The Dule Module System and Programming Language Project

Mikołaj Konarski
mikon@mimuw.edu.pl

Institute of Informatics, Warsaw University

The Dule project is an experiment in large-scale fine-grained modular programming employing a terse notation based on an elementary categorical model. The Dule module system remedies the known bureaucratic, debugging and maintenance problems of functor-based modular programming (SML, OCaml) by introducing simple modular operations with succinct notation inspired by the simplicity of its semantic categorical model and a modularization methodology biased towards program structuring, rather than code-reuse. The same categorical model and its natural extensions induce an abstract machine based on their equational theories and inspire novel functional core language features that sometimes complement nicely the modular mechanisms and sometimes are language experiments on their own.

The assets of the project are gathered on its homepage at http://www.mimuw.edu.pl/~mikon/dule.html, where the formal definition of the current version of the language as well as an experimental compiler with a body of Dule programs can be obtained. In its current state, the definition is complete, but some of the theoretical hypotheses it has brought out are unverified and some of the suggested ideas for future user language and abstract machine mechanisms are unexplored. The compiler works adequately, but it still does not bootstrap and the lack of low-level libraries precludes practical applications. The compiler (except the type reconstruction algorithms) and the abstract machine are a straightforward implementation of the formal definition, so their efficiency can (and should) be vastly improved. There is ten thousand lines of fine-grained modular Dule code, but much more will be needed to discover all scaling problems and fine-tune the modularization methodology, possibly changing the modular operations accordingly (but not the underlying categorical model at this stage, we hope).

1 BACKGROUND

Modular programming is necessary due to the growing size, complication and the requirement of modifiability of computer programs. The most useful approach appears to be the integration of rigorous module systems into high level programming languages, as in Standard ML [30] or OCaml [27]. However, strictly modular programming style, with explicitly defined modular dependencies, can be sustained only in small programming projects written in these languages. In large projects, managing the many layers of abstraction, introduced with the module hierarchy,

turns out to be harder even than manual tracing of each particular dependency in unstructured code [33]. Abstract and fine-grained modular programming is cumbersome, because the headers to be written are complicated and module applications often trigger global type sharing [25] errors. When a module has n parameters, $O(n^2)$ potential typing conflicts await at each application, making not only creation of modules, but also their use and maintenance very costly.

Another problem is the tension between abstraction, expressiveness and applicability of a module. In simplification: the more abstract a specification of a module result is, the easier it is to implement the module, but the harder it is to use the module. Without specialized module system mechanisms, this global tension can only be solved by making each module excessively powerful and general. However, the usual practical approach is to gradually sacrifice abstraction as the program grows, whereas, especially for ensuring correctness of large programs, the abstraction is crucial [4]. Huge collections of interdependent modules themselves require modularization, lest managing the modules becomes as tedious as tending the mass of individual entities in non-modular programs. Grouping of modules performed using the mechanism of submodules often overloads the programmer with type sharing bureaucracy or requires a violation of the abstraction guarantees. Other solutions, some of them using external tools, are usually very crude and incur the risk of name-space clashes or ignore abstraction.

2 DULE LANGUAGE FEATURES

Dule is a programming language and a module system with a constructive semantics in an abstract categorical model. Dule features mechanisms inspired by category theory, enabling fine-grained modularization without prohibitive programming overhead. Our categorical model is simple and general, admitting many variants and extensions, such as data types expressible as adjunctions, (co)inductive constructions and mutually dependent modules.

The abstract nature of our mathematical model allows us to prove fundamental properties of programming notions, such as module grouping with sharing requirements or a generalized mapping combinator [12]. On the other hand, our model is linked to program execution via combinator reduction system [9] based on equational axiomatization of the semantic model. The compiler of Dule, after type and specification reconstruction, computes the semantics of modular programs in the categorical model and verifies its type-correctness, almost literally following formal semantic definitions. Programs written in Dule compile to the language of our basic categorical model and yield the expected results through the implemented combinator reduction system. We find fascinating the immediate practical influence our theoretical results provide in this framework and the impact of implementation and experiments on the theoretical constructions.

Dule is an an experimental language with an unusual semantics and, accordingly, alien syntax. Since we cannot here define the semantics and could hardly

even fit a basic syntax description, we refrain from citing any Dule code examples. Instead we list the more interesting language features and refer the reader to the long informal tutorial of Dule available from its homepage that accompanies the Dule formal definition.

