Chapter 12

Conclusion

We have developed extensional polymorphism, an ad-hoc polymorphism extension over the ML language. Generic values, ad-hoc polymorphic values in our framework, provide us various useful features, which were not possible in the classical ML parametric polymorphism: type dependent semantics of ad-hoc polymorphism, run time type tagged values such as dynamic values and for typed value I/O, and user definability of overloaded values. This extension of polymorphism reduces irritation of programmers a lot, who were captured inside the simple but restricted ML parametric polymorphism.

Core system

To achieve the type dependent semantics in the classical ML run time system without types, we proposed a compilation scheme which translates our full type-annotated source language called GML to an intermediate language called IML without type annotations. Type informations required for generic values are converted to normal ML values called run time types. These run time types are dispatched and delivered to generic values in the manner of type applications and abstractions of system F.

The run time type dispatching is useless for normal parametric polymorphic values, since their semantics are independent of types. Therefore, for efficiency, these unwanted type dispatching should be eliminated from the system F style translation with full type applications and abstractions. For this purpose, generic values must be distinguished from normal parametric polymorphic values in our type system: quantified type variables in type schemes for ad-hoc polymorphism have a special dynamic attribute. Thanks to this distinction, type dispatching is performed only for generalizations and instantiations of these dynamic type variables, therefore classical programs without generic values are compiled to themselves and are evaluated without any overhead.

The mechanism of run time type dispatching allows us to introduce ad-hoc polymorphic primitive values, called generic primitives. Their type-dependent semantics are only defined in the level of the intermediate language, as functions which take run time types indicating type instantiation of primitives. Having type instantiation information, generic primitives can perform type-dependent calculation, such as ML value printing for example. In the source language, derived generic values can be defined by users using let definitions with generic primitives or other derived generic values.

Applications of generic primitives – dynamic values and type safe value I/O

As an illustration of the power of extensional polymorphism framework, we implemented dynamic values. Dynamic values are an abstract data type, which encapsulates a value and its type. The primitives of encapsulation and decapsulation of values and their types can be implemented as generic primitives: for example, the encapsulation primitive `dynamic` is compiled to a simple function which takes a type and a value, then returns a tuple of them. Thus, compilation of dynamic value primitives are obtained almost for free in extensional polymorphism. However, special care must be taken for their typing. Captured inside dynamic values, type variables can be exported outside of their scope. Therefore encapsulation of type variables inside the type-value pair of a dynamic value must be restricted only to those generalizable and its content with type variables are considered to be parametric polymorphic. For this purpose, we added another poly-dynamic attribute to type variables related to dynamization and a special typing rule of `dynamic` primitive for explicit type generalization check of its argument: polymorphic dynamization, encapsulation of type variables into dynamic values is limited to the argument of `dynamic` primitive. We can still define derived dynamizers, derived generic values using `dynamic` primitive polymorphically inside their definitions, but they can only encapsulate values with closed types. We have also studied further typing discipline of derived dynamizers which reduces this limitation inside ML's first order polymorphism.

As another application example based on generic primitives, we proposed type safe ML value I/O system, which assures type safety of value communication between different ML programs. The same approach as dynamic values is taken again: when we output a value, it is automatically tagged with its type information using type dispatching mechanism of extensional polymorphism. When it is read back, encapsulated type information of the type-value pair is compared to the expected type in the context for the type safety. Unlike dynamic values, type-value pairs of value I/O can be exchanged between arbitrary different ML programs via files or streams. Therefore encapsulated type information must be universally meaningful. Even if data types of different programs share the same name, they may represent completely unrelated data structure. Hence, in type safe value I/O, data types are not identified by their names as in dynamic values, but by their semantics, which are given intuitively by their definitions. More specifically in Objective Caml, since names of data type definitions do not affect their internal value representation, we use nameless version of data type definitions as their identifications in type safe value I/O, which allows type safe value exchange between different data types with the same internal data structure. In the actual implementation, to avoid the size explosion of type information of type-value pairs for type safe value I/O, nameless type definitions are stored as encrypted checksums called data type digests.

