively. The corresponding record type would be of the form $\{x :: A, y :: B, z :: C\}$. Note that the labels contribute to record identity, consequently $\{x = 3, y = 5\}$ is completely different from $\{foo = 3, bar = 5\}$.

2.1.2 Record schemes

Unlike ordinary tuples, records are better suited to a property oriented view in the sense of "record r has field l". As a concise means to refer to classes of records featuring certain fields we introduce *schemes*, both on the level of records and record types. Patterns of the form $\{x = a, y = b, \ldots\}$ refer to any record having at least fields x, y of value a, b, respectively. The corresponding type scheme is written as $\{x :: A, y :: B, \ldots\}$. The dots "$\ldots$" are actually part of our notation and are pronounced "more". The more part of record schemes may be instantiated by zero or more further components. In particular, the concrete record $\{x = a, y = b\}$ is considered a (trivial) instance of the scheme $\{x = a, y = b, \ldots\}$.

As an example of relating records consider schemes $\{x = a, y = b, \ldots\}$ and $\{x = a, y = b, z = c, \ldots\}$. These are related in the sense that the latter is an extension of the former by addition of field $z = c$. On the level of types,

one might say that any $\{x :: A, y :: B, z :: C, \ldots\}$ is a *structural subtype* of $\{x :: A, y :: B, \ldots\}$. Note that (in our framework) record subtyping may only hold if the parent is an *extensible* record scheme. As a counterexample, instances of $\{x :: A, y :: B, z :: C, \ldots\}$ are *not* considered extensions of the *concrete* record type $\{x :: A, y :: B\}$.

With record schemes at the term and type level we have already "extensible records" at our disposal. In particular, we can define functions that operate on whole classes of records schematically, like $f\ \{x = a, y = b, \ldots\} \equiv t$. Here the l. h. s. is supposed to bind variables a, b and "..." by pattern matching. To improve readability, we occasionally abbreviate $\{x = x, y = y, \ldots\}$ by $\{x, y, \ldots\}$, even on the r. h. s. provided this does not cause any ambiguity.

Before discussing encodings of this general concept of extensible records in formal logical systems we demonstrate its use by an example.

2.2 Example: abstract algebraic structures

Consider some bits of group theory: A *monoid* is a structure with carrier α and operations $\circ :: \alpha \to \alpha \to \alpha$ and $1 :: \alpha$ such that $\circ$ is associative and 1 is a left and right unit element (w.r.t. $\circ$). A *group* is a monoid with additional operation $inv :: \alpha \to \alpha$ such that *inv* is left inverse (w.r.t. $\circ$ and 1). An *agroup* (abelian group) is a group where $\circ$ is commutative.

A well-known approach to abstract theories in HOL [3] uses n-ary predicates over the structures' operations (carrier types are included implicitly via polymorphism). Then *monoid* would be a predicate on pairs and *group*, *agroup* on triples as follows (below we use fancy syntax $\circ$, 1 for variables):

defs
$monoid :: (\alpha \to \alpha \to \alpha) \times \alpha \to bool$
$monoid\ (\circ, 1) \equiv \forall x\ y\ z.\ (x \circ y) \circ z = x \circ (y \circ z)\ \wedge\ 1 \circ x = x\ \wedge\ x \circ 1 = x$
$group :: (\alpha \to \alpha \to \alpha) \times \alpha \times (\alpha \to \alpha) \to bool$
$group\ (\circ, 1, inv) \equiv monoid\ (\circ, 1)\ \wedge\ \forall x.\ (inv\ x) \circ x = 1$
$agroup :: (\alpha \to \alpha \to \alpha) \times \alpha \times (\alpha \to \alpha) \to bool$
$agroup\ (\circ, 1, inv) \equiv group\ (\circ, 1, inv)\ \wedge\ \forall x\ y.\ x \circ y = y \circ x$

Note that *monoid* and *group*, acting on different signatures, do not admit an immediate notion of inclusion. To express that any group is a monoid one has to apply an appropriate forgetful functor first, mapping $(\circ, 1, inv)$ to $(\circ, 1)$. Operations on monoids cannot be applied to groups without this coercion.

We now use extensible records instead of fixed tuples to model algebraic structures. This will eliminate above problem of incompatible signatures, as record subtyping automatically takes care of this. Monoids are defined as follows:

record α *monoid-sig* =
$\circ :: \alpha \to \alpha \to \alpha$ (**infix**)
$1 :: \alpha$
defs
$monoid :: \{\circ :: \alpha \to \alpha \to \alpha, 1 :: \alpha, \ldots\} \to bool$
$monoid\ \{\circ, 1, \ldots\} \equiv$
$\forall x\ y\ z.\ (x \circ y) \circ z = x \circ (y \circ z)\ \wedge\ 1 \circ x = x\ \wedge\ x \circ 1 = x$

The **record** declaration introduces type scheme $\{\circ :: \alpha \to \alpha \to \alpha, 1 :: \alpha, \ldots\}$ together with several basic operations like constructors, selectors and updates (with the usual properties). Selectors are functions of the same name as the corresponding fields, e. g. $1 :: \{\circ :: \alpha \to \alpha \to \alpha, 1 :: \alpha, \ldots\} \to \alpha$. Thus $(1\ M)$ refers to the unit element of structure M. To improve readability, we also write selector application in subscript (1_M). The update operation for any field x is called *update-x*.

