Constructive Type Classes in Isabelle

Florian Haftmann* and Makarius Wenzel **

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany
http://www.in.tum.de/~haftmann/
http://www.in.tum.de/~wenzelm/

Abstract. We reconsider the well-known concept of Haskell-style type classes within the logical framework of Isabelle. So far, axiomatic type classes in Isabelle merely account for the logical aspect as predicates over types, while the operational part is only a convention based on raw overloading. Our more elaborate approach to constructive type classes provides a seamless integration with Isabelle locales, which are able to manage both operations and logical properties uniformly. Thus we combine the convenience of type classes and the flexibility of locales. Furthermore, we construct dictionary terms derived from notions of the type system. This additional internal structure provides satisfactory foundations of type classes, and supports further applications, such as code generation and export of theories and theorems to environments without type classes.

1 Introduction

The well-known concept of type classes [18, 15, 6, 13, 10, 19] offers a useful structuring mechanism for programs and proofs, which is more light-weight than a fully featured module mechanism. Type classes are able to qualify types by associating operations and logical properties. For example, class *eq* could provide an equivalence relation $=$ on type α, and class *ord* could extend *eq* by providing a strict order $<$ etc.

Programming languages like Haskell merely handle the operational part [18, 15, 6]. In contrast, type classes in Isabelle [14, 12] directly represent the logical properties, but the associated operations are treated as a mere convention imposed on the user.

Recent Isabelle add-ons have demanded more careful support of type classes, most notably code generation from Isabelle/HOL to SML, or conversion of Isabelle/HOL theories and theorems to other versions of HOL. Here the target language lacks direct support for type classes, so the source representation in the Isabelle framework somehow needs to accommodate this, using similar techniques as those performed by the static analysis of Haskell.

How does this work exactly? Haskell is not a logical environment, and internal program transformations are taken on faith without explicit deductions. In traditional Isabelle type classes, the purely logical part could be directly embedded into the logic [19], although some justifications were only done on paper. The operational aspect, which cannot be fully internalized into the logic, was explained by raw overloading.

* Supported by DFG project NI 491/10-1
** Supported by BMBF project "Verisoft"

Furthermore, the key disadvantage of raw type classes as "little theories" used to be the lack of flexibility in the signature part: operations being represented by polymorphic constants are fixed for any given type. On the other hand, in recent years the Isabelle infrastructure for structured specifications and proofs has been greatly improved, thanks to the Isar proof language [20, 21, 11] and locales [8, 1, 2]. We think it is time to reconsider the existing type class concepts, and see how they can benefit from these improvements without sacrificing their advantages.

The present work integrates Isabelle type classes and locales (by means of locale interpretation), and provides more detailed explanations of type classes with operations and logical propositions within the existing logical framework. Here we heavily re-use a careful selection of existing concepts, putting them into a greater perspective. We also reconstruct the essential relationship between the type system and its constructive interpretation by producing dictionary terms for class operations. The resulting concept of "constructive type classes" in Isabelle is both more convenient for the user, and more satisfactory from the foundational viewpoint.

2 Example

We demonstrate common elements of structured specifications and abstract reasoning with type classes by the algebraic hierarchy of semigroups and groups. Our background theory is that of Isabelle/HOL [12], which uses fairly standard notation from mathematics and functional programming. We also refer to basic vernacular commands for definitions and statements, e.g. **definition** and **lemma**; proofs will be recorded using structured elements of Isabelle/Isar [20, 21, 11], notably **proof/qed** and **fix/assume/show**.

Our main concern are the new **class** and **instance** elements used below — they will be explained in terms of existing Isabelle concepts later (§5). Here we merely present the look-and-feel for end users, which is quite similar to Haskell's `class` and `instance` [6], but augmented by logical specifications and proofs.

2.1 Class definition

Depending on an arbitrary type α, class *semigroup* introduces a binary operation $\circ$ that is assumed to be associative:

class *semigroup* =
  fixes *mult* :: $\alpha \Rightarrow \alpha \Rightarrow \alpha$ (**infix** $\circ$ 70)
  assumes *assoc*: $(x \circ y) \circ z = x \circ (y \circ z)$

This **class** specification consists of two parts: the *operational* part names the class operation (**fixes**), the *logical* part specifies properties on them (**assumes**). The local **fixes** and **assumes** are lifted to the theory toplevel, yielding the global operation *mult* :: α::*semigroup* $\Rightarrow \alpha \Rightarrow \alpha$ and the global theorem *semigroup.assoc*: $\bigwedge x\ y\ z$::α::*semigroup*. $(x \circ y) \circ z = x \circ (y \circ z)$.

2.2 Class instantiation

The concrete type *int* is made a *semigroup* instance by providing a suitable definition for the class operation *mult* and a proof for the specification of *assoc*.

instance *int* :: *semigroup*
  mult-int-def: $\bigwedge i\ j :: int.\ i \circ j \equiv i + j$
proof
  fix *i j k* :: *int* **have** $(i + j) + k = i + (j + k)$ **by** *simp*
  then show $(i \circ j) \circ k = i \circ (j \circ k)$ **unfolding** *mult-int-def* **.**
qed

From now on, the type-checker will consider *int* as a *semigroup* automatically, i.e. any general results are immediately available on concrete instances.

