Formal Modules (Abstract Data Types)

and object oriented programming

- Jason J. Hickey
- Cornell University

Outline

- Programming motivation
- Modularity
- Very-dependent types
- Related work
- Grand Unified Type Theory (GUTT)

Scalable programming

- Large sd•ftware systems

Formal programming

- Logical specifications
- Functional programming

  Automatic program extraction

    Unified framework

  - Lack of modularity

    Q7 versus:

  Q67 Fault tolerance

Guidelines for formal software design

Compromise

- Require more explicit specifications
  - Use constructivism constructively
  - Higher level types (specification for free)
  - Higher level assertions
  - Static (as well as runtime) verification
- Provide tools for modularity
  - Abstraction of object properties
  - Abstraction barriers to isolate implementations

Definitions

- Abstract Data Type = Module specification = Class
- Module implementation = Object
  Object

Modularity

(A real-life example)

```
type Torso =
    { s:Spine, r:Rib list,
```

Modularity

(A mathematical example)

Semi-group is *car* : Type

£ =: ! ! car car Prop

£ 8*x*: :*x* = *x*

£ 8*x*; *y*: :*x* *y*) = *y* *x*

x; *y*; *z*: :*x* *y*) = *y* *z*) = *x* *z*

£ ' :

x; *y*; *z*: :(*x* *y*) = *z* *x* (*y* *z*)

£ 1:

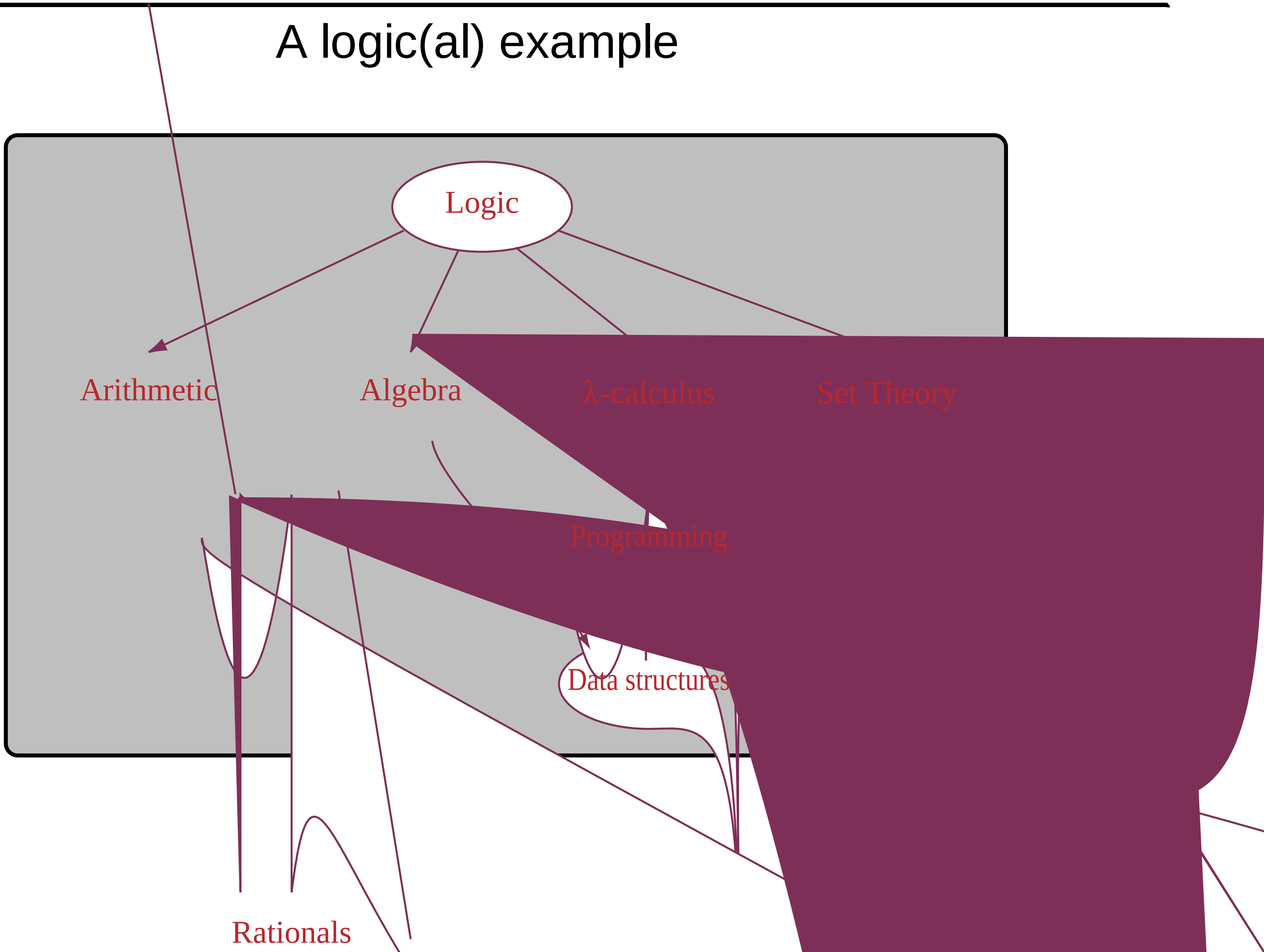
£ 8*x*: :*x* 1 = *x*

is

£ 8*x*: :9*y* :*x* ' *y* = 1

Modularity

A logic(al) example



How are these concepts related?

- Proposal: they are all identical
- They can be implemented the same way
- They have similar, if not exactly the same, informal semantics
- Each is an instance of a qualified assertion
- Great observation—what does it buy us?
  - A formal concept of an “object”
  - A formal relation between the logic and the object specification
  - A reflection of modularity onto the semantics
  - Representation coercions are the identity

How can we extend the formalism?

- Implement a class as an extensible, dependent recdXrd type

Primitive object

- A collection of “parent” specif-2•cations

Very dependent types

Dependent type x:D Æ R(x)

- Range specification depends on argument
- D is a type, and for any x ∈ D, R(x) is a type
- f ∈ x:D → R if f is a function, and for x ∈ D, f(x) ∈ R(x)
- Month functions: m:{1.gı12}→ {1...DaysPerMonth(m)}

l: ! Person case *l* of

Field	Type
name	String
age	*N*
employer	Company
salary	*Z*

Very dependent types

Very dependent function type {f | x:D → R(x, f) }

$f\,(car$ f

car car car $' \quad ' \quad = \quad ' \quad '$

car car $Prop$

$car\ '$ $=$ $op\{assoc$ $car\ f car$

$\forall x; y; z:$ Typ

$\forall Label$ $f\,(car$ $f\,(=)$

car $'$ $=$ $op\{assoc$

f $f\,(\ '\ )\,(\ '$

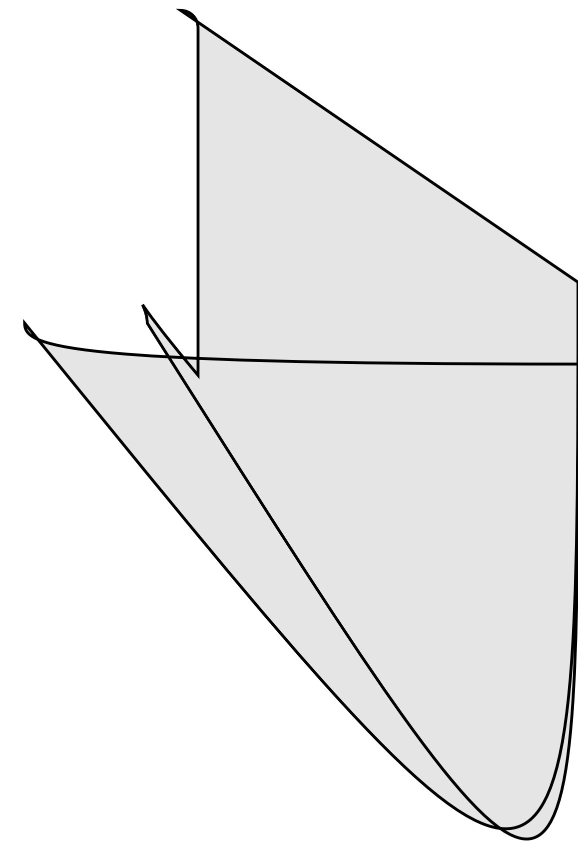
...

Do we need a formal definition?