Under the existence of value I/O, type information encapsulated inside dynamic value's type-value pair can also be exported outside of programs. In such a circumstance, data type identification at coercions of dynamic values cannot be done only by names either. At coercion of a data type, we also have to check its semantics are really compatible to the expected data type for type security. Therefore data type information encapsulated inside dynamic values must be tagged with their data type digests, too. The type-value pairs of dynamic values and type safe value I/O are extended to one unified format, external values: type information stored inside an external value contains both of data type names and their digests. The encapsulation primitives are also unified into the unique `dynamic` primitive, which creates this new type-value pair. On the other hand, we have two decapsulation primitives `import` and `coerce`, corresponding with two decapsulation schemes which come from both systems: `import` primitive only checks data type digests at decapsulation and allows value exchange between different data types but with the same internal representation. `coerce` primitive provides more strict decapsulation of external values: in addition to the digest check, it also compares data type names.

User definition of generic values

Based on the core system of extensional polymorphism, we proposed user-definability of generic values in the source language. The new `generic` binding defines a user-defined generic value, by overloading a fixed list of type cases, expressions indexed by their type schemes called type patterns. Type case selection in evaluation of a user-defined generic value instance is performed in the manner of ML pattern matching: its evaluation type context is matched against the type patterns, in their declaration order. If there are more than one matching type case, the first one is always selected. This type case selection scheme is achieved by pattern matching of dispatched run time types in the compilation of user-defined generic value definitions. Thanks to the selection priority embedded in the `generic` binding, we have almost no restrictions between type cases overloaded in one generic definition. For example, a user-defined generic value can have type cases whose type patterns overlap each other, which are not permitted in Haskell type classes where overloaded instances are defined separately and therefore there is no their clear selection priority. Though user generics are not open to extensions *in the sense of Haskell type classes*, this type case overloading without any restriction of `generic` binding provides us a great expressive power of overloading. Just by listing type cases, we can easily define generic values from simple overloaded arithmetic operators to complex recursive generic values such as functions with variable numbers of arguments, which are not definable, or are definable but difficult to use in the Haskell framework.

In our ML style typing system, the type scheme of a user-defined generic value is given as the least general anti-unifiers of its type patterns. But usually this type scheme is too general: a user-defined generic value can be non-exhaustive compared to its type scheme and its semantics may not be defined for some instance type. Normal type schemes with dynamic type variables do not have enough power to express non-exhaustiveness of user generics. To detect illegal use of user generics without semantics, we have built a new generic constraint typing system. For the theoretical cleanness and implementation modularity, this generic constraint typing forms an independent typing layer of the existing ML style normal typing layer. In the constraint typing system, generic values are assigned new type information called generic constraints. Intuitively a generic constraint of a user-defined generic value consists of the list of its type patterns, and constrains instantiation of the type scheme of the generic value which is assigned in the normal typing layer.

The generic constraint typing layer validates user-defined generic instances by simulating all the possible type case selections statically. A validated program is compiled to a code which perform the same type case selections by pattern matching of dispatched run time types at run time. This run time pattern matching of run time types is clearly redundant once the program is validated by the constraint typing. We can improve the efficiency of type case selections of user generics by dispatching some "witness" of their well-typedness obtained in the generic constraint typing, instead of run time types. This witness is an integer graph called a generic flow and obtained from the generic constraint type checking. Each node of a generic flow corresponds with a generic instance evaluation, and its integer index specifies which type case should be selected in the evaluation. There are two important benefits of using generic flows instead of run time type matching: the one is efficiency of type case selection. In the classical compilation with run time type matching, the time for type case selection was proportional to length and complexity of patterns. With generic flows, type case selection is done in constant time, no matter how many and complex the type patterns are. In addition, even in examples with the simplest type pattern, type case selection by an integer index of generic flow passing is 1.5 time faster than the selection by pattern matching. The other more important benefit of generic flows is separation of type case selection strategy from run time restrictions. By migrating actual work in the static typing phase, we can perform back-tracking of type case selections, which was practically impossible in the type case selection by run time type matching due to its complexity. Back-tracking of type case selection expands semantics and adds more valid instance types of user generics, preserving all of the semantics of generic instances which were valid in the type matching based system. In addition, under the back-tracking selection, we have a type substitution lemma of user generic instances, which is a very important property for the type reconstruction of user generics.

Generic flow dispatching can be considered as a generalization of dictionary passing of Haskell. While dictionaries consist of overloaded instances themselves to be evaluated, in the generic flows, a selected type case is represented as an integer index, which selects the type case on the fly. This indirection is required in extensional polymorphism, since user generics can be defined locally, and their type case definition are not always globally available, while class instances must be global in Haskell.

