Types for Parameterized Modules

Stevie Strickland

@inproceedings{hl-popl94:ho-modules,
    author    = {Robert Harper and Mark Lillibridge},
    title     = {A Type-Theoretic Approach to Higher-Order Modules with Sharing},
    booktitle = {POPL},
    year      = {1994},
    pages     = {123-137},}

Paper [http://doi.acm.org/10.1145/174675.176927](http://doi.acm.org/10.1145/174675.176927)

Summary The first module systems for ML contain transparent structures. That is, the types provided by a structure are fully known by any client. In addition, signatures, or module interfaces, in these systems are opaque—the programmer could only list provided type names in signatures. This disallows type equations that might give clues to how different provided types might be able to interact.

In this paper, Harper and Lillibridge describe the model for a new module system for ML where structures are represented as translucent sums, an extension of weak dependent sums. The model allows for subsumption between translucent sum types, and this subsumption allows for hiding portions of the types provided by the module—in other words, it provides translucent modules.

Contributions This work is one of the first two approaches\footnote{The second was manifest types\cite{leroy} by Xavier Leroy.} to adding support for partial type abstraction to the ML module system. This enables sealing of modules, so that abstraction can be enforced in the type system. In addition, this work, as well as the other, models generative functors. The
previous model for ML modules, which used strong dependent sums, could only describe applicative functors.

@inproceedings{ds-icfp96:mixin-modules,
    author = {Dominic Duggan and Constantinos Sourelis},
    title = {Mixin Modules},
    booktitle = {ICFP},
    year = {1996},
    pages = {262-273},}

Paper  http://doi.acm.org/10.1145/232627.232654

Summary  Duggan and Sourelis take the ideas of mixins from the Common Lisp Object System and apply them to the ML module system. Their system allows for open recursion between datatypes and function definitions in different mixins. This provides a method of extension that supports the development of programs that would otherwise require cyclic dependencies between modules.

Contributions  This work describes the first attempt to design mutually recursive typed modules. However, their mixins cannot be compiled separately, since the described semantics requires the ability to combine the mixins in a syntactic fashion into a single, normal ML module when “closed.” This system has not been used or extended, and thus represents a dead-end in the design space.

@inproceedings{ff-pldi98:units,
    author = {Matthew Flatt and Matthias Felleisen},
    title = {Units: Cool Modules for HOT Languages},
    booktitle = {PLDI},
    year = {1998},
    pages = {236-248},}

Paper  http://doi.acm.org/10.1145/277650.277730
Summary  Flatt and Felleisen describe a system of first-class modules called units. Each unit provides a list of imported and exported names as well as a list of definitions and a body expression. Units can be combined into new compound unit values, and the body of a unit whose imports are completely satisfied can be executed.

They provide three models for the system. The first model is dynamically typed, the second model adds a type system and the ability to import and export opaque types, and the third model adds the ability to specify the dependencies between imported and exported types.

Contributions  This work presents the first full model and type system for a language with first-class, mutually recursive modules. This work, extended by Owens and Flatt[2], is the basis for the PLT Scheme unit system.

@inproceedings{dch-popl03:ho-modules,
    author    = {Derek Dreyer and Karl Crary and Robert Harper},
    title     = {A type system for higher-order modules},
    booktitle = {POPL},
    year      = {2003},
    pages     = {236-249},}

Paper  http://doi.acm.org/10.1145/640128.604151


Summary  Dreyer et al. provide a model of ML modules that can describe both the generative functors of Standard ML and the applicative functors of OCaml. Previous approaches to modeling ML modules introduce new language features which represent modules to a base λ-calculus model. Instead, Dreyer et al. show how to compile modules into already-existing features of System Fω, their base calculus. In addition, they separate the notion of signature sealing into weak and strong versions, where the weak notion still allows the result to be used applicatively.
Contributions  This work removes the need for special language constructs to model ML modules. It also improves on Shao’s ICFP 1999 paper[3], which also aimed to model both applicative and generative functors, by adding another notion of sealing which does not always force generativity.

References