Based on this abstract theory of monoids, we may now introduce derived notions and prove generic theorems. For example, consider the following definition of exponentiation (by primitive recursion), together with an obvious lemma stating that $x^{m+n} = x^m \circ x^n$ holds in monoids:

defs
$pow :: \{\circ :: \alpha \to \alpha \to \alpha, 1 :: \alpha, \ldots\} \to nat \to \alpha \to \alpha$
$pow\ \{\circ, 1, \ldots\}\ 0\ x \equiv 1$
$pow\ \{\circ, 1, \ldots\}\ (Suc\ n)\ x \equiv x \circ (pow\ \{\circ, 1, \ldots\}\ n\ x)$
lemmas
$monoid\ M \Rightarrow pow_M\ (m+n)\ x = (pow_M\ m\ x) \circ_M (pow_M\ n\ x)$

Next we define groups as an extension of monoids as follows:

record α *group-sig* $=$ α *monoid-sig* $+$
$inv :: \alpha \to \alpha$
defs
$group, agroup :: \{\circ :: \alpha \to \alpha \to \alpha, 1 :: \alpha, inv :: \alpha \to \alpha, \ldots\} \to bool$
$group\ \{\circ, 1, inv, \ldots\} \equiv monoid\ \{\circ, 1, inv, \ldots\} \wedge \forall x.\ (inv\ x) \circ x = 1$
$agroup\ \{\circ, 1, inv, \ldots\} \equiv group\ \{\circ, 1, inv, \ldots\} \wedge \forall x\ y.\ x \circ y = y \circ x$

The *group-sig* type scheme has been defined as child of *monoid-sig* and directly inherits all primitive and derived operations (in particular selectors etc.). Apparently, any $\{\circ, 1, inv, \ldots\}$ is also an instance of $\{\circ, 1, \ldots\}$. Therefore, functions operating on the latter, also work on the former. For example consider the instance $pow\ \{\circ, 1, inv, \ldots\}$ for exponentiation on group structures.

By using extensible records we got for free what had to be done by explicit coercions (type casts) in other systems. Even more: apart from adapting *argument* types, *result* types are instantiated as well in our setting. As an example consider the following "functor" that reverses the binary operation of monoids:

defs
$rev\ \{\circ, 1, \ldots\} \equiv \{\circ = \lambda x\ y.\ y \circ x, 1 = 1, \ldots\}$

This function generically maps objects of type *monoid-sig* to *monoid-sig* and *group-sig* to *group-sig*:

$rev :: \{\circ :: \alpha \to \alpha \to \alpha, 1 :: \alpha, \ldots\} \to \{\circ :: \alpha \to \alpha \to \alpha, 1 :: \alpha, \ldots\}$
$rev :: \{\circ :: \alpha \to \alpha \to \alpha, 1 :: \alpha, inv :: \alpha \to \alpha, \ldots\}$
$\quad \to \{\circ :: \alpha \to \alpha \to \alpha, 1 :: \alpha, inv :: \alpha \to \alpha, \ldots\}$

Note that a naive approach with type casts would have yielded only *group-sig* to *monoid-sig* in the latter case.

In our setting, the type system will always take care of adapting the signatures of the mathematical structures automatically. Actual structures are restricted by additional logical properties, though, as expressed by the predicates *monoid*, *group*, *agroup*. Using simple properties of monoids and groups,

like $x \circ (inv\ x) = (inv\ x) \circ x$, we may actually prove that all three kinds of structures are logically invariant under the *rev* functor:

lemmas
monoid $M \Rightarrow$ *monoid* (*rev* M)
group $G \Rightarrow$ *group* (*rev* G)
agroup $G \Rightarrow$ *agroup* (*rev* G)

In general, functors may not propagate that nicely down the hierarchy of algebras. If so, one might want to consider changing the meaning of such operations depending on the actual type of the argument structure. For example, some functor on monoids might be redefined on groups in order to take the additional *inv* field into account. Redefining functions this way amounts to *overriding methods* in object-oriented parlance (see §4 of how to achieve this).

3 Extensible records with structural subtyping in HOL

3.1 The HOL logic

3.1.1 Syntax and semantics

The syntax of HOL is that of simply-typed λ-calculus with a first-order language of types. *Types* are either variables α, or applications $(\tau_1, \ldots, \tau_n)\, t$; we drop the parentheses for $n \in \{0, 1\}$. Binary constructors are often written infix, e. g. function types $\tau_1 \to \tau_2$ (associating right). There is no way to bind type variables or make types depend on terms in HOL.

Terms are either typed constants c_τ or variables x_τ, applications $t\ u$ or abstractions $\lambda x.\ t$. As usual, application associates to the left and binds most tightly. An abstraction body ranges from the dot as far to the right as possible. Nested abstractions like $\lambda x.\ \lambda y.\ t$ are abbreviated to $\lambda x\ y.\ t$. Terms have to be well-typed according to a standard set of typing rules.

HOL can be understood as a very simple version of typed set theory, with two distinct kinds of objects: terms denoting set theoretic individuals (numbers, tuples, functions etc.) and types denoting corresponding sets classifying the individuals. In ordinary untyped set theory everything is just a set, of course.

3.1.2 Theories

HOL theories consist of a *signature* (declaring type constructors $(\alpha_1, \ldots, \alpha_n)\, t$ and polymorphic constant schemes $c :: \sigma$) and *axioms*. All theories are assumed to contain a certain basis, including at least types *bool* and $\alpha \to \beta$ and several constants like logical connectives $\wedge, \vee, \Rightarrow :: bool \to bool \to bool$, quantifiers $\forall, \exists :: (\alpha \to bool) \to bool$ and equality $= :: \alpha \to \alpha \to bool$.

Any theory induces a set of derivable theorems, depending on a fixed set of deduction rules that state several "obvious" facts of classical set theory.

Arbitrary axiomatizations are considered anathema in the HOL context. It is customary to use only *definitional extensions* (guaranteeing certain nice deductive and semantic properties) and honestly toil in deriving the desired properties from the definitions. HOL offers definition schemes for constants and types [13].

