Formalizing the Theory Concept in Nuprl

Jason Hickey
March 15, 1994

Outline

- Goals of formalization
- Mechanism
- Problems along the way
- Final result

Goals of formalization

Majority vote—multiset

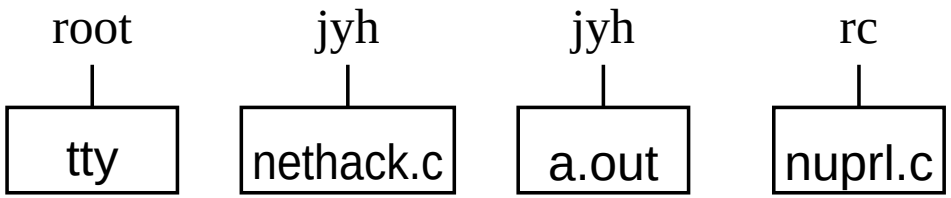
- Abstract Data Type
  - Axioms
  - Theorems
  - Definitions (abstractions)
- Problems with dependent product type
  - rigid structure
  - fixed number of axioms
  - difficult access
  - extensions are hard to make

Goal generalization

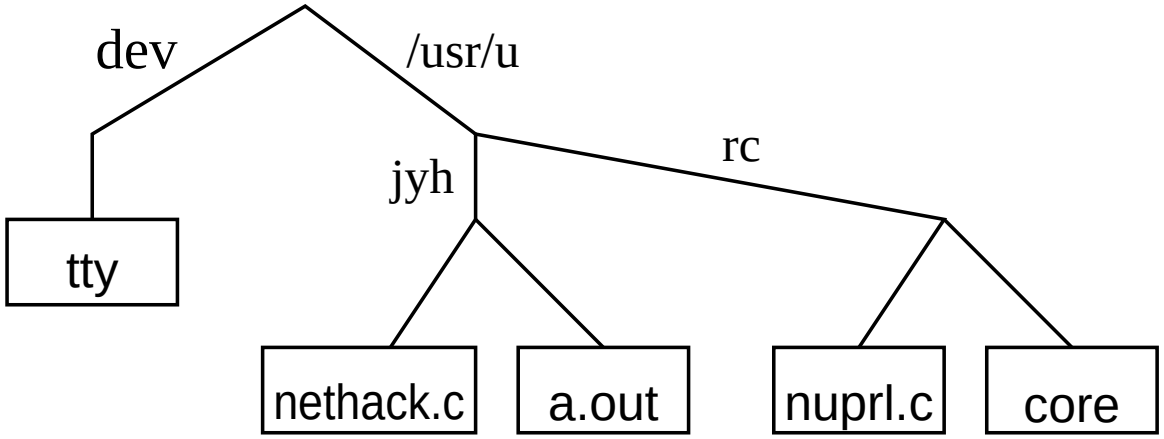
- Operate within a context of Theorems, Axioms, Definitions
- Primary goal is to extend the context
- Problems
  - flat name space
  - unrestricted access
  - general form of inheritance
  - variations on a theme
  - reflection