Alternative semantics

Other very dependent types

Very dependent W-types



Other very dependent types

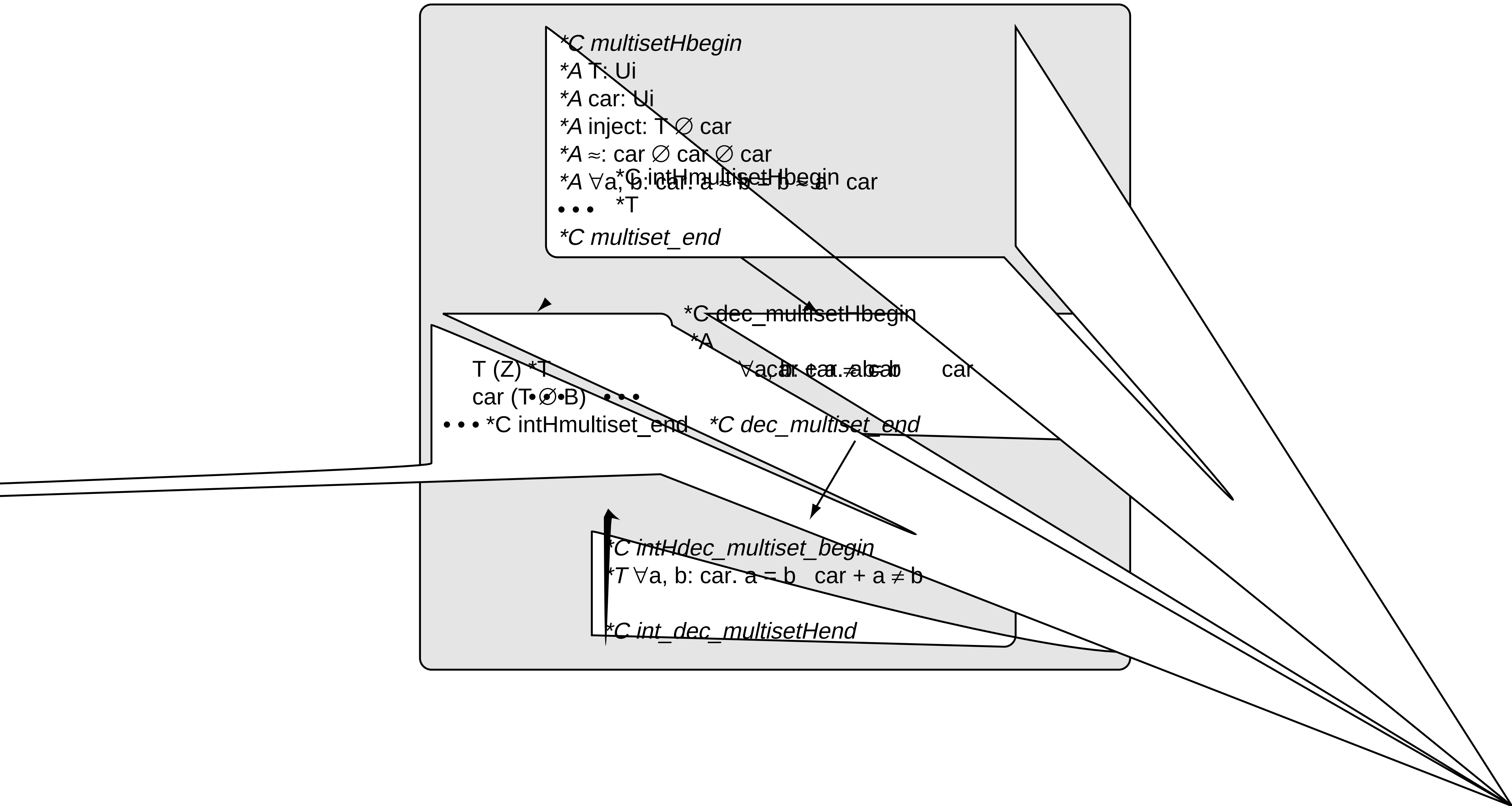
Very dependent recursive types

- List of T: „$(X{:}\mathit{Unit} + X$ £ $T)$
  Dependent list of T: „$(X{:}\mathit{Unit} + x{:}X$ £ $T(x))$

- May have no fixed point
- The meaning is clear:
  - The type is its • unrolling
  - Union of all unrollings
- We don't need an extension to the type theory—just one ordinal i: ! £ T^i
- Or an indexed union $[_{i 2}\ T^i$

Class definitions

- A class is a theory



Construction by formation

- A class can be defined as a theory
  - Axioms are method specifications
  - Theorems are method implementations
  - Other objects (definitions, etc) behave the same
  - Inheritance is implicit
- Each class has a type
  - Methods are projections
  - IsA hierarchy is explicit, or maintained as theory dependencies
- Still support explicit methods
- Coercions can be provided (explicitely, or as a theory)

Points to consider

- Recursive classes

Grand Unified Theory

What is a rule?

- H:Sequent* $\times$ C:Sequent
- If H are tue, then C is tue

What is a theory?

H:Type* $\times$ T:ype*

- H and T are well-formed
- If H are true, then T are true

Unify rules and theorh˝6 by removing well-formedness constraints

- Incorporate hierarchical nature
- Generalize sequents by making them recursive:
  - H:(Sequent + Type) $\times$ C:(Sequent + Type)
  - If H are true, well-formed, and functional, then is true

The unification point:

- We can automatically convert between:
  - Rules
  - Steps
  - Sequents
  - Theori
  - Type theories
  - Tactics
- Rest remain as-is:
  - Computation rules
  - Definitions (special case of computation)