A Type System for Higher-Order Modules (Survey)

Ruchira Datta

CS 263: Design of Programming Languages
University of California
Berkeley, California

December 5, 2001

What Are We Talking About?

- *Type System*: the "type" of a module e is its *signature* σ
  - signature judgement $e : \sigma$
- *Higher-Order*: a *functor* $\Pi s : \sigma . \tau$ takes a module s as argument and returns another module
  - functor signature must specify argument signature σ and return signature τ
- Surveying eponymous paper of Crary, Harper, and Dreyer
- Module calculus extends and unifies work of Harper, Lillibridge, Leroy, Russo, etc.

Controlled Abstraction

- modules are a kind of sum: tuples containing module elements
- *opaque module* (*strong sum*) hides all actual types used in concrete implementation during typechecking
  - provides abstraction through hiding
  - difficult to use
- *transparent module* reveals representation during typechecking
  - allows typechecker to see same type used in different modules
  - breaks abstraction
- *translucent sum* reveals those types the programmer wants to reveal, hides the rest
- choose opacity through sealing: *strong* $M :> \sigma$ or *weak* $M :: \sigma$

Generativity

- judge equivalence of modules by equivalence of all static components
- may want all instances of an abstract type to be inequivalent
- each instance comes with a fresh stamp
- stamping is a side-effect, so such instantiation is impure
- functors can be
  - *generative*: all result modules are inequivalent (e.g., Standard ML)
  - *applicative*: applications to equivalent argument modules give equivalent result modules (e.g., Ocaml)
- here programmer can choose: Π^{gen} vs. Π, :> vs. ::

Singleton Signatures

- *singleton signature* packs type τ into signature $[\tau]$ containing only single component, of type τ
- can extract type from a module M with singleton signature by $Ext\, M$, i.e., $Ext[T] = T$
- $\mathcal{S}_\sigma(M)$ is singleton signature of modules having signature σ, which are (statically) equivalent to M
- $\mathcal{S}_\sigma(M) \leq \sigma$
- module functions primitive, ordinary ones built from them
  - e.g., $\lambda x : \tau . e(x) \equiv \Lambda s : [\tau] : e(ext\, s)$

Determinacy

- module is *determinate* if it can be equivalent to another module
- generative signatures are never determinate
- module is *pure* if effect-free
- if module is pure, weak sealing of that module is pure (opaque, but nongenerative)
- functor with weakly sealed modules in body can be applicative

Principal Signatures and Decidable Typechecking

- *principal signature*: most generous signature for module
- used in separate compilation: check only signatures
- allow subsignatures, but disallow subtypes (of values)
  - otherwise checking subtyping, subsignatures would be mutually recursive
  - termination would require some metric which would not be invariant (might grow) under substitution

Quantifiers

- so far cannot compute principal signature due to *avoidance problem*
  - let functor F have signature $\Pi s : \sigma_1 . \sigma_2$
  - want to typecheck $F(M :> \sigma_1)$
  - $M :> \sigma_1$ is indeterminate, so must leave σ_1 alone
  - must find σ_2' which avoids mention of s, with $\sigma_2 \leq \sigma_2'$
  - no unique one, so use existential quantifier: $\sigma_2 \leq \exists s : \sigma_1 . \sigma_2$
- quantifiers require higher-order unification, prevent decidable typechecking
- only typechecker allowed to use quantifiers; programmers can't use them

Modules as First-Class Values

- would like to be able to use modules as first-class values
- could choose appropriate implementation of signature at run-time
- wrap second-class modules as existential packages to get first-class modules
  - $\langle|\sigma|\rangle \equiv \forall\alpha.(\sigma \rightarrow \alpha) \rightarrow \alpha$
  - $\langle|M|\rangle \equiv \Lambda\alpha.\lambda f : (\sigma \rightarrow \alpha).f M$