Modules

15-150 Lec 2, Frank Pfenning

Lecture 16

Thursday, March 19, 2020

Learning Objectives

Learning Objectives

- What is a module?

Learning Objectives

- What is a module?
- Which properties may a module possess or not?

Learning Objectives

- What is a module?
- Which properties may a module possess or not?
- What are the consequences for the programmer?

Learning Objectives

- What is a module?
- Which properties may a module possess or not?
- What are the consequences for the programmer?

- signatures and structures in SML

Learning Objectives

- What is a module?
- Which properties may a module possess or not?
- What are the consequences for the programmer?

- signatures and structures in SML
- Implementing data structures

Learning Objectives

- What is a module?
- Which properties may a module possess or not?
- What are the consequences for the programmer?

- signatures and structures in SML
- Implementing data structures

Learning Objectives

- What is a module?
- Which properties may a module possess or not?
- What are the consequences for the programmer?

- signatures and structures in SML
- Implementing data structures

- Next lecture: functors

Outline

Outline

- Module concepts
  - Transparent and opaque signature ascription
  - Name space management
  - Data abstraction / representation independence
  - SML Standard Basis Library

Outline

- Module concepts
  - Transparent and opaque signature ascription
  - Name space management
  - Data abstraction / representation independence
  - SML Standard Basis Library
- Data structures
  - Representation invariants
  - Persistent vs. ephemeral data structures
  - Queues, Binary search trees

Units of Code

Units of Code

- Expression (command)

Units of Code

- Expression (command)
- Function (procedure)

Units of Code

- Expression (command)
- Function (procedure)
- Structure (module, library)

Units of Code

Type as interface to compiler/runtime system

- Expression (command)
- Function (procedure)
- Structure (module, library)

Units of Code

- Expression (command)
- Function (procedure)
- Structure (module, library)

Type as interface to compiler/runtime system

Type as interface to function

Units of Code

- Expression (command)
- Function (procedure)
- Structure (module, library)

Type as interface to compiler/runtime system

Type as interface to function

Signature as interface to structure

Units of Code

- Expression (command)
- Function (procedure)
- Structure (module, library)

Type as interface to compiler/runtime system

Type as interface to function

Signature as interface to structure

Typing at each level accomplishes different things but bigger units rely on properties established for smaller units

Name Space Management

Name Space Management

- `structure <struct> : <sig> = struct … end`

Name Space Management

- `structure <struct> : <sig> = struct … end`
- Only types and values in signature can be mentioned
  - `<struct>.<type>` or `<struct>.<value>`

Name Space Management

- `structure <struct> : <sig> = struct … end`
- Only types and values in signature can be mentioned
  - `<struct>.<type>` or `<struct>.<value>`
- But type definitions are transparent
  - Lack of data abstraction
  - Only name space management

Name Space Management

- `structure <struct> : <sig> = struct … end`
- Only types and values in signature can be mentioned
  - `<struct>.<type>` or `<struct>.<value>`
- But type definitions are transparent
  - Lack of data abstraction
  - Only name space management

I never use transparent ascription for structures (too easy to get tripped up because of the illusion of abstraction)

Name Space Management

- `structure <struct> : <sig> = struct … end`
- Only types and values in signature can be mentioned
  - `<struct>.<type>` or `<struct>.<value>`
- But type definitions are transparent
  - Lack of data abstraction
  - Only name space management

I never use transparent ascription for structures (too easy to get tripped up because of the illusion of abstraction)

Need to reconsider in next lecture for functors

Data Structure Persistence

Data Structure Persistence

- Pure data structures are persistent
  - “Old generations” of a data structure may remain accessible
  - “Garbage collection” deallocates

Data Structure Persistence

- Pure data structures are persistent
  - “Old generations” of a data structure may remain accessible
  - “Garbage collection” deallocates
- Mutable data structures are ephemeral
  - Old versions are no longer accessible
  - Need to track state and state changes globally
  - May need to deallocate manually

Data Structure Persistence

- Pure data structures are persistent
  - “Old generations” of a data structure may remain accessible
  - “Garbage collection” deallocates
- Mutable data structures are ephemeral
  - Old versions are no longer accessible
  - Need to track state and state changes globally
  - May need to deallocate manually
- Ephemeral data structures are a significant source of bugs

Data Abstraction

Data Abstraction

- Client cannot tell or exploit representation

Data Abstraction

- Client cannot tell or exploit representation
- Does this remind you of anything?

Data Abstraction

- Client cannot tell or exploit representation
- Does this remind you of anything?

Yes! It lifts parametric polymorphism at the level of functions to structures

Data Abstraction

- Client cannot tell or exploit representation
- Does this remind you of anything?

Yes! It lifts parametric polymorphism at the level of functions to structures

- For example, `? : 'a -> 'a -> 'a`

Data Abstraction

- Client cannot tell or exploit representation
- Does this remind you of anything?

Yes! It lifts parametric polymorphism at the level of functions to structures

- For example, `? : 'a -> 'a -> 'a`
- Important consequence
  - We can replace an implementation with a better one!
  - As long as that is (also) correct, the client will continue to work
  - Very few languages support this form of guarantee

Summary

Summary

- Signatures are interfaces to structures
  - Contain concrete and abstract types
  - Contain declarations of types for values (including functions, of course)

Summary

- Signatures are interfaces to structures
  - Contain concrete and abstract types
  - Contain declarations of types for values (including functions, of course)
- Transparent signature ascription provides name space management

Summary

- Signatures are interfaces to structures
  - Contain concrete and abstract types
  - Contain declarations of types for values (including functions, of course)
- Transparent signature ascription provides name space management
- Opaque signature ascription provides data abstraction
  - Reap the full benefits of a well-designed language and type system
  - Guarantees for every program, automatically, rather than conventions

Summary

- Signatures are interfaces to structures
  - Contain concrete and abstract types
  - Contain declarations of types for values (including functions, of course)
- Transparent signature ascription provides name space management
- Opaque signature ascription provides data abstraction
  - Reap the full benefits of a well-designed language and type system
  - Guarantees for every program, automatically, rather than conventions
- Only data abstraction/representation independence makes programming truly modular