2.3 Subclasses

We define a subclass *group* by extending *semigroup* with additional operations *neutral* and *inverse*, together with the usual *left-neutral* and *left-inverse* properties.

class *group* = *semigroup* +
  fixes *neutral* :: α $(\mathbf{1})$
    and *inverse* :: $\alpha \Rightarrow \alpha$ $((\text{-}^{-\mathbf{1}})\ [1000]\ 999)$
  assumes *left-neutral*: $\mathbf{1} \circ x = x$
    and *left-inverse*: $x^{-\mathbf{1}} \circ x = \mathbf{1}$

Again, type *int* is made an instance, by providing definitions for the operations and proofs for the axioms of the additional group specification.

instance *int* :: *group*
  neutral-int-def: $\mathbf{1} \equiv 0$
  inverse-int-def: $i^{-1} \equiv -\ i$
proof
  fix *i* :: *int* **have** $0 + i = i$ **by** *simp*
  then show $\mathbf{1} \circ i = i$ **unfolding** *mult-int-def* **and** *neutral-int-def* **.**
  have $-i + i = 0$ **by** *simp*
  then show $i^{-1} \circ i = \mathbf{1}$ **unfolding** *mult-int-def* **and** *neutral-int-def* **and** *inverse-int-def* **.**
qed

2.4 Abstract reasoning

Abstract theories enable reasoning at a general level, while results are implicitly transferred to all instances. For example, we can now establish the *left-cancel* lemma for groups, which states that the function $(x \circ)$ is injective:

lemma (**in** *group*) *left-cancel*: $x \circ y = x \circ z \leftrightarrow y = z$
proof
  assume $x \circ y = x \circ z$
  then have $x^{-\mathbf{1}} \circ (x \circ y) = x^{-\mathbf{1}} \circ (x \circ z)$ **by** *simp*
  then have $(x^{-\mathbf{1}} \circ x) \circ y = (x^{-\mathbf{1}} \circ x) \circ z$ **using** *assoc* **by** *simp*
  then show $y = z$ **using** *left-neutral* **and** *left-inverse* **by** *simp*

next
  assume $y = z$
  then show $x \circ y = x \circ z$ **by** *simp*
qed

Here the "**in** *group*" target specification indicates that the result is recorded within that context for later use. This local theorem is also lifted to the global one *group.left-cancel*: $\bigwedge x\ y\ z{::}\alpha{::}$*group*. $x \circ y = x \circ z \leftrightarrow y = z$. Since type *int* has been made an instance of *group* before, we may refer to that fact as well: $\bigwedge x\ y\ z{::}$*int*. $x \circ y = x \circ z \leftrightarrow y = z$.

3 Logical foundations

We briefly review fundamental concepts of the Isabelle/Isar framework, from the Pure logic to Isar proof contexts (structured proofs) and locales (structured specifications).

3.1 The Isabelle/Pure framework

The Pure logic [14] is an intuitionistic fragment of higher-order logic. In type-theoretic parlance, there are three levels of λ-calculus with corresponding arrows: $\Rightarrow$ for syntactic function space (terms depending on terms), $\bigwedge$ for universal quantification (proofs depending on terms), and $\Longrightarrow$ for implication (proofs depending on proofs).

Types are formed as simple first-order structures, consisting of type variables α or type constructor applications $\kappa\ \tau_1 \ldots \tau_k$ (where κ has always k arguments).

Term syntax provides explicit abstraction $\lambda x :: \alpha.\ b(x)$ and application $t\ u$, while types are usually implicit thanks to type-inference; terms of type *prop* are called propositions. Logical statements are composed via $\bigwedge x :: \alpha.\ B(x)$ and $A \Longrightarrow B$. Primitive reasoning operates on judgments of the form $\Gamma \vdash \varphi$, with standard introduction and elimination rules for $\bigwedge$ and $\Longrightarrow$ that refer to fixed parameters x and hypotheses A from the context Γ. The corresponding proof terms are left implicit, although they could be exploited separately [3].

The framework also provides definitional equality $\equiv :: \alpha \Rightarrow \alpha \Rightarrow$ *prop*, with $\alpha\beta\eta$-conversion rules. The internal conjunction $\& ::$ *prop* $\Rightarrow$ *prop* $\Rightarrow$ *prop* allows to represent simultaneous statements with multiple conclusions.

Derivations are relative to a given theory Θ, which consists of declarations for type constructors κ (constructor name with number of arguments), term constants $c :: \sigma$ (constant name with most general type scheme), and axioms $\vdash \varphi$ (proposition being asserted). Theories are always closed by type-instantiation: arbitrary instances $c :: \tau$ of $c :: \sigma$ are well-formed; likewise for axiom schemes. Schematic polymorphism carries over to term formation and derivations, i.e. it is admissible to derive any type instance $\Gamma \vdash B(\tau)$ from $\Gamma \vdash B(\alpha)$, provided that α does not occur in the hypotheses of Γ.

3.2 Isar proof contexts

In judgments $\Gamma \vdash \varphi$ of the primitive framework, Γ essentially acts like a proof context. Isar elaborates this idea towards a higher-level notion, with separate information for

type-inference, term abbreviations, local facts, and generic hypotheses (parameterized by specific discharge rules). For example, the context element **assumes** A introduces a hypothesis with $\Longrightarrow$ introduction as discharge rule; **notes** $a = b$ defines local facts; **defines** $x \equiv a$ and **fixes** $x :: \alpha$ introduce local terms.