3.1.3 Constant definitions

The basic mechanism only admits introducing some axiom $\vdash c \equiv t$ for a new constant c not occurring in t (and some further technical restrictions). We generalize the pure scheme to admit arguments of function definitions applied on the l. h. s. rather than abstracted on the r. h. s.: $\vdash f\ \ x\ \ y \equiv t$ instead of $\vdash f \equiv \lambda x\ y.\ t$. Furthermore, tuple abstraction, definitions by cases etc. may be written using ML-style pattern matching, e. g. $\vdash f\ (x, y) \equiv t$ (which applies the pair eliminator *split* $:: \alpha \times \beta \to (\alpha \to \beta \to \gamma) \to \gamma$ behind the scenes).

Later we will also use a proper extension of the HOL constant definition scheme, namely *overloading* [14]. Currently only Isabelle/HOL implements this. Here is a sample overloaded definition of some polymorphic constant 0:

defs
$0 :: \alpha$
$0_{nat} \equiv zero$
$0_{\alpha\ list} \equiv nil$
$0_{\alpha \times \beta} \equiv (0_\alpha, 0_\beta)$
$0_{\alpha \to \beta} \equiv \lambda x_\alpha.\ 0_\beta$

Note that we do not have to cover all types of 0 here; additional clauses may be added later, provided overall consistency of the set of equations is preserved.

3.1.4 Type definitions

New polymorphic type schemes may be introduced in HOL systematically as follows: exhibiting a non-empty representing subset A of an existing type (with further technical restrictions) one may introduce a new axiom stating that $(\alpha_1, \ldots, \alpha_n)\, t$, for a new type constructor t, is in bijection with A. This basically identifies the new type with the representing subset.

HOL type definitions are peculiar as they only state equivalence up to isomorphism. There is no way to enforce actual equality, as do type conversions in type theories. As a consequence, the HOL algebra of types can be considered as freely generated (without loss of generality), always admitting an initial model where types of different names denote different sets. This freeness property will be quite important later for distinctness of record types (§3.2). Even more fundamental, it underlies overloading [14], which is used in §4 to implement methods.

Paradoxically, more powerful logical systems like full set theory or the HOL-version underlying PVS are not quite suitable for our way of encoding extensible records, mainly because they no longer admit the freeness assumption of types.

3.2 Encoding extensible records

3.2.1 A representation in untyped set theory

Thinking in ordinary mathematics one may model extensible records as follows [7, §2.7.2]: fixing a set L of labels and a family of sets of values $(A_l)_{l \in L}$, the set of extensible records over these shall be the (dependent) partial function space $l \in L \rightharpoonup A_l$. That is, any record r is a partial function such that $r(l) \in A_l$, if $r(l)$ is defined. For example, record $\{x = 3,\ y = 5\}$ would be the function $r\colon x \mapsto 3, y \mapsto 5$, undefined elsewhere.

This encoding is rather "deep", labels and values are both first class individuals. We can express many notations of extensible records directly within the system as set theoretic functions or predicates. In particular, the relation "r has component l" would be "$r(l)$ is defined". Furthermore, relation "r' extends r" and operations "add component $l = x$ to r", "merge r and r'" could be expressed via set inclusion, insertion, union, respectively. Also note that these records are commutative: $\{x = 3,\, y = 5\}$ and $\{y = 5,\, x = 3\}$ are equal.

3.2.2 A deep encoding in HOL?

Above encoding of records would in principle also work in HOL. We could encode partial functions as relations, or total functions to a range type with explicit *undefined* element. There is a snag, though, making this version of records very awkward to use in practice: it doesn't fit very well within the HOL type system. In particular, the sets of values A_l from above would have to be within the same type! If one wanted to have different HOL types for different fields, explicit injections were required (via disjoint sums).

A better encoding of records in HOL should try to exploit the type system as much as possible. Such a representation would be much preferable even if it lost some of the properties and expressiveness of the set theoretic version. This is yet another example of applied logic within a concrete working environment where pure expressiveness may be quite unrelated to usefulness.

3.2.3 Shallow encoding of records in HOL

To make a long story short, extensible records are basically just tuples that contain an extra "more" variable for possible extensions. Ignoring the fact that field names contribute to record identity for a while, the representation of $\{x = 3,\, y = 5,\, f = \mathit{true},\, \ldots\}$ is just $(3, (5, (\mathit{true}, \mathit{more})))$ where *more* is a suitable term variable. The corresponding type $\{x :: \mathit{int},\, y :: \mathit{int},\, f :: \mathit{bool},\, \ldots\}$ is a nested product $(\mathit{int} \times (\mathit{int} \times (\mathit{bool} \times \alpha)))$, for some free type variable α.

Refining the *more* slot yields instances with additional fields, for example $\{x = 3,\, y = 5,\, f = \mathit{true},\, z = 42,\, \ldots\}$ represented by $(3, (5, (\mathit{true}, (42, \mathit{more}'))))$. Containing free variables, record schemes are not basic values. Typically, they only appear in definitions of generic functions where *more* is bound by functional abstraction. On the level of types, the *more* position amounts to polymorphism.

Actual concrete record values can be achieved by instantiating the more slot to $()$, the sole element of the *unit* type, thus terminating the chain properly without affecting the semantics. For example, $\{x = 3,\, y = 5,\, f = \mathit{true}\}$ would be $(3, (5, (\mathit{true}, ())))$, and consequently its type $\{x :: \mathit{int},\, y :: \mathit{int},\, f :: \mathit{bool}\}$ would be $(\mathit{int} \times (\mathit{int} \times (\mathit{bool} \times \mathit{unit})))$.

We now focus again on labels. These shall act as a means to distinguish records with different field names. As we have already said earlier, HOL's algebra of types is so weak that it admits a freeness assumption: types of different names can never be enforced to be actually the same within the logic. This gives rise to the following technique to make field names contribute to record identity without having to bother about labels as first-class individuals.

