type classes & locales

Last Time

- more C verification
- preventing undefined execution
- finite machine words
- concrete C data types
- C memory model and pointers

Slide 1

Content

- Intro & motivation, getting started
  - Rough timeline
  
- Foundations & Principles
  - Lambda Calculus, natural deduction
  - Higher Order Logic
  - Term rewriting

- Proof & Specification Techniques
  - Isar
  - Inductively defined sets, rule induction
  - Datatypes, recursion, induction
  - Calculational reasoning, mathematics style proofs
  - Hoare logic, proofs about programs

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.

Slide 3
Type Class Example

Example:

```plaintext
class semigroup = 
  fixes mult :: 'a ⇒ 'a ⇒ 'a (infix · 70)
  assumes assoc: (x · y) · z = x · (y · 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:

```plaintext
definition sq :: ('a :: semigroup) ⇒ 'a where sq x ≡ x · x
```

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

Can reason abstractly:

```plaintext
lemma "sq x · sq x = x · x · x · x · x"
```

Can instantiate:

```plaintext
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:

```plaintext
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:

```plaintext
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.

Result

Result:

```
semigroup  monoid  com_monoid
```

More Subclasses

Example structure:

```
semigroup  rmonoid  com_monoid
monoid
```

Can prove: every com_monoid is also a monoid.
Can tell Isabelle that connection:

```
subclass (in com_monoid) monoid < proof >
```

Limitations

Operations (fixes) are implemented by overloading

- each type constructor can implement each operation only once

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:
  \[ k : (c_1, \ldots, c_n) \quad \text{and} \quad k' : (c'_1, \ldots, c'_n) \quad \text{and} \quad c_i \subseteq c'_i \quad \rightarrow \quad \forall i. c_i \subseteq c'_i \]
DEMO: SUBCLASSES

Isar Is Based On Contexts

theorem $\forall x. A \Rightarrow C$
proof
  fix $x$
  assume $Ass: A$
  : $x$ and $Ass$ are visible
  from $Ass$ show $C$ ...
qed

Beyond Isar Contexts

Locales are extended contexts, look similar to type classes
  ➔ Locales are named
  ➔ Fixed variables may have syntax
  ➔ 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
Declaring Locales

Declaring locale (named context) loc:

```plaintext
locale loc = 
  locale1 + 
  Import 
  fixes ... Context elements 
  assumes ...
```

Slide 17

Theorems may be stated relative to a named locale.

```plaintext
lemma (in loc) P [simp]: proposition 
  proof
```

or

```plaintext
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.

Slide 18

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.

Slide 19

Slide 20
Locale Expressions

Locale name: n
Rename: e_1, ..., e_n
Change names of parameters in e_i.
Give new locale the name prefix n (optional)
Merge: e_1 + e_2
Context elements of e_1, then e_2.

Normal Form of Locale Expressions

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

Demo: Locales 2
Sublocales

Similar to type classes:

\texttt{sublocale} (in \texttt{sub_loc}) \texttt{parent_loc} < proof >

makes facts of \texttt{parent_loc} available in \texttt{sub_loc}.

We have seen today ...

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

DEMO: LOCALES 3

Slide 26