Top-level theorem statements may refer directly to Isar context elements to establish a conclusion within an enriched environment; the final result will be in discharged form. For example, proofs of $\bigwedge x.\ B\ x$, and $A \Longrightarrow B$, and $B\ a$ can be written as follows:

lemma	**lemma**	**lemma**
fixes x	**assumes** A	**defines** $x \equiv a$
shows $B\ x$ $\langle proof \rangle$	**shows** B $\langle proof \rangle$	**shows** $B\ x$ $\langle proof \rangle$

There are separate Isar commands to build contexts within a proof body, notably **fix**, **assume** etc. These elements have essentially the same effect, only that the result lives still within a local proof body rather than the target theory context. For example:

{	{	{
fix x	**assume** A	**def** $x \equiv a$
have $B\ x$ $\langle proof \rangle$	**have** B $\langle proof \rangle$	**have** $B\ x$ $\langle proof \rangle$
}	}	}

Building on top of structured proof contexts, the Isar proof engine now merely imposes a certain policy for interpreting formal texts, in order to support structured proof composition [21, Chapter 3]. The very same notion of contexts may be re-used a second time for structured theory specifications, namely by Isabelle locales (see below).

3.3 Locales

Isabelle locales [8, 1] provide a powerful mechanism for managing local proof context elements, most notably **fixes** and **assumes**. For example:

locale $l =$
  fixes x
  assumes $A\ x$

This defines both a predicate $l\ x \equiv A\ x$ (by abstracting the body of assumptions over the fixed parameters), and provides some internal infrastructure for structured reasoning. In particular, consequences of the locale specification may be proved at any time, e.g.:

lemma (**in** l)
  shows b: $B\ x$ $\langle proof \rangle$

The result b: $B\ x$ is available for further proofs within the same context. There is also a global version $l.b$: $\bigwedge x.\ l\ x \Longrightarrow B\ x$, with the context predicate being discharged.

Locale expressions provide means for high-level composition of complex proof contexts from basic principles (e.g. **locale** *ring* $=$ *abelian-group* R $+$ *monoid* R $+$ $\ldots$). Expressions e are formed inductively as $e = l$ (named locale), or $e = e'\ x_1 \ldots x_n$ (renaming of parameters), or $e = e_1 + e_2$ (merge). Locale merges result in general acyclic graphs of sections of context elements — internally, the locale mechanism produces a canonical order with implicitly shared sub-graphs.

Locale interpretation is a separate mechanism for applying locale expressions in the current theory or proof context [2]. After providing terms for the **fixes** and proving the **assumes**, the corresponding instances of locale facts become available. For example:

interpretation *m*: *l* [*a*]
proof (*rule l.intro*)
  show *A a* ⟨*proof*⟩
qed

Here the previous locale fact *l.b*: $\bigwedge x.\ l\ x \Longrightarrow B\ x$ becomes *m.b*: *B a*. The link between the interpreted context and the original locale acts like a dynamic subscription: any new results emerging within *l* will be automatically propagated to the theory context by means of the same interpretation. For example:

lemma (**in** *l*)
  shows *c*: *C x* ⟨*proof*⟩

This makes both *l.c*: $\bigwedge x.\ l\ x \Longrightarrow C\ x$ and *m.c*: *C a* available to the current theory.

4 Type classes and disciplined overloading

Starting from well-known concepts of order-sorted algebra, we recount the existing axiomatic type classes of Isabelle. Then we broaden the perspective towards explicit construction of dictionary terms, which explain disciplined overloading constructively.

4.1 An order-sorted algebra of types

The well-known concepts of order-sorted algebra (e.g. [17]) have been transferred early to the simply-typed framework of Isabelle [13, 10].

A type class *c* is an abstract entity that describes a collection of types. A sort *s* is a symbolic intersection of finitely many classes, written as expression $c_1 \cap \ldots \cap c_m$ (note that Isabelle uses the concrete syntax $\{c_1, \ldots, c_m\}$). We assume that type variables are decorated by explicit sort constraints α_s, while plain α refers to a vacuous constraint of the empty intersection of classes (the universal sort). An order-sorted algebra consists of a set *C* of classes, together with an acyclic subclass relation $<$, and a collection of type constructor arities $\kappa :: (s_1, \ldots, s_k)c$ (for constructor κ with *k* arguments). This induces an inductive relation $\tau : c$ on types τ and classes *c* by the rules given below (on sorts $\tau : c_1 \cap \ldots \cap c_m$ is defined as $\tau : c_i$ for all $i = 1, \ldots, m$ collectively).

$$\frac{\tau : c_1 \quad c_1 < c_2}{\tau : c_2}\ (\textit{classrel})$$

$$\frac{\tau_1 : s_1 \quad \ldots \quad \tau_k : s_k \quad \kappa :: (s_1, \ldots, s_k)c}{\kappa\ \tau_1\ \ldots\ \tau_k : c}\ (\textit{constructor})$$

$$\frac{}{\alpha_{c_1 \cap \ldots \cap c_m} : c_i}\ (\textit{variable})$$

We also define canonical subclass and subsort relations on top of this: $c_1 \subseteq c_2$ iff $\forall\, \alpha.\ \alpha : c_1 \Longrightarrow \alpha : c_2$ for classes, and $s_1 \subseteq s_2$ iff $\forall\, \alpha.\ \alpha : s_1 \Longrightarrow \alpha : s_2$ for sorts.

Observe that class inclusion $c_1 \subseteq c_2$ is the reflexive-transitive closure of the original relation $c_1 < c_2$. Proof: consequences $\alpha : c_1 \Longrightarrow \alpha : c_2$ emerge exactly by zero or more application of the *classrel* rule.