Flat name space

Flat namespace



Hierarchy

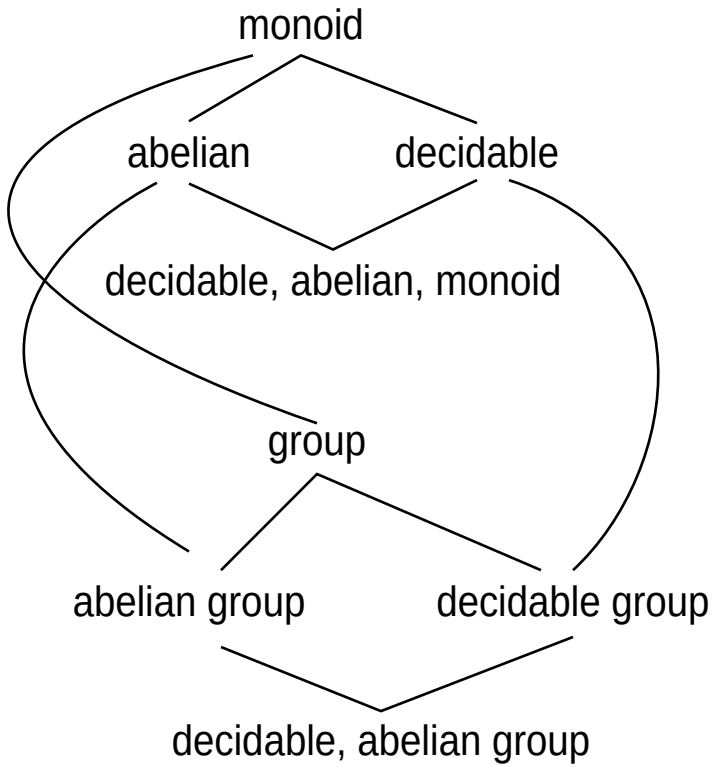


Unrestricted Access

- Utility theorems/functions
-
```
(*
 * Quick sort.
 *)
fun qsort lessp l =
    let fun sort [] = []
          | sort [x] = [x]
          | sort (a::bs) =
            (* The head a is the pivot *)
            let fun partition (left, right, []) =
                    (sort left) @ (a :: sort right)
                  | partition (left, right, x::xs) =
                    if lessp x a then
                        partition (x::left, right, xs)
                    else
                        partition (left, x::right, xs)
            in
                partition([], [], bs)
            end
    in
            sort l
    end;
```

Inheritance

Running example



Reflection

- Want to reason about
  - proofs
  - theories
  - tactics

Object Orientation

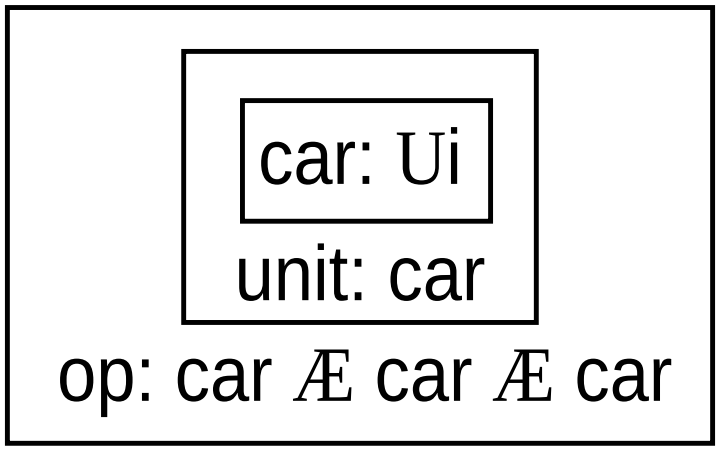
- Abstract Data Typing
- Inheritance
- Protection