For any field $x :: \sigma$ we introduce an isomorphic copy of the HOL pair type $\times$ by type definition, calling it $\times_x$. We also obtain copies of the pair constructor and projections etc., with their usual properties. The copied constructor shall

be *x-field* $:: \sigma \to \beta \to \sigma \times_x \beta$. It is declared only at an instance of the general scheme $\alpha \to \beta \to \alpha \times_x \beta$ in order to obey the type constraint for field x as specified in the record type declaration.

Using a separate pair type for any field we now get the following encoding of records: $\{x = 3, y = 5, f = \mathit{true}, \ldots\}$ is (*x-field* 3 (*y-field* 5 (*f-field* *true* *more*))), its type $\{x :: \mathit{int}, y :: \mathit{int}, f :: \mathit{bool}, \ldots\}$ becomes $(\mathit{int} \times_x (\mathit{int} \times_y (\mathit{bool} \times_f \alpha)))$. Constructing records this way is like building inhomogeneous lists, with a separate *cons* operator for each field. The system implementation can easily provide concrete syntax for our records and do the conversion to the representation.

There are several distinguishing features of our encoding of extensible records in HOL, as compared to the set theoretic one presented earlier.

Most prominently, labels are not first class, but part of constant and type names (*x-field* and $\times_x$). Thus we can no longer refer directly to fields within the logic, "record r has field l" is not a HOL relation in our setting. Yet this does not prevent us to write generic functions $f\ \{x = a, y = b, \ldots\}$ that expect certain fields. This is actually the way we get record subtyping for free, in the guise of ordinary polymorphism. So we gain a lot by directly employing the HOL type system for record types.

Also, our records are not commutative: $\{x = 3, y = 5\}$ and $\{y = 5, x = 3\}$ are different, even of incompatible types. So one has to ensure that records obey a canonical order of fields, which is not considered an actual limitation.

Furthermore, we do not provide a record merge operation. This would be basically concatenation of record types, requiring an associative operator. HOL with its free first-order type system cannot express this. We merely loose multiple inheritance because of this.

Note that our way of encoding extensible records via nested copies of product types $\alpha \times_x \beta \times_y \cdots$ could be applied to functional programming languages like ML, too. Thus one would get a form of functional object-oriented language very easily, ML let-polymorphism would take care of schematic record subtyping in the same way as in HOL. There would be some extra limitation in ML, though, because HOL-style ad-hoc polymorphism is unavailable. This would prevent the encoding of overridable methods as presented in §5.1.

Obviously one could also apply the same basic idea to co-product types, yielding extensible *co*-records: disjoint sums $\alpha +_x \beta +_y \cdots$. Then one could express functions that are generic wrt. the available cases of the sum type. Note that there is no proper way of terminating schematic sums in HOL, as the empty type (co-*unit*) is unavailable; using actual *unit* instead might still yield a useful concept in practice, though.

The next step beyond products and sums is recursion, namely inductive or co-inductive datatypes. It is still to be seen if our way of internalizing "..." has any useful application in the latter area.

4 An environment for object-oriented verification

We now introduce a logical environment that supports object-oriented concepts like classes, instantiation and inheritance. Our theory syntax will be similar to conventional object-oriented languages, like the one proposed in [10]. In this section we will only give some hints on how all of this can be implemented in terms of ordinary HOL declarations and definitions, see §5 for more details.

We use points, coloured points and rectangles as a running example. The root class *point* has x- and y-coordinates as fields, method *move* for moving points by a given offset and methods *reflect-X*, *reflect-Y*, *reflect-O* for reflecting them along the abscissa, ordinate, origin, respectively. Class *cpoint* adds a colour component to points. Class *rectangle* is a subclass of *cpoint* and specifies rectangles, which are determined by a reference point (bottom-left) together with the width and height. Rectangles are always in parallel to the x/y-axes. We also introduce a class *rectangle-hilite* of rectangles that set the colour to red when being moved.

4.1 Classes

To begin with the example, we define a root class *point*.

class *point* =

fields $x, y :: int$

methods

$move :: \{x :: int, y :: int, \ldots\} \to int \to int \to \{x :: int, y :: int, \ldots\}$
$move\ \{x, y, \ldots\}\ dx\ dy \equiv \{x + dx, y + dy, \ldots\}$
$reflect\text{-}X :: \{x :: int, y :: int, \ldots\} \to \{x :: int, y :: int, \ldots\}$
$reflect\text{-}X\ \{x, y, \ldots\} \equiv this.move\ \{x, y, \ldots\}\ 0\ (-2 \cdot y)$
$reflect\text{-}Y :: \{x :: int, y :: int, \ldots\} \to \{x :: int, y :: int, \ldots\}$
$reflect\text{-}Y\ \{x, y, \ldots\} \equiv this.move\ \{x, y, \ldots\}\ (-2 \cdot x)\ 0$

final methods

$reflect\text{-}O :: \{x :: int, y :: int, \ldots\} \to \{x :: int, y :: int, \ldots\}$
$reflect\text{-}O \equiv this.reflect\text{-}Y \circ this.reflect\text{-}X$

specification

$move\ p\ 0\ 0 =^{x,y} p$ (1)
$x\ (move\ p\ dx\ dy) = x\ p + dx$ (2)
$y\ (move\ p\ dx\ dy) = y\ p + dy$ (3)
$this.move\ (reflect\text{-}X\ p)\ dx\ dy =^{x,y}$ (4)
$\quad reflect\text{-}X\ (this.move\ p\ dx\ (-dy))$
$reflect\text{-}Y\ (reflect\text{-}X\ p) =^{x,y} reflect\text{-}X\ (reflect\text{-}Y\ p)$
$reflect\text{-}X\ (reflect\text{-}X\ p) =^{x,y} p$
$reflect\text{-}Y\ (reflect\text{-}Y\ p) =^{x,y} p$
$reflect\text{-}O\ (reflect\text{-}O\ p) =^{x,y} p$

We refer to methods in two ways, written with or without a prefix *this*. This distinction plays a vital rôle for inheritance, but can be ignored at the moment. Note that we use a particular equality $=^{x,y}$ which expresses that two points coincide on the coordinates, but not necessarily on the remaining fields. To improve readability, correctness proofs are not shown here. Verification issues are discussed in §6.