Moreover, sort inclusion $s_1 \subseteq s_2$ for $s_1 = c_1 \cap \ldots \cap c_m$ and $s_2 = d_1 \cap \ldots \cap d_n$ can be characterized as $\forall j.\ \exists i.\ c_i \subseteq d_j$. Proof: $c_1 \cap \ldots \cap c_m \subseteq d_1 \cap \ldots \cap d_n$ is equivalent to $\alpha : c_1 \cap \ldots \cap c_m \Longrightarrow \alpha : d_1 \cap \ldots \cap d_n$, i.e. $(\forall i.\ \alpha : c_i) \Longrightarrow (\forall j.\ \alpha : d_j)$, which is equivalent to $\forall j.\ \exists i.\ (\alpha : c_i \Longrightarrow \alpha : d_j)$.

An order-sorted algebra is called *coregular* iff for all $c \subseteq c'$, any $\kappa :: (s_1, \ldots, s_k)c$ and $\kappa :: (s'_1, \ldots, s'_k)c'$ have related argument sorts $\forall i.\ s_i \subseteq s'_i$. Coregularity expresses the key correspondence of the global class hierarchy with individual type constructor arities. This achieves most general unification and principal type schemes [17, 13].

4.2 Axiomatic type classes in Isabelle

Axiomatic type classes [10, 19] are based on a purely logical interpretation of the order-sorted algebra of types as predicates. Any closed proposition $\varphi(\alpha)$ depending on exactly one type variable can be understood as a predicate on types. The trick is to represent predicate constants adequately in order to support type class definitions and abstract reasoning over type classes. Following [19], any type class $c \in C$ of the underlying algebra is turned into a logical constant *c-class* $:: \alpha$ *itself* $\Rightarrow$ *prop*, where α *itself* is an uninterpreted type with constant *TYPE* $:: \alpha$ *itself* as canonical representative.[1] Propositions of the form *c-class* (*TYPE* $:: \tau$ *itself*) shall be written as $(\!|\tau : c|\!)$.

The existing **axclass** mechanism defines type classes via $\vdash (\!|\alpha : c|\!) \equiv (\!|\alpha : d_1|\!)\ \&\ \ldots\ \&\ (\!|\alpha : d_n|\!)\ \&\ A_1(\alpha)\ \&\ \ldots\ \&\ A_m(\alpha)$, where $d_1, \ldots, d_n$ are super-classes and $A_1, \ldots, A_m$ class axioms as intended by the user. From this the system derives an introduction rule $\vdash (\!|\alpha : d_1|\!) \Longrightarrow \ldots (\!|\alpha : d_n|\!) \Longrightarrow A_1(\alpha) \Longrightarrow \ldots A_m(\alpha) \Longrightarrow (\!|\alpha : c|\!)$ (for instantiation proofs), explicit class inclusions $\vdash (\!|\alpha : c|\!) \Longrightarrow (\!|\alpha : d_j|\!)$ (added to the order-sorted type algebra), and abstract lemmas $\vdash (\!|\alpha : c|\!) \Longrightarrow A_i(\alpha)$ (also called "class axioms"). The latter are represented compactly using sort constraints $\vdash A_i(\alpha_c)$. Isabelle inferences will use order-sorted type-unification in order to produce well-sorted instantiations $\vdash A_i(\tau)$ on the fly — this implicit reasoning is the main convenience of type classes.

It is easy to see that the interpretation of class membership $\tau : c$ as $(\!|\tau : c|\!)$ is correct in the sense that the notions of order-sorted type algebra approximate Pure derivations. In particular, the inference system for $\tau : c$ represents the following rules for $(\!|\tau : c|\!)$ (due to modus ponens and type instantiation):

$$\frac{(\!|\tau : c_1|\!) \quad \vdash (\!|\alpha : c_1|\!) \Longrightarrow (\!|\alpha : c_2|\!)}{(\!|\tau : c_2|\!)}$$

$$\frac{(\!|\tau_1 : s_1|\!) \quad \ldots \quad (\!|\tau_k : s_k|\!) \quad \vdash (\!|\alpha_1 : s_1|\!) \Longrightarrow \ldots (\!|\alpha_k : s_k|\!) \Longrightarrow (\!|\kappa\ \alpha_1\ \ldots\ \alpha_k : c|\!)}{(\!|\kappa\ \tau_1\ \ldots\ \tau_k|\!) : c}$$

$$\frac{}{(\!|\alpha : c_1|\!), \ldots, (\!|\alpha : c_m|\!) \vdash (\!|\alpha : c_i|\!)}$$

[1] This type could be defined explicitly as **datatype** α *itself* $=$ *TYPE* in ML / Haskell / HOL, but Isabelle/Pure refrains from stating any specific properties.

Here the rule conditions $c_1 < c_2$ and $\kappa :: (\overline{s})c$ have been interpreted by schematic implications, and sort constraints of type variables have been turned into explicit hypotheses.

The general principle above is to interpret the inductive definition of $\tau : c$, by giving a constructive reading to its derivations. Thus inferences taking place during internal type-checking operations are turned into proofs of the Pure framework.

In conclusion, we observe that axiomatic type classes are able to model the logical part (**assumes**) of our **class** mechanism. The second half is proper management of class operations (**fixes**) which will be based on a disciplined version of overloaded definitions.

4.3 Disciplined overloading for Isabelle

Simple definitions essentially introduce abbreviations in terms of basic principles, by stating definitional equalities within the formal theory. A definitional theory extension $\Theta' = \Theta \cup c :: \sigma \cup \vdash c_\sigma \equiv t$ is well-formed iff c is a fresh constant name, t is a closed term that does not mention c, and all type variables occurring in t also occur in σ.

