COMP 4161

NICTA Advanced Course

Advanced Topics in Software Verification

Toby Murray, June Andronick, Gerwin Klein

type classes & locales

Content

➜ Intro & motivation, getting started [1]

➜ Foundations & Principles

- Lambda Calculus, natural deduction [1,2]
- Higher Order Logic [3[a]]
- Term rewriting [4]

➜ Proof & Specification Techniques

- Inductively defined sets, rule induction [5]
- Datatypes, recursion, induction [6, 7]
- Hoare logic, proofs about programs, C verification [8[b],9]
- (mid-semester break)
- Writing Automated Proof Methods [10]
- Isar, codegen, typeclasses, locales [11[c],12]

[a] a1 due; [b] a2 due; [c] a3 due

Type Classes

Common pattern in Mathematics:

- ➜ Define abstract structures (semigroup, group, ring, field, etc)
- ➜ Study and derive properties in these structures
- ➜ Instantiate to concrete structure: (nats with + and * from a ring)
- ➜ Can use all abstract laws for concrete structure

Type classes in functional languages:

- ➜ Declare a set of functions with signatures (e.g. plus, zero)
- ➜ give them a name (e.g. c)
- ➜ Have syntax ’a :: c for: type ’a supports the operations of c
- ➜ Can write abstract polymorphic functions that use plus and zero
- ➜ Can instantiate specific types like nat to c

Isabelle supports both.

Type Class Example

Example:

class semigroup =

fixes mult :: 'a $\Rightarrow$ 'a $\Rightarrow$ 'a (**infix** $\cdot$ 70)

assumes assoc: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

Declares:

- ➜ a name (semigroup)
- ➜ a set of operations (fixes mult)
- ➜ a set of properties/axioms (assumes assoc)

Type Class Use

Can constrain type variables 'a with a class:

definition sq :: ('a :: semigroup) $\Rightarrow$ 'a **where** sq x $\equiv$ x · x

More than one constraint allowed. Sets of class constraints are called **sort**.

Can reason abstractly:

lemma "sq x · sq x = x · x · x · x"

Can instantiate:

instantiation nat :: semigroup

begin

definition "(x::nat) · y = x * y"

instance $< proof >$

end

DEMO: TYPE CLASSES

Type constructors

Basic type instantiation is a special case.

In general:

Type constructors can be seen as functions from classes to classes.

Example:

product type   prod :: (semigroup, semigroup) semigroup

(or: pairs of semigroup elements again form a semigroup)

Declarations such as *(semigroup, semigroup) semigroup* are called **arities**.

Fully integrated with automatic type inference.

Subclasses

Type classes can be extended:

class rmonoid = semigroup +

fixes one :: 'a

assumes x · one = x

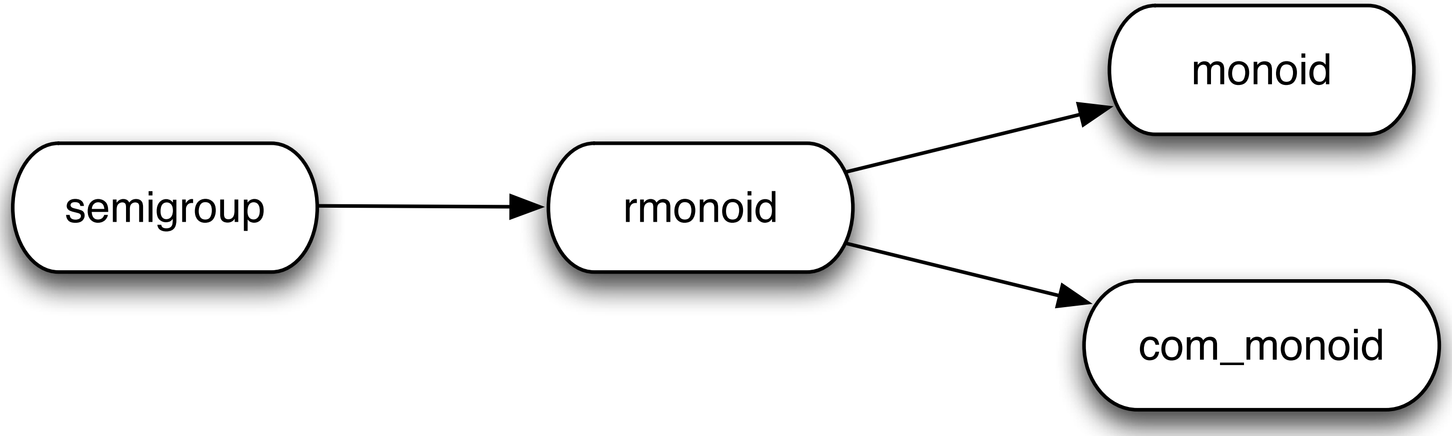
rmonoid is a **subclass** of semigroup

Has all operations & assumptions of semigroup + additional ones.

Can build hierarchies of abstract structures.