Since we are within a functional setting, state-modifying methods are modeled as functions mapping states to states. To be more precise, methods do not operate on particular states but on arbitrary instances of a given state scheme.

Mutual dependencies of methods are acceptable as long as they are non-circular. Recursive definition of methods is not supported as a primitive. The user has to express this using appropriate operators from the underlying logic (e. g. well-founded recursion).

4.2 Objects, instantiation, and method invocation

Objects are *instantiated* from classes by specialization. Instantiating some concrete object *MyPoint* from class *point* is achieved by specializing the state-space from $\{x :: int, y :: int, \ldots\}$ to $\{x :: int, y :: int\}$ and determining the initial values for the coordinates. We write $MyPoint \equiv new\ point\ \{x = 3, y = 5\}$.

Method invocation is simply achieved by function application. For example, we can reset the object *MyPoint* by $move\ MyPoint\ (-x_{MyPoint})\ (-y_{MyPoint})$.

4.3 Inheritance

Inheritance means being able to reuse code of superclasses in subclasses without explicit alteration. At first sight, this problem seems to be trivial just by duplicating code, but the problem is slightly more complicated. Consider, for example, the point methods, which operate on an x- and y-coordinate whereas the same methods (seen as methods of coloured points) have to operate on an extended state-space, which contains a colour field, too. Using extensible records we are able to write code for point methods generically such that the methods can operate on any state-space which contains at least x- and y-coordinates. Hence our implementation of the *point* methods can be used in a class of coloured points *cpoint* without alteration. This is what we achieve by the following definition:

datatype *colour* = *Red* | *Green* | *Blue*
class *cpoint* = *point* + **fields** *col* :: *colour*

As suggested by above "+" notation, class *cpoint* includes all fields and methods from *point*.

4.4 Overriding

To continue with the example, we define a new class *rectangle*, adding fields w, h and method *area*. Reflecting a rectangle cannot be achieved by simply reflecting the reference point. When reflecting the bottom-left point along the x-axis it becomes the top-left point, so we have to subtract the height of the rectangle from its y-value to fix this. An analogous correction has to be performed for

the reflection along the x-axis.

class *rectangle* = *cpoint* +

fields $w, h :: nat$

methods

$area :: \{x :: int, y :: int, col :: colour, w :: nat, h :: nat, \ldots\} \to nat$

$area\ \{x, y, col, w, h, \ldots\} \equiv w \cdot h$

override methods

$reflect\text{-}X\ \{x, y, col, w, h, \ldots\} \equiv$
$\quad this.move\ (point.reflect\text{-}X\ \{x, y, col, w, h, \ldots\})\ 0\ (-h)$

$reflect\text{-}Y\ \{x, y, col, w, h, \ldots\} \equiv$
$\quad this.move\ (point.reflect\text{-}Y\ \{x, y, col, w, h, \ldots\})\ (-w)\ 0$

specification

$area\ (move\ \{x, y, col, w, h, \ldots\}\ dx\ dy) = area\ \{x, y, col, w, h, \ldots\}$

Apart from *reflect-X* and *reflect-Y* all methods and all lemmas of *cpoint* are inherited. At first sight it appears evident what we mean by saying "all other methods are inherited". But life is not as easy as it seems. Recall the definition of *reflect-O* in *point*: $reflect\text{-}O \equiv this.reflect\text{-}Y \circ this.reflect\text{-}X$. On the one hand we have inherited this method, on the other hand we have overridden the methods *reflect-X* and *reflect-Y* in *rectangle*. If *this.reflect-X* and *this.reflect-Y* referred statically to the methods defined in *point* the method *reflect-O* would not behave as expected for rectangles. Instead, the references to *reflect-X* and *reflect-Y* in the inherited method *reflect-O* must refer dynamically to the redefined methods. In the following section we will have a closer look at this dynamic binding of methods which sometimes is also called *late-binding*.

4.5 Late-binding

Late-binding of methods is a powerful mechanism, making reuse of code very flexible. To back up this claim we extend rectangles by a class *rectangle-hilite*. The idea is that relocated rectangles are highlighted in red colour. Without late-binding of methods we would have to redefine *all* methods (except for *area*). The impact of these modifications on the correctness proofs would be disastrous: almost all proofs about rectangles would have to be repeated, quite redundantly though. Using late-binding of methods, the definition of *rectangle-hilite* is very simple because all methods relocating rectangles are defined directly or indirectly in terms of the generic *move*.

class *rectangle-hilite* = *rectangle* +

override methods

$move \equiv (update\text{-}colour\ \ Red) \circ rectangle.move$

specification

$col\ (reflect\text{-}X\ \{x, y, col, w, h, \ldots\}) = Red$
$col\ (reflect\text{-}Y\ \{x, y, col, w, h, \ldots\}) = Red$
$col\ (reflect\text{-}O\ \{x, y, col, w, h, \ldots\}) = Red$

The fact that we can prove these properties of *reflect-X*, *reflect-Y* and *reflect-O* is remarkable. Without having redefined any of these methods, the change of the *move* method has been propagated automatically. This demonstrates that object-oriented verification really does work in our environment.