The latter condition ensures $\vartheta(t) = \vartheta'(t) \Longrightarrow \vartheta(c_\sigma) = \vartheta'(c_\sigma)$ for arbitrary type instantiations ϑ and ϑ', i.e. there is a one-to-one relationship between the LHS and RHS. Due to substitution of $\equiv$, this means $\Gamma \vdash \varphi(\vartheta(c_\sigma))$ iff $\Gamma \vdash \varphi(\vartheta(t))$ in Θ'.

Moreover, $\Gamma \vdash \varphi(c)$ is derivable in Θ' iff $\Gamma \vdash \varphi(t)$ is derivable in Θ. Proof: (1) assume $\Gamma \vdash \varphi(c)$; hence $\Gamma \vdash \varphi(t)$ in Θ' by definition. Let $\psi = \varphi(t)$, which is a formula of Θ and theorem of Θ'. Show by induction over derivations that $\Gamma \cup \vartheta_1(c_\sigma \equiv t), \ldots, \vartheta_n(c_\sigma \equiv t) \vdash \psi$ in Θ, for some collection of type instantiations $\vartheta_1, \ldots, \vartheta_n$ (stemming from instances of the definitional axiom occurring in the proof trees). Finally discharge these assumptions by reflexivity of $\equiv$. (2) the other direction is trivial.

A definitional theory may be presented in an incremental fashion, where later definitions refer to previously defined entities on the RHS. For example, define c_1, then c_2 in terms of c_1, then c_3 in terms of c_1, c_2 etc. Formally, we introduce a dependency relation between constant names: $c \rightarrow b$ iff constant b is mentioned on the RHS of the definition of c. Provided that $\rightarrow$ is well-founded, incremental definitions can be normalized such that the RHSes only mention basic principles. Thus simple definitions determine an immediate mapping from defined entities to basic principles.

Overloading (or "ad-hoc polymorphism") means to specify constants depending on the syntactic structure of their respective type instances. For example, $0 :: \alpha$ could be defined separately for 0_{nat}, 0_{bool}, $0_{\beta \times \gamma}$ (in terms of 0_β and 0_γ) etc. Unrestricted overloading sacrifices most of the key syntactic properties sketched above.

Subsequently, we borrow some notation from System F [16], notably type schemes $\forall \alpha.\ \sigma(\alpha)$ and type application $f\ [\tau]$. For example, the polymorphic identity function $id \equiv \lambda x.\ x$ can be given the most general type scheme $\forall \alpha.\ \alpha \Rightarrow \alpha$. System F also provides explicit type abstraction $\Lambda \alpha.\ t(\alpha)$, although this will not be required here, because naive polymorphism in the Pure framework is restricted to outermost constants (and axioms): instead of $id \equiv \Lambda \alpha.\ \lambda x{::}\alpha.\ x$ we write $id\ [\alpha] \equiv \lambda x{::}\alpha.\ x$ in applied form.

This quasi-polymorphic perspective allows an adequate view on constant declarations and type instances as required for overloading. Any declaration $c :: \sigma$ can be turned into an explicit type scheme $c :: \forall\, \overline{\alpha}.\ \sigma(\overline{\alpha})$ by presenting the type variables $\overline{\alpha}$ of the body σ in some canonical order. Type instances can now be written as $c\ [\overline{\tau}]$, where $\overline{\tau}$ emerges by matching against σ and putting the RHSes of the resulting substitution $[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$ into the same canonical order.

Linear polymorphic declarations ($n = 1$) are an important special case of this. Here $c :: \forall\, \alpha.\ \sigma(\alpha)$ acts like a function that maps a type τ to a term $c\ [\tau]$ of type $\sigma(\tau)$.

Restricted overloading is a theory extension $\Theta \cup c :: \forall\, \alpha.\ \sigma(\alpha) \cup \vdash c\ [\tau] \equiv t \cup \ldots$ that introduces a fresh constant declaration c, followed by a collection of specifications $\vdash c\ [\tau] \equiv t$ each, where t is a closed term, and all type variables of t also occur in τ. The defining equations for c are further restricted to

$$c\ [\kappa\ \overline{\alpha}] \equiv \ldots\ b\ [\overline{\tau}]\ \ldots\ d\ [\alpha_i]\ \ldots$$

such that the type argument of the LHS is a constructor κ applied to distinct variables $\overline{\alpha}$, and the RHS (after normalizations with respect to simple definitions) only mentions further constants as follows:

1. arbitrary instances of constants named b, provided that $c \rightarrow b$ holds according to a given well-founded dependency relation on constant names;
2. argument projections on overloaded constants $d\ [\alpha_i]$, selecting some α_i from $\overline{\alpha}$.

Moreover the following global conditions have to be observed:

- There is at most one specification $c\ [\kappa\ \overline{\alpha}] \equiv \ldots$ for each type constructor κ.
- Overloaded specifications are upwards-complete: for any $c_1 \rightarrow^+ c_2$, the presence of $c_1\ [\kappa\ \overline{\alpha}] \equiv \ldots$ implies the presence of $c_2\ [\kappa\ \overline{\alpha}] \equiv \ldots$.

Note that the restriction of $c \rightarrow b$ is independent of actual type instances and essentially decouples general interdependencies from overloading. For example, the specification of $c\ [\mathit{nat}] \equiv \ldots\ b\ [\mathit{bool}]\ \ldots$ and $b\ [\mathit{nat}] \equiv \ldots\ c\ [\mathit{bool}]\ \ldots$ is ruled out, due to the cycle $c \rightarrow b \rightarrow c$ on constant names.