Dependent Product

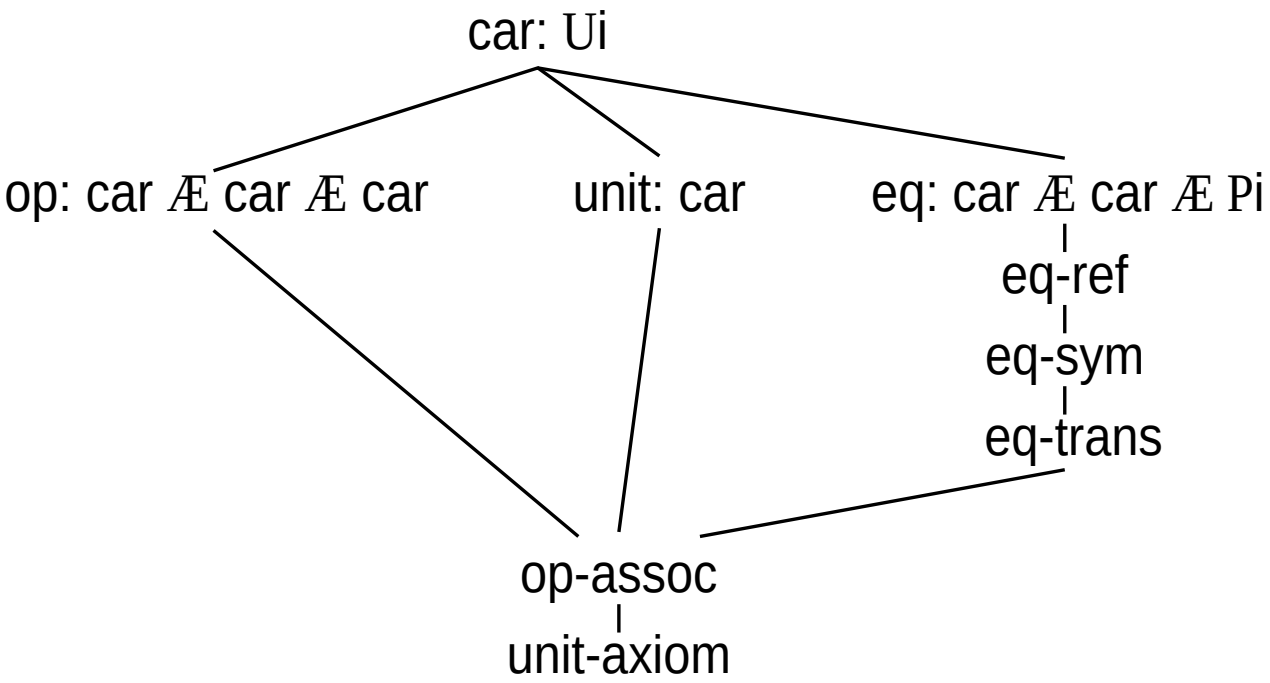
```
* ABS monoid
Monoid{i} == car:U{i}
             unit:car
           × op:(car → car → car)
           × eq:(car → car → P{i})
           × eq-ref:(∀a:car. a eq a)
           × eq-sym:(∀a:car. ∀b:car. a eq b ⇒ b eq a)
           × eq-trans:(∀a:car. ∀b:car. ∀c:car. a eq b ⇒ b eq c ⇒ a eq c)
           × unit-axiom:((∀a:car. (a op 1) eq a) ∧ (∀a:car. (1 op a) eq a))
           × (∀a:car. ∀b:car. ∀c:car. ((a op b) op c) eq (a op (b op c)))
```

Generalize the dependent product

- Number of elements is arbitrary
- Order of elements is not important
- Theory will be a DAG, together with a list of theories it depends on



Least constraining ordering



TheoryItem

- Reference this axiom by name
- Use inhabitants of previous axioms by looking through ancestors
- What is an axiom?

```
* ABS theory
Theory{i} == rec(Theory.name:IDList
                 × label:Atom
                 × flags:List of Z
                 × lib:List of TheoryLibType
                 × preds:List of Theory
                 × Axiom?
```

Axiom type

- Define lookup function
  - Lookup(Type) name in preds
  - Axiom: (name: ID →Lookup(Type) name in preds)
    →Ui
- Need inhabitant
  - TheoryType(preds)
  - Axiom: parents: TheoryType(preds)
    →(name:ID →Lookup(Type) name in parents using preds)
    →Ui

Example

- MonoidCar: <"car", [ ], lparents, lookup. Ui>
- MonoidUnit:

  <"unit",

  [MonoidCar],

  λparents, lookup.

  Lookup "car" in parents using MonoidCar>

Real Example

```
* ABS monoid car
MonoidCar{i} == {car}(U{i})

* ABS monoid car thy
MonoidCarThy{i} == Axiom:
                           Name = car/ ,
                           Flags = CNil,
                           Lib = CNil,
                           Preds = CNil,
                           Axiom(par, pre) = MonoidCar{i}
```

MonoidUnit

```
* ABS monoid unit preds
MonoidUnitPreds{i} == MonoidCarThy{i}::CNil

* ABS monoid unit
MonoidUnit(parents, preds) == TheoryEnv[parents, preds]
                                  car = car
                              in{unit}
                                  car

* ABS monoid unit thy
MonoidUnitThy{i} == Axiom:
                          Name = unit/ ,
                          Flags = CNil,
                          Lib = CNil,
                          Preds = MonoidUnitPreds{i},
                          Axiom(par, pre) = MonoidUnit(par, pre)
```