Now we have arrived at a point where we can clarify the distinction between those methods prefixed by *this* and those which are not. Methods prefixed by *this* are late-bound and may change in subclasses whereas the others are fixed. For a better understanding of the distinction recall equation (4) from *point*:

$$this.move\ (reflect\text{-}X\ p)\ dx\ dy =^{x,y} reflect\text{-}X\ (this.move\ p\ dx\ (-dy))$$

This equation expresses that all implementations of the late-bound method *this.move* in subclasses are well behaved together with the particular implementation *reflect-X* of *point*. Expanding the definition of *reflect-X* — we cannot expand any definition of *this.move* since it is late-bound — in *point* yields:

$$\begin{aligned} &this.move\ (this.move\ p\ 0\ (-2 \cdot (y\ p)))\ dx\ dy =^{x,y} \\ &\quad this.move\ (this.move\ p\ dx\ (-dy))\ 0\ (-2 \cdot (y\ (this.move\ p\ dx\ (-dy)))) \end{aligned}$$

Of course, there are implementations of *this.move* invalidating this equation. However, it is true for all those implementations satisfying the equations for *move* given in *point*. Assuming that equations (1)–(3) hold for all implementations of *this.move* in subclasses, we can always show equation (4). This implies that (4) can be inherited in *rectangle-hilite* although method *move* has been overridden. Of course, we do not get all proofs for free. Since we have overridden *move*, we have to redo the proofs for all equations containing a particular implementation of *move* (without prefix *this*, that is).

5 Encoding of object-oriented concepts in HOL

We now show that the object-oriented concepts presented in §4 are only a stone's throw away from a rigorous encoding in HOL.

States are represented as extensible records and methods as state transforming functions. As we have already seen, we can achieve inheritance simply by record subtyping. Things are getting much more complicated when taking late-binding into account. What makes it hard to model is that the semantics of late-bound methods changes relatively to the position in the inheritance hierarchy. Assuming different field types for different levels of the hierarchy, we can use *overloading* to achieve different meaning of methods in different contexts. Assuming different field types for different levels is no real restriction, since we can always enforce them by adding dummy fields.

5.1 Classes

First of all, the fields of any class definition become a record type definition:

record *point* =
  $x, y :: int$

Methods are more involved. The simplest method of *point* is *move*, because it is not late-bound.

First attempt One might try to realize method *move* in HOL directly as suggested in the *point* class definition:

defs
  $move :: \{x :: int, y :: int, \ldots\} \to int \to int \to \{x :: int, y :: int, \ldots\}$
  $move\ \{x, y, \ldots\}\ dx\ dy \equiv \{x + dx, y + dy, \ldots\}$

The problem with this definition is that it is too generic. Since *move* is defined for *all* records containing x- and y-coordinates, we cannot override this definition in subclasses any more.

Second attempt To remedy this problem one might *declare* the method generically, but *define* it on concrete records only:

defs
$move :: \{x :: int, y :: int, \ldots\} \to int \to int \to \{x :: int, y :: int, \ldots\}$
$move\ \{x, y\}\ dx\ dy \equiv \{x + dx, y + dy\}$

With this definition we are able to express overriding and late-binding. Overriding is achieved simply by defining *move* on a different concrete instance of the scheme $\{x :: int, y :: int, \ldots\}$, say on $\{x :: int, y :: int, col :: colour\}$. To see, how we can achieve late-binding consider the definition of a method *reset* which sets points to the origin:

defs
$reset :: \{x :: int, y :: int, \ldots\} \to \{x :: int, y :: int, \ldots\}$
$reset\ \{x, y, \ldots\} \equiv move\ \{x, y, \ldots\}\ (-x)\ (-y)$

Since we have given no definition of *move* on the extensible record type, its semantics and hence the semantics of *reset* is unspecified. Restricting the extensible record type to the concrete one $\{x :: int, y :: int\}$, determines a meaning as given by definition of *move*. Restricting it to a different concrete record may result in a different meaning, depending on the definition of *move* given there.

There is still a snag: we have ruled out inheritance. By defining *move* on a concrete record type we lose the ability to reuse code in subclasses.

The solution To achieve both overriding and inheritance, we define two constants *point.move* and *this.move* rather than a single constant *move*, allowing the character "." to be part of identifiers. The actual implementation of the *move* method in HOL is as follows:

defs
$point.move, this.move ::$
$\quad \{x :: int, y :: int, \ldots\} \to int \to int \to \{x :: int, y :: int, \ldots\}$
$point.move\ \{x, y, \ldots\}\ dx\ dy \equiv \{x + dx, y + dy, \ldots\}$
$this.move\ \{x, y\} \equiv point.move\ \{x, y\}$

Apart from *reflect-O*, the remaining methods are defined analogously. Since *reflect-O* is a *final* method, we have to guarantee that it cannot be overridden. Defining it on an extensible record type achieves this:

defs
$point.reflect\text{-}O, this.reflect\text{-}O ::$
$\quad \{x :: int, y :: int, \ldots\} \to \{x :: int, y :: int, \ldots\}$
$point.reflect\text{-}O \equiv this.reflect\text{-}Y \circ this.reflect\text{-}X$
$this.reflect\text{-}O \equiv point.reflect\text{-}O$

The need of two definitions for one method is no real problem for the user. These definitions can be generated automatically by some extra-logical system support.

5.2 Objects and instantiation

Instantiation is trivial in our framework: let $MyPoint \equiv \{x = 3, y = 5\}$. The simplicity of instantiation stems from the fact that we generate both generic *class methods* and concrete *object methods* in classes. In a sense, we have anticipated instantiation already by the way we define classes.

5.3 Inheritance

Inheritance is just as simple as instantiation. For inheritance, all we have to do is specialize the class methods of the superclass to concrete object methods of the subclass. Class *cpoint* leads to the following definitions in HOL:

record $cpoint = point\ +$
$col :: colour$

defs
$this.move\ \{x, y, col\} \equiv point.move\ \{x, y, col\}$
$this.reflect\text{-}X\ \{x, y, col\} \equiv point.reflect\text{-}X\ \{x, y, col\}$
$this.reflect\text{-}Y\ \{x, y, col\} \equiv point.reflect\text{-}Y\ \{x, y, col\}$

Interestingly, we do not have to give a definition for *reflect-O* once more. Since *reflect-O* was defined for the scheme $\{x :: int, y :: int, \ldots\}$ its definition works equally well on the concrete type $\{x :: int, y :: int, col :: colour\}$.