Like other systems for overloading, extensional polymorphism faces to type ambiguity problem of user generics, since they lack principal typing. We have proposed a type reconstruction algorithm for user generics with automatic ambiguity resolution. As the generic constraint typing forms an independent layer of the normal ML style typing, generic constraint type reconstruction algorithm is separated from the normal type reconstruction. Source programs without types are first typed in the normal type reconstruction, then the results are sent to the generic constraint type reconstruction algorithm, which instantiates too general types of user generic instances so that they can be valid instances in the constraint typing layer. Same as the type case selection mechanism, we make use of the definition order of type cases in the ambiguity resolution: if a generic instance has ambiguity: more than one type patterns which may match with its current type, the first match possibility is selected automatically and its type is instantiated so that it can really match with the selected type pattern. This ambiguity resolution by type case definition ordering allows us to control default typing of user generics easily by reordering type case definitions.

Finally, we have argued extensibility of user generics. A user-defined generic value was defined by `generic` binding, listing all the type case definitions once and for all. This goes against programming modularity, since in many cases, overloaded type cases are semantically independent each other and only they share is the name and the vague notion of their functionality, "plus" for example. Because of this independence of semantics of type cases, it is desirable that type case could be defined in different places. Our approach to the extension of overloaded values are different from Haskell type classes based on the open world assumption, where each class instance declaration extends overloaded values globally. Such open extension system does not match with the eagerly evaluated ML semantics, since under the open world assumption, the semantics of overloaded values are dynamic until link time. Our extension of user generics is closed: each extension defines a new value and therefore the semantics of generic values are static. A new type case inclusion syntax construct is introduced to `generic` binding. By "including" already existing generic values inside a new definition of user generics, the type cases of included generics can be easily imported into the newly defined generic value. Merging type cases of type cases of user generics, we can quickly define generic values with extended semantics. Thus, our closed extension is based on redefinition of values, however, it is not just so simple. It also supports open recursion: semantics of a recursive generics are open to extensions, so that merged type cases by case inclusion can call each other recursively. For the redirection of recursive calls in the open recursion, compilation of a type case has a new functional argument for its recursive call back point and the recursive occurrences of user generics are replaced by this functional abstraction. The semantics of a user-defined generic value is given as the fixed point of the set of type case compilation functions.

Bibliography

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[ACPR92] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. In *Proceedings of the 1992 workshop on ML and its applications*, 1992.

[BTCGS94] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 197–245. The MIT Press, Cambridge, MA, 1994.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[dR] Daniel de Rauglaudre. IoXML: an OCaml syntax extension to parse and print one's OCaml values in XML format. Available at `http://cristal.inria.fr/~ddr/IoXML/`.

[DRW95] Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *Proceedings of the 22th ACM Conference on Principles of Programming Languages*, January 1995.

[Gil] Stephen Gilmore. Programming in Standard ML '97. Available at `http://www.dcs.ed.ac.uk/home/stg/NOTES/`.

[GM96] James Gosling and Henry McGilton. The Java language environment, a white paper. Available at `http://java.sun.com/docs/white/langenv/`, 1996.

[GTL90] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1990.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[HJW92] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell, version 1.2. Technical report, Yale University, 1992.

[JJM97] S. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space, 1997.

[Jon97] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1997.

[Jon00] Mark P. Jones. Type classes with functional dependencies. *Lecture Notes in Computer Science*, 1782:230–, 2000.

[Jon94] Mark P. Jones. The implementation of the Gofer functional programming system. Technical Report YALEU/DCS/RR-1030, New Haven, Connecticut, USA, May 1994, May 94.

[JP] Mark P. Jones and John C. Peterson. Hugs 98: A functional programming system based on haskell 98 - user manual.

[Ler] Xavier Leroy. The Objective Caml system release 3.06. Available at `http://caml.inria.fr/ocaml/htmlman/index.html`.

[LM94] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1994.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, number 167 in Lecture Notes in Computer Science, pages 217–228. Springer-Verlag, 1984.

[Oho89] Atsushi Ohori. A simple semantics for ml polymorphism. In *Proc ACM/IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 281–292, 1989.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Functional Programming Languages and Computer Architecture*, pages 135–146, 1995.

[PL99] Francois Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Symposium on Principles of Programming Languages*, pages 276–290, 1999.

[Riv92] R. Rivest. The MD5 message-digest algorithm, 1992.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[TA95] Andrew P. Tolmach and Andrew W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[W3C00] W3C. Extensible markup language (XML) 1.0. Available at `http://www.w3.org/TR/2000/REC-xml-20001006`, 2000.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

Contents

Index