MonoidOpAssoc

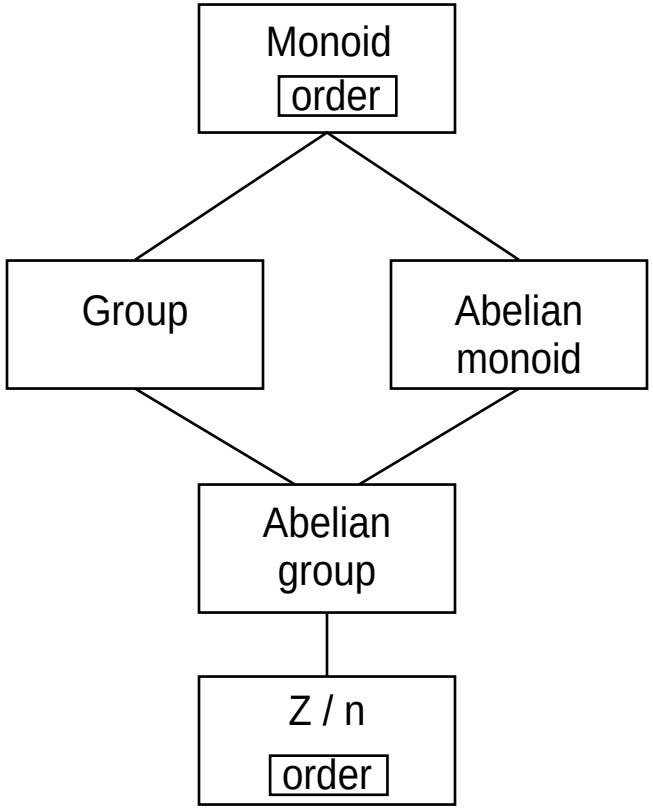
```
* ABS monoid unit preds
MonoidUnitPreds{i} == MonoidCarThy{i}::CNil

* ABS monoid unit
MonoidUnit(parents, preds) == TheoryEnv[parents, preds]
                                  car = car
                              in{unit}
                                  car

* ABS monoid unit thy
MonoidUnitThy{i} == Axiom:
                          Name = unit/ ,
                          Flags = CNil,
                          Lib = CNil,
                          Preds = MonoidUnitPreds{i},
                          Axiom(par, pre) = MonoidUnit(par, pre)
```

Theorems

- Axiom: "every monoid has an order"
- Theorem extract: function that computes the order
- May want several proofs of the theorem depending on the particular monoid



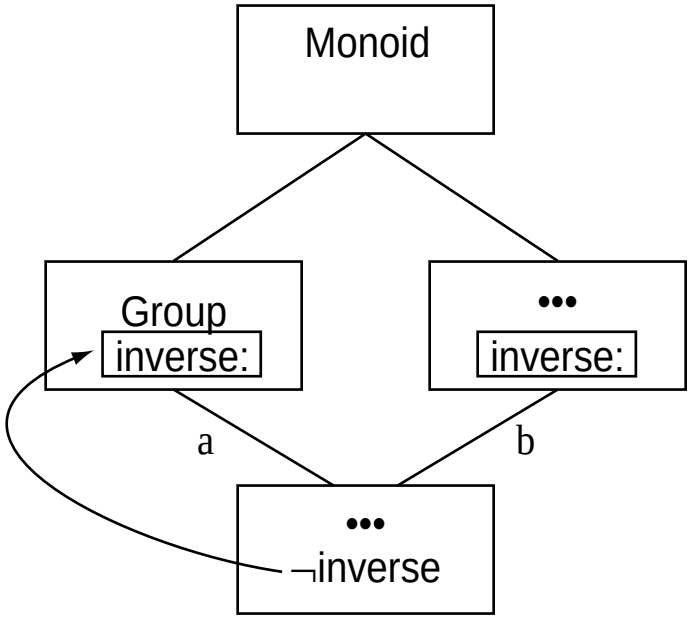
Why do we want Theorems?