Since the methods have not been altered in *cpoint*, lemmas proved for points also hold for coloured points. Sticking to object-oriented terminology, we might say that the proofs are inherited. In a type-theoretic framework with explicit proof-terms this terminology fits perfectly well (see also [5]).

5.4 Overriding

Class *cpoint* serves as an example for inheritance, but it does not demonstrate overriding. Overriding is achieved simply by defining new methods. In case of class *rectangle* we define methods *rectangle.reflect-X* and *rectangle.reflect-Y*:

record $rectangle = cpoint\ +$
$w, h :: nat$

defs
$rectangle.reflect\text{-}X\ \{x, y, col, w, h, \ldots\} \equiv$
$\quad this.move\ (point.reflect\text{-}X\ \{x, y, col, w, h, \ldots\})\ 0\ (-h)$ (5)
$this.reflect\text{-}X\ \{x, y, col, w, h\} \equiv rectangle.reflect\text{-}X\ \{x, y, col, w, h\}$
$rectangle.reflect\text{-}Y\ \{x, y, col, w, h, \ldots\} \equiv$
$\quad this.move\ (point.reflect\text{-}Y\ \{x, y, col, w, h, \ldots\})\ (-w)\ 0$
$this.reflect\text{-}Y\ \{x, y, col, w, h\} \equiv rectangle.reflect\text{-}Y\ \{x, y, col, w, h\}$

5.5 Late-binding

Class *rectangle-hilite* is a good example for late-binding of methods. Apart from late-binding, class *rectangle-hilite* is interesting because it introduces no new fields. Since we have identified class membership with field types, we have to tell

the field types of *rectangle-hilite* and *rectangle* apart by adding an artificial field *dummy* of type *unit*. For simplicity we omit some obvious method definitions.

record *rectangle-hilite* = *rectangle* +
  dummy :: *unit*
defs
  rectangle-hilite.move ::
    $\{x :: int, y :: int, col :: colour, w :: nat, h :: nat, dummy :: unit, \ldots\} \rightarrow$
      $\{x :: int, y :: int, col :: colour, w :: nat, h :: nat, dummy :: unit, \ldots\}$
  rectangle-hilite.move $\equiv$ (*update-col Red*) $\circ$ *rectangle.move* (6)
  this.move $\{x, y, col, w, h, dummy\} \equiv$
    rectangle-hilite.move $\{x, y, col, w, h, dummy\}$ (7)
  this.reflect-X $\{x, y, col, w, h, dummy\} \equiv$
    rectangle.reflect-X $\{x, y, col, w, h, dummy\}$ (8)

In this class, method *move* additionally sets the colour to *Red*. All methods defined in terms of *move* show the same effect, as can be seen by expansion of their definitions. Take for example method *this.reflect-X* (we abbreviate the term *point.reflect-X* $\{x, y, col, w, h, dummy\}$ by Δ):

proof
  this.reflect-X $\{x, y, col, w, h, dummy\}$ = by (8)
  rectangle.reflect-X $\{x, y, col, w, h, dummy\}$ = by (5)
  this.move $\Delta\ 0\ (-h)$ = by (7)
  rectangle-hilite.move $\Delta\ 0\ (-h)$ = by (6)
  ((*update-col Red*) $\circ$ *rectangle.move*) $\Delta\ 0\ (-h)$

Be aware, that *this.reflect-X* may have a completely different meaning on different state spaces.

6 Object-oriented verification in HOOL

Subsequently we investigate up to what extent object-oriented concepts, developed to structure programs, provide means to structure verification, too. Since we have introduced two kinds of methods, class methods and object methods, we naturally expect two kinds of lemmas. In the end, though, we are only interested in those lemmas about object methods.

Object methods What distinguishes object methods and class methods, anyhow? There are two main characteristics for object methods: they are prefixed by *this* (which is merely a syntactic convention) and they are only defined on concrete records. Proving lemmas about object methods does not require any particular methodology. Take for example the following equation

$$this.move\ \{x, y\}\ 0\ 0\ =^{x,y} \{x, y\}$$

which is immediately proven by rewriting. Proving lemmas on object methods directly, though possible in principle, is not very clever: we do not exploit object-oriented structuring principles for verification. We argue now that verification of class methods entails abstract and thus structured verification.

Class methods There are both late-bound and fixed class methods. Late-bound methods are prefixed by *this* (again, this is only a syntactic convention). More importantly, they are *only predeclared*, without fixing a definition yet,

e. g $this.move :: \{x :: int, y :: int, \ldots\} \to int \to int \to \{x :: int, y :: int, \ldots\}$. Fixed class methods are prefixed by class names, *declared and defined* on extensible record types and may use late-bound methods for definition. As an example consider:

defs
$point.reflect\text{-}X :: \{x :: int, y :: int, \ldots\} \to \{x :: int, y :: int, \ldots\}$
$point.reflect\text{-}X\ \{x, y, \ldots\} \equiv this.move\ \{x, y, \ldots\}\ 0\ (-2 \cdot y)$

For fixed methods not referring to late-bound methods we can prove lemmas directly. Take for example equation (1) with every occurrence of *move* replaced by *point.move* — we write $(1)[point.move/move]$. This lemma is immediately proven by rewriting. Since the lemma expresses a property for all state-spaces which contain at least x- and y-coordinates, this lemma holds in all subclasses as long as method *point.move* is inherited. The same holds for the next three equations. By restricting the state-space to concrete records, we get the corresponding lemmas for object methods for free (by HOL type instantiation).