The following example illustrates restricted overloading of constants *eq* and *ord* for types *nat* and $\times$:

$$\begin{aligned}
&\mathit{eq} :: \forall\, \alpha.\ \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}\\
&\mathit{eq}\ [\mathit{nat}] \equiv \lambda m\ n.\ m = n\\
&\mathit{eq}\ [\beta \times \gamma] \equiv \lambda p\ q.\ \mathit{eq}\ [\beta]\ (\mathit{fst}\ p)\ (\mathit{fst}\ q) \wedge \mathit{eq}\ [\gamma]\ (\mathit{snd}\ p)\ (\mathit{snd}\ q)\\
&\mathit{ord} :: \forall\, \alpha.\ \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}\\
&\mathit{ord}\ [\mathit{nat}] \equiv \lambda m\ n.\ m < n\\
&\mathit{ord}\ [\beta \times \gamma] \equiv \lambda p\ q.\ \mathit{ord}\ [\beta]\ (\mathit{fst}\ p)\ (\mathit{fst}\ q) \vee\\
&\quad \mathit{eq}\ [\beta]\ (\mathit{fst}\ p)\ (\mathit{fst}\ q) \wedge \mathit{ord}\ [\gamma]\ (\mathit{snd}\ p)\ (\mathit{snd}\ q)
\end{aligned}$$

In general, restricted overloading and simple definitions may be presented incrementally, with alternating dependencies of overloaded vs. non-overloaded constants.

The resulting theory still describes a mapping from defined entities to basic principles — as sketched before for simple definitions alone. The key idea is to traverse the system along the lexicographic product of the global dependency relation $c \to b$ and the substructural order on types $\kappa\ \overline{\alpha} \to \alpha_i$, which is also well-founded.

Overloading as order-sorted type-algebra is a slightly more abstract view on the structure of interdependent overloaded specifications. After expanding all simple (non-overloaded) definitions, the resulting algebra of overloading is achieved as follows.

Classes: Each overloaded operation is turned into a type class of the same name.[2]
Class relation: The global dependency relation $\to$ is restricted to overloaded constants, i.e. $c_1 < c_2$ iff $c_1 \to^+ c_2$ on classes.
Constructor arities: The local dependencies of definitional equations are turned into constructor arities, i.e. $\kappa :: (s_1, \ldots, s_k)c$ for each constructor κ and class c, where $s_i = \bigcap d$ such that $d[\alpha_i]$ occurs on the RHS of some specification $c'\ [\kappa\ \alpha_1\ \ldots\ \alpha_k] \equiv \ldots$ for some $c' \supseteq c$.

Observe that this algebra is coregular by construction, because the argument sorts of type arities account for the upwards-completion of definitions explicitly.

For example, the previous overloaded definitions of *eq* and *ord* result in the algebra consisting of classes $\mathit{ord} < \mathit{eq}$ with constructor arities $\mathit{nat} :: \mathit{eq}$, and $\times :: (\mathit{eq}, \mathit{eq})\mathit{eq}$, and $\mathit{nat} :: \mathit{ord}$, and $\times :: (\mathit{eq} \cap \mathit{ord}, \mathit{eq} \cap \mathit{ord})\mathit{ord}$.

We now employ the order-sorted algebra to expand disciplined overloading: for any $\vdash \varphi$ mentioning well-defined instances $c\ [\tau]$ of overloaded constants, we produce $\vdash \varphi'$ that refers only to basic principles. In the first stage, we normalize by all definitional equalities, which removes non-overloaded constants and reduces overloaded ones to occurrences c_α on type variables. In the second stage we construct dictionary terms.

A *dictionary* δ for class c is a collection of terms $[t_1, \ldots, t_n]$ that provide implementations for the class operations $[c_1, \ldots, c_n]$, for the collection of classes $c' \supseteq c$ presented in canonical order. The construction works by interpreting the derivation of $\tau : c$ for each $c\ [\tau]$ occurring in $\vdash \varphi$. The base case refers to locally fixed dictionary parameters $p^c :: \sigma(\alpha)$ for each $c\ [\alpha] :: \sigma(\alpha)$ in $\vdash \varphi$. The type constructor case refers to the collection $\overline{\varrho}$ of RHSes of all specifications $c'\ [\kappa\ \overline{\alpha} \equiv \ldots]$ for $c' \supseteq c$, as in the construction of type arities $\kappa :: (\overline{s})c$ above. The notation $\{\!|\delta : c|\!\}$ means that δ contains a dictionary term for c. We now get the following rules:

$$\frac{\{\!|\delta : c_1|\!\} \quad c_1 < c_2}{\{\!|\delta : c_2|\!\}}\ (\mathit{classrel})$$

$$\frac{\{\!|\delta_1 : s_1|\!\} \quad \ldots \quad \{\!|\delta_k : s_k|\!\} \quad \kappa :: (s_1, \ldots, s_k)c}{\{\!|\overline{\varrho}(\delta_1, \ldots, \delta_k) : c|\!\}}\ (\mathit{constructor})$$

$$\frac{}{\{\!|[p^{c_1}, \ldots, p^{c_m}] : c_i|\!\}}\ (\mathit{variable})$$

[2] We essentially assume that each type class corresponds to exactly one operation of the same name. Minor re-formulations will admit the more liberal scheme seen in practice (e.g. §2).

For example, $\vdash P\ (ord\ [\beta \times \gamma])$ can be expanded to $\vdash P\ (\lambda p\ q.\ ord_1\ (fst\ p)\ (fst\ q) \vee eq_1\ (fst\ p)\ (fst\ q) \wedge ord_2\ (snd\ p)\ (snd\ q))$, for new local variables eq_1, eq_2, ord_1, ord_2.