More Subclasses

Example structure:

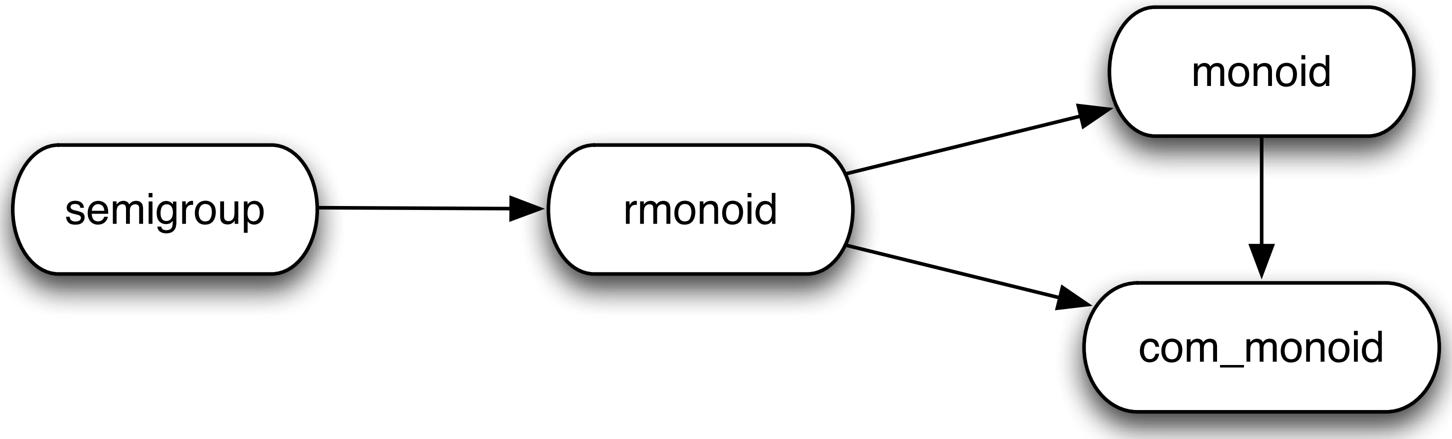


Can prove: every com_monoid is also a monoid.

Can tell Isabelle that connection:

subclass (in com_monoid) monoid $< proof >$

Result:



Limitations

Operations (fixes) are implemented by overloading

- ➜ each type constructor can implement each operation only once
- ➜ semigroup must be instantiated to addition or multiplication, not both

Type inference must remain automatic, with unique most general types

- ➜ type classes can mention only one type variable
- ➜ type constructor arities must be co-regular:

  $K :: (c_1, ..., c_n)c$ and $K :: (c'_1, ..., c'_n)c'$ and $c \subseteq c' \implies \forall i.\ c_i \subseteq c'_i$

Demo: Subclasses

From Types to Logic

NICTA

Type classes use the type system to store facts.

lemma	**lemma**
fixes $x :: \alpha :: \text{rmonoid}$	**fixes** $x :: \alpha$
	assumes $\text{OFCLASS}(\alpha, \text{rmonoid})$
shows $x \cdot one \cdot y = c \cdot y$	**shows** $x \cdot one \cdot y = c \cdot y$

The type system allows us to manage type assertions **implicitly**.

What if we could implicitly manage a **lemma**? We get **locales**.

Declaring Locales

Declaring **locale** (named context) loc:

locale loc =	
$loc1$ +	Import other locales
fixes . . .	variables
assumes . . .	facts

The **fixes** and **assumes** taken together are called context elements.

Declaring Locales

Theorems may be stated relative to a named locale.

lemma (in loc**)** P [simp]: $proposition$

$proof$

or

context loc **begin**

lemma P [simp]: $proposition$

$proof$

end

- ➜ Adds theorem P to context loc.
- ➜ Theorem P is in the simpset in context loc.
- ➜ Exported theorem $loc.P$ visible in the entire theory.

Isar Is Based On Contexts

Structured proofs (Isar) have some similar properties to locales.

theorem $\bigwedge x.\ A \Longrightarrow C$

proof -

fix x

assume Ass: A

$\vdots$

from Ass **show** C ...

qed

x and Ass are visible inside this context

Beyond Isar Contexts

Locales are extended contexts, look similar to type classes

- ➜ Locales are **named**
- ➜ Fixed variables may have **syntax**
- ➜ Locale may be entered and exited repeatedly
- ➜ It is possible to **add** and **export** theorems
- ➜ It is possible to **instantiate** locales
- ➜ Locale expression: **combine** and **modify** locales
- ➜ No limitation on type variables
- ➜ Term level, not type level: no automatic inference

Context Elements

Locales consist of **context elements**.

fixes	Parameter, with syntax
assumes	Assumption
defines	Definition
notes	Record a theorem

Demo: Locales 1

Parameters Must Be Consistent!

- ➜ Parameters in **fixes** are distinct.
- ➜ Free variables in **defines** occur in preceding **fixes**.
- ➜ Defined parameters cannot occur in preceding **assumes** nor **defines**.

Locale Expressions

Locale name: n

Rename: $n: \; e \; q_1 \ldots q_n$

Change names of parameters in e,

Give new locale the name prefix n (optional)

Merge: $e_1 + e_2$

Context elements of e_1, then e_2.

DEMO: LOCALES 2

Normal Form of Locale Expressions

NICTA

Locale expressions are converted to flattened lists of locale names.

- ➜ With full parameter lists
- ➜ **Duplicates removed**

Allows for **multiple inheritance**!

Instantiation

Move from **abstract** to **concrete**.

interpretation label: loc ”parameter 1” . . . ”parameter n”

- ➜ Instantiates locale **loc** with provided parameters.
- ➜ Imports all theorems of **loc** into current context.
  - Instantiates theorems with provided parameters.
  - Interprets attributes of theorems.
  - Prefixes theorem names with **label**
- ➜ version for local Isar proof: **interpret**

Sublocales

Similar to type classes:

sublocale (in sub_loc) parent_loc $< proof >$

makes facts of parent_loc available in sub_loc.

DEMO: LOCALES 3

We have seen today ...

- ➜ Type Classes + Instantiation
- ➜ Locale Declarations + Theorems in Locales
- ➜ Locale Expressions + Inheritance
- ➜ Locale Instantiation