What happens if fixed methods refer to late-bound methods? Since late-bound methods are only declared, we cannot expect non-trivial lemmas to hold for such methods. To prove interesting lemmas we have to assume properties of the late-bound methods. For class definitions we apply the convention that the lemmas are ordered by position and (implicitly) have the preceding lemmas as assumptions. To be precise, the n-th lemma of *point* is translated to the following formula in HOL (where (i) stands for the i-th lemma of *point*, and $[this.m/m]$ or $[point.m/m]$ for prefixing all non-prefixed methods by *this* or *point*, respectively):

$$\bigwedge_{i=1}^{n-1} (i)[this.m/m] \Rightarrow (n)[point.m/m]$$

The question arising immediately is how to get rid of assumptions. On the one hand, we cannot discharge them in classes (which are basically abstract theories). On the other hand, assumptions can be eliminated in any concrete instance, where class methods are specialized to object methods.

Finally, we explain observational equality $=^{x,y}$ which is defined as follows:

defs
$=^{x,y} :: \{x :: int, y :: int, \ldots\} \to \{x :: int, y :: int, \ldots\} \to bool$
$p =^{x,y} q \equiv (x\ p) = (x\ q) \wedge (y\ p) = (y\ q)$

We use observational equality $=^{x,y}$ rather than ordinary equality for specification in cases when we only want to fix the meaning of methods on the coordinates. To see why this is more appropriate than actual equality, recall the definition of *move* in *rectangle-hilite*: for full equality, equation (1) would no longer hold in *rectangle-hilite* because the *col* field is manipulated there.

Let us leave our running example and see what we have achieved. We can specify non-late-bound class methods generically and thus inherit the proofs in subclasses immediately. To deal with late-bound methods we have to add assumptions to the equations to be proven. But we know that in all subclasses we can discharge the assumptions.

What happens if we override methods? Depending on which kind of method is used we get different consequences. If a method is late-bound, we cannot use any information about its implementation for the correctness proof and hence

we can inherit the proof even if the method is overridden. If a method occurs non-late-bound at least once, we have to perform a new proof.

So to cut a long story short, appropriate use of late-bound methods does not only cater for flexible reuse of code, it also provides a mechanism for generic and thus reusable correctness proofs.

7 Conclusions

Stocktaking

We have seen how to obtain extensible records with structural subtyping in HOL in a very natural way. Records have been encoded as nested tuples with a special "more" slot for extension, structural subtyping turned out to be subsumed by parametric polymorphism. We also employed overloading to account for overridable methods and late-binding. Together with concrete syntax and some minimal extra-logical system support we arrived at an environment for object-oriented specification and verification (HOOL).

The only major object-oriented concept we have not dealt with explicitly in this paper is encapsulation (hiding). Due to the lack of existential types in HOL, we cannot use the standard trick [8] for data abstraction. Any abstract theory mechanism, e. g. axiomatic type classes [14], achieves abstraction from concrete implementations as well.

Note that in actual verification tasks, simple concepts like type classes are more convenient than existential types. In particular, method invocation is quite involved in encodings based on existential types [12]: the encapsulated state has to be opened and (in the case of state-modifying methods) be repacked again.

Related work

Modeling object-oriented concepts has become a vast field of research in itself. We only give a few hints of how HOOL is related to other work in the context of theorem proving in higher-order logics.

Hensel, Jacobs et al. [4, 6] define an object-oriented specification language based on co-algebra and give a translation to generic higher-order logic, resulting in a tool environment for object-oriented system verification (LOOP). Their translation does not fully preserve the object-oriented structure, though, yet it is quite independent of the underlying logic. Thus LOOP allows to do the reasoning about classes in different target theorem proving systems (e. g. PVS, Isabelle).

Hofmann et al. [5] employ type theory (ECC) for object-oriented verification. They use object-oriented concepts in a very similar way. Compared with their work, the most important contribution of HOOL is its simplicity. It does not require any advanced type theoretical concepts such as dependent types and Σ-types for the encoding.

Nevertheless, HOOL can compete with their approach quite well since the resulting expressiveness is almost the same from the user's point of view. Restricting to the much simpler HOL logical framework, though, pays off by shortening the effort for correctness proofs substantially.

Naraschewski [10] proposed an environment for object-oriented verification similar to HOOL, but based on a rather different encoding of objects in ECC. Though Naraschewski sketched a translation from this environment to an encoding in LEGO, there are two drawbacks of this approach. First, translating the proposed environment to the LEGO system would require extensive extra-logical support. Second, the encoding in LEGO, though feasible in principle, is not directly applicable for practical work.

Does object-oriented verification actually pay off in practice? The only case-study performed so far deals with a hierarchy of collection classes [9]. It turned out that late-binding of methods really helps to structure verification. Using the complicated encoding of [5], though, it was not possible to scale up to larger examples. With the simple encoding of object-oriented concepts in HOOL, we expect to have overcome these limitations.

References

[1] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.

[2] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56–68, 1940.

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

[4] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In *Proceedings of ESOP/ETAPS*, volume 1381 of *Lecture Notes in Computer Science.* Springer-Verlag, 1998.

[5] M. Hofmann, W. Naraschewski, M. Steffen, and T. Stroup. Inheritance of proofs. *Theory and Practice of Object Systems, Special Issue on Third Workshop on Foundations of Object-Oriented Languages (FOOL 3)*, 4(1):51–69, 1998.

[6] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes (preliminary report). Technical Report CSI-R9812, Computing Science Institute Nijmegen, April 1998.

[7] L. Lamport and L. C. Paulson. Should your specification language be typed? Technical Report 425, University of Cambridge Computer Laboratory, 1997.

[8] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Prog. Lang. Syst.*, 10(3):470–502, July 1988.

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

[10] W. Naraschewski. Towards an object-oriented progification language. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, pages 215–230. Springer-Verlag, 1997.

[11] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[12] B. C. Pierce and D. N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[13] A. Pitts. The HOL logic. In Gordon and Melham [3], pages 191–232.

[14] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs'97*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer-Verlag, 1997.