We see that disciplined overloading can be linked to the order-sorted type-algebra quite naturally. The key benefit is that well-definedness of $c\ [\tau]$ is reduced to well-sortedness $\tau : c$, while a constructive reading provides the dictionary expansion.

Thus we have managed to make "ad-hoc polymorphism less ad-hoc", although by quite different means than the original Haskell type class system [18]. In more general versions of type theory, the reconstruction of dictionary terms (for the operations) and proof terms (for the logical part) would have coincided anyway, but Pure has two distinctive categories of formal entities that appear to the user as **fixes** and **assumes**.

5 Integration

We are ready to integrate the concepts of §3 and §4 to explain our version of **class** and **instance**. Essentially, we shall introduce (I) a locale that manages both the **fixes** and **assumes** explicitly, (II) type class infrastructure that replaces the **fixes** by global operations according to disciplined overloading, and (III) a formal link between the locale and type class by locale interpretation. We illustrate this by the example from §2.

5.1 Class definition

(I) The syntax for **class** specifications is the same as for **locale**, restricted to exactly one type variable α. Thus a **class** is literally made a **locale** of the same name. E.g.

locale *semigroup* =
  fixes *mult* :: $\alpha \Rightarrow \alpha \Rightarrow \alpha$ (**infix** $\circ$ 70)
  assumes *assoc*: $(x \circ y) \circ z = x \circ (y \circ z)$

(II) The same specification is turned into type class infrastructure as follows.

1. For all class operations (**fixes**) introduce global operations (**consts**) with the same name and type. E.g.

   consts
     mult :: $\alpha \Rightarrow \alpha \Rightarrow \alpha$ (**infix** $\circ$ 70)

2. Introduce an *axiomatic type class* whose axioms are the class premises (**assumes**), applied to the newly introduced **consts**. Since a locale definition already defines a predicate corresponding to the body, we can use a compact representation. E.g.

   axclass *semigroup*
     axiom: *semigroup* (*mult* :: $\alpha \Rightarrow \alpha \Rightarrow \alpha$)

3. Restrict subsequent uses of the global operations to the new type class. E.g.

   constraints
     mult :: α::*semigroup* $\Rightarrow \alpha \Rightarrow \alpha$

   This is merely an extra-logical hint for type-inference, which ensures that occurrences of the operations will be well-defined.

(III) Finally link the locale and type class infrastructure by means of locale interpretation: the global operations (**consts**) are inserted for the local ones (**fixes**), and the (already derived) class axiom is inserted for the locale premises (**assumes**). E.g.

interpretation *semigroup* [*mult* :: α::*semigroup* $\Rightarrow \alpha \Rightarrow \alpha$]
  by (*rule semigroup-class.axiom*)

This reduces the generality of locale results by fixing the operations, but α remains free.

5.2 Class instantiation

An instance provides term definitions and proofs on particular type patterns $\kappa\ \overline{\alpha}$. The class operations are introduced by the existing primitive for overloaded definitions, which is only used in the restricted sense of §4.3. E.g.

defs (**overloaded**)
  mult-int-def: (*i*::*int*) $\circ$ *j* $\equiv$ *i* + *j*

The actual instance proof uses the original **axclass** instantiation mechanism. E.g.

instance *int* :: *semigroup* — (existing version of axclass **instance**)
proof
  fix *i j k* :: *int* **have** $(i + j) + k = i + (j + k)$ **by** *simp*
  then show $(i \circ j) \circ k = i \circ (j \circ k)$ **unfolding** *mult-int-def* **.**
qed

5.3 Subclasses

(I) In order to derive a new class *c* from existing super-classes $b_1, \ldots, b_n$ we simply produce parallel hierarchies of locales and type classes. For locales this means to import the merge $b_1 + \ldots + b_n$ of the corresponding parent locales. E.g.

locale *group* = *semigroup* +
  fixes *neutral* :: α ($\mathbf{1}$)
    and *inverse* :: $\alpha \Rightarrow \alpha$ (($\text{-}^{-\mathbf{1}}$) [1000] 999)
  assumes *left-neutral*: $\mathbf{1} \circ x = x$
    and *left-inverse*: $x^{-\mathbf{1}} \circ x = \mathbf{1}$

(II) The type class setup is analogous; **axclass** treats super-classes as expected. E.g.

consts
  neutral :: α ($\mathbf{1}$)
  inverse :: $\alpha \Rightarrow \alpha$ ((-^{-1}) [1000] 999)

axclass *group* < *semigroup*
  axiom: *group mult neutral inverse*

constraints
  neutral :: α::*group*
  inverse :: α::*group* $\Rightarrow \alpha$

(III) The link between locale and class definition is again by interpretation. The implicit import of results established in parent locales [2] works without further ado. E.g.

interpretation
group [*mult* :: α::*group* $\Rightarrow \alpha \Rightarrow \alpha$ *neutral* :: α::*group* *inverse* :: α::*group* $\Rightarrow \alpha$]
by (*rule group-class.axiom*)

5.4 Abstract reasoning

Nothing special needs to be done here — we benefit directly from the existing mechanisms of locale lemmas. E.g. "**lemma** (*in group*) ..." refers to the target locale *group*, even if this happens to be related to a type class of the same name. Abstract reasoning is performed in full generality at the locale level relative to **fixes** and **assumes**.