- To show something is true
  - "Every monoid of prime order is cyclic"
- To compute
  - "Every monoid has an order"
- Don't handle extracts!
- A Theorem is just a link from an axiom to an object in the library
- Later: prove (∃ theory: TheoryType(...))
- Abstractions and other objects are just links to library objects

Naming considerations

Can pull in any theory

Must inherit all axioms as well



Problems

- rec(Theory. •••) does not work!
  - Typing rule is too simple.
- Build type on top of Y-combinator

```
add rec def  TheoryItemType{i}(n)
     name:IDList
     × label:Atom
     × flags:List of Z
     × lib:List of TheoryLibType
     × preds:List of m:Nn    TheoryItemType{i}(m)
     × ((parents:TheoryTypePreds(preds)
       ⟶ (name:ID ⟶ Lookup(Type,preds,fail) name in parents using preds)
       ⟶ U{i}) | ((m:Nn × TheoryItemType{i}(m)) × LibId) | Unit) ;;
```

Y-combinator

- Can hide all instances of bounds by using derived rules
- All proofs are by (normal) induction
- No fixed-point semantics

Data structure

- In a functional language where equality is undecidable, how can we represent a DAG? (Would like an equality over strings)
- My solution
  - need an arbitrary total order.
  - the DAG edges are integer indices into the total order
  - Not elegant!

Results

- Can reason about theories
  A theory is a normal object, reasoning is typically by induction
- Examples
  - Knowledge monotonically increases
  - Equivalence of theories
  - "Lifting" of theorems: any theorem can be lifted to be beneath its immediate predecessors

Results II

- Naming is better
- Inheritance is built-in
- Theories are extendible
- Type theory is pretty powerful—but we are skirting the bounds of its abilities

Problems

- Whenever a theorem is proved, the axioms are extracted by direct computation
- Not scalable—better ways to do it
- Theory mechanism is too visible
- Naming should be extended to the library mechanism
- Abstraction should extend to ML and proofs
- Many problems require "deep thought"

Future work

- Package theories in a hierarchy
  - Conceptual blocks
  - Assist in naming
- Theory modification—list element deletion

TheoryType

```
* ML theory type switch ml
% Given a  TheoryItemType{i}(n) , construct a product of all the axioms
    in the theory.  This effectively squashes all the junk out of a theory
    to provide a type that can be quantified over.
  %
  add rec def  TheoryTypeSwitch(case, item)
      case case
      of inl( ) =>
          parents:TheoryTypeSwitch(inr  , item.preds)
          × Switch x = item.term of
            Axiom ⟶ x parents (λname.Lookup(preds,fail) name in parents using item.preds)
            Theorem ⟶ Unit
            Else ⟶ Unit
      | inr( ) =>
          Case item of
          CNil => Unit
          hd::tl => TheoryTypeSwitch(inl  , hd.theory) × TheoryTypeSwitch(inr  , tl) ;;
```

TheoryLookup

```
* ML theory lookup switch ml
% Given a  TheoryItemType{i}(n) , and a term of  TheoryType(item) ,
    and a name of type  ID , return the term inhabinting the axiom by that name.
  %
  add rec def  Lookup[switch] name in term using item
       case switch
       of inl( ) =>
           if name ∈ item.name
           then inr term.2
           else Lookup[inr ] name in term.1 using item.preds
           fi
       | inr( ) =>
           Case item of
           CNil => inl
           hd::tl => case Lookup[inl ] name in term.1 using hd.theory
                     of inl( ) =>
                         Lookup[inr ] name in term.2 using tl
                     | inr(x) =>
                         inr x ;;
```

TheoryLookupType

```
* ML theory lookup type switch ml
% Give a  TheoryItemType{i}(n) , and a term of  TheoryType(item) ,
    and a name  ID , return the axiom of that name.
  %
  add rec def  Lookup(type)[switch] name in term using item

   case switch
   of inl( ) =>
       if name ∈ item.name
       then inr Switch x = item.term of
                Axiom ⟶ x (term.1) ( name.Lookup(preds,fail) name in term.1 using item.preds)
                Theorem ⟶ Unit
                Else ⟶ Unit
       else Lookup(type)[inr ] name in term.1 using item.preds
       fi
   | inr( ) =>
       Case item of
       CNil => inl Unit
       hd::tl => case Lookup(type)[inl ] name in term.1 using hd.theory
                  of inl( ) =>
                      Lookup(type)[inr ] name in term.2 using tl
                  | inr(x) =>
                      inr x ;;
```