6 Conclusion

Stocktaking. The present approach to constructive type classes in Isabelle integrates a fair amount of existing concepts into a coherent mechanism for the end-user, without having to extend the underlying logical foundations. Apart from collecting existing concepts, our main contribution is twofold: (1) explicit reconstruction of proofs and dictionary terms, guided by constructive interpretation of order-sorted type algebras, (2) relating **locale** and **class** concepts by means of interpretation.

The first aspect has foundational impact, the formal content of type classes is explained more thoroughly in terms of basic principles. Moreover, applications that build on the internal representations of theories and proofs may benefit from this additional structure (e.g. code generation for ML or proof export for other versions of HOL).

The second aspect is very important for user-level reasoning with type classes within the formal system. Our link to the locale mechanism [8, 1, 2] overcomes the former restriction of axiomatic type-classes to a fixed "signature" of overloaded constants. Our classes admit abstract reasoning in the general locale context, where operations are local parameters; results are implicitly passed down to the actual type class thanks to locale interpretation. Thus we essentially combine the best of both worlds.

Even more, several type classes can be linked to the same locale, using the additional **includes** element to refer to a renamed locale specification: e.g. **class** *abelian-group* = **includes** *group add* (**infix** + 60) **assumes** *commute*: ... etc. General lemmas established in *group* will then become available for both type classes *group* and *abelian-group*.

The present work has resulted in clarification of various Isabelle internals[3]. In particular, the constructive interpretation of order-sorted type-algebra is now explicit in the internal workings of **axclass**, so far some justifications have been only on paper [19]. There is now also a separation of constant declarations $c :: \forall\, \alpha.\ \sigma(\alpha)$, and extra-logical type-inference constraints $c :: \forall\, \alpha{::}c.\ \sigma(\alpha)$.

[3] See `http://isabelle.in.tum.de/devel/` for a development snapshot.

Related work. Module systems (especially for theorem provers) provide a more general perspective on our work. Roughly speaking, the huge amount of existing approaches can be categorized as follows: (1) full / explicit module languages vs. (2) restricted / implicit structuring mechanisms. ML functors [16] and Coq modules [4, 5] represent the first kind, type classes in Haskell or Isabelle the second, more light-weight one. Our work helps to bridge the gap between these two extremes, by enhancing the basic type class concepts towards a more explicit notion of modules, thanks to the underlying locale infrastructure.

Compared to a full-grown module system, locales do have some limitations: no truly polymorphic parameters, no type-constructors as parameters. For example, a theory of monads would be hard to formalize. However, explaining locales (and classes) in terms of existing Isabelle/Pure concepts avoids tinkering with the logic itself.

Type classes have first appeared in Haskell [18, 15, 6], to make "ad-hoc polymorphism less ad-hoc". The underlying ideas have later been rephrased as a problem of Hindley-Milner type-checking within an order-sorted algebra of types [13], and integrated into the Isabelle/Pure type-checker [10]. Isabelle type classes acquired their first logical interpretation in [19]. Note that more recent extensions of the original Haskell type classes (including *constructor classes* and *multi-parameter classes* [7]) are not covered in this work, mostly due to fundamental limitations of the underlying logic.

Future work. Our constructive combination of type classes and locales essentially organizes lemmas (proofs) that emerge in related contexts. This principle could be transferred to derived operations (terms). Recent experiments on "**definition** (*in l*)" for locales could be generalized to handle classes as well, by producing parallel definitions internally that refer either to locale parameters (**fixes**) or overloaded operations (**consts**).

Further considerations need to be spent on **instance** definitions. So far this is limited to simple definitions of Pure, but realistic applications demand more flexibility. The key question is how to combine derived definitional mechanisms with class instantiations in a modular fashion, without hardwiring one into the other. Then a package like [9] for general recursive functions could be used to specify class operations.

Acknowledgment. Alexander Krauss and Tobias Nipkow have commented on draft versions of this paper.

References

[1] C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. Berardi et al., editors, *Types for Proofs and Programs (TYPES 2003)*, LNCS 3085, 2004.

[2] C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. Borwein and W. Farmer, editors, *Mathematical Knowledge Management (MKM 2006)*, LNAI 4108, 2006.

[3] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2000)*, LNCS 1869, 2000.

[4] J. Chrzaszcz. *Modules in type theory with generative definitions*. PhD thesis, Université Paris-Sud, 2004.

[5] J. Courant. $\mathcal{MC}_2$: A Module Calculus for Pure Type Systems. *The Journal of Functional Programming*, 2006. To appear.

[6] C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2), 1996.

[7] S. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space, 1997.

[8] F. Kammüller, M. Wenzel, and L. C. Paulson. Locales: A sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics (TPHOLs '99)*, LNCS 1690, 1999.

[9] A. Krauss. Partial recursive functions in higher-order logic. In U. Furbach and N. Shankar, editors, *Int. Joint Conference on Automated Reasoning (IJCAR 2006)*, LNCS, 2006.

[10] T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical Environments*. Cambridge University Press, 1993.

[11] T. Nipkow. Structured proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, LNCS 2646, 2003.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*. LNCS 2283. 2002.

[13] T. Nipkow and C. Prehofer. Type checking type classes. In *ACM Symp. Principles of Programming Languages*, 1993.

[14] L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[15] J. Peterson and M. P. Jones. Implementing type classes. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

[16] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[17] M. Schmidt-Schauß. *Computational aspects of an order-sorted logic with term declarations*. LNAI 395. 1989.

[18] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symp. Principles of Programming Languages*, 1989.

[19] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics (TPHOLs '97)*, LNCS 1275, 1997.

[20] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, LNCS 1690, 1999.

[21] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, TU München, 2002.