2.1 Module language features

- module composition with both transparent and non-transparent versions (the latter is the categorical composition from the semantic model)
- various module grouping mechanisms (including module categorical product operations)
- no sharing equations (nor "`with`" clauses nor type abbreviations); default sharing by names
- no substructures (submodules)
- pseudo-higher-order notation for module specifications (flattened, because no higher-order modules, but type dependencies on parameters remain)
- recursive specifications (flattened in a more complex way)
- (co)inductive modules (mutually dependent modules [8] defined using inductive types instead of recursive types)
- default implicit module composition
- specifications of transparent functor applications [26] are expressible (actually all specifications of module operations are expressible)
- compositional semantics ensures separate compilation
- no references to environments in the semantics ensures the types from module parameters are abstract; on the other hand the abstraction can be sidestepped in a controlled way and limited scope, if necessary

2.2 Core language features

- statically and strictly typed applicative language
- categorical composition (explicit substitution) available to the user; expresses, e.g., record field access
- sum types [11] separate from inductive types [18]
- both structured recursion (inductive and coinductive) and general recursion operations

- general mapping combinator precluding principal typing property
- no polymorphism nor any parameterization by types at the core language level
- no dependencies between declared core language level entities (all dependencies are expressed through modules)

2.3 Semantic model features

- elementary notion of 2-category [22] with products as the categorical model of the minimal core language
- Simple Category of Modules (SCM) constructed from the model of the core language; Dule modules denote SCM morphisms, signatures denote SCM objects, module composition is SCM composition and categorical domain lists module parameters
- in effect, our module system has set-theoretic models [31] and its construction is independent of the core language; in particular, it does not require function types nor second-order polymorphism [2] (nor dependent types [32])
- SCM is cartesian and has enough limits to model type sharing; the proof is constructive and so provides implementation of related module operations
- typed combinator reduction based on equational theory of the 2-categories provides computational meaning to modules via the core language

3 FUTURE WORK

By basing our module system on an elementary categorical model and aiming its semantics at expressing dependencies rather than enabling code-reuse [14], we have overcome what we perceive the main obstacles to practical modular programming. The operations available in our module system provide for modular programming style with no mandatory module headers, no explicit module applications and no hard to localize module errors. The model of our module system facilitates viewing the same module with different levels of abstraction. A new methodology resulting from our study of inductive types grants precise control over the concreteness of module interfaces. The grouping of modules can be done is several powerful and specialized ways, without verbose notation and without sacrificing abstraction.

The Dule abstract model and formal semantics allows us to state many interesting questions, many of which are still unanswered. Numerous programming language improvements and extensions will surely follow. In the subsequent sections we signal a few questions, problems and tasks we would like to see undertaken in

the immediate future. We are aware some of them are hard to understand out of the context of the formal language definition, but we hope they can still convey an impression of the future directions of the project.

3.1 Theory

Our elementary categorical model of the core language, on which the SCM is based, is extended with data types expressible as adjunctions, with (co)inductive constructions and with functors of mixed variance; in particular, with exponents fully parameterized by types. It would be interesting to make a detailed comparison of our construction enabling functors of mixed variance with other approaches known from literature [15] and further explore the properties of this notion.

Our module system features no higher-order modules (and even if it had them, the first-order behavior would be the default, to maintain succinct notation). We conjecture that SCM, our category of modules, has no exponent, even if its core language categorical model has all exponents and even if the definition of signatures and modules is substantially refined. It would be interesting to prove this conjecture (after stating it precisely), as well as several other conjectures related to the semantics. Yet, perhaps it is possible to add higher-order functors after all, not into the SCM, but at the outer layer of semantics? Or maybe some advanced external tools for managing libraries for code reuse — a library database and semi-automated code-adjusting engine [1] — would be a better investment of effort?

To increase the usefulness of the general mapping combinator, we have to experiment with extending its rewriting rules for the case of function types [23]. The general categorical model is already in place, just as all the necessary syntax, other rewriting rules and an analysis of rewriting of general multiplication by a functor. We would suggest that the rules are proposed first for strictly positive types, then for types with only positive occurrences of type variables and then for simple cases of mixed variance.

We expect there is a universal categorical characterization of the mapping combinator, similar, though much more general (meta-polytypic), as for the other combinators of our categorical model. Currently, the definition is by cases on all other combinators, and so it is complete only for reachable models. We do not expect the characterization to be used as a rewrite rule, but it would be very unfortunate if all such characterizations proved to be completely external to our formalism. If a characterization can be expressed in the language of our categories, it would aid in verification and systemization of the mapping combinator equations and rules, just as our general framework of adjunctions does for other combinators.

One of most interesting future extensions to our module language, the structured recursion over types produced by the (co)inductive module construction necessitates additional theoretical work. Both the programming methodology employing such recursion and the implementation issues seem to be very promising topics of research, but first we have to overcome some problems with extending the core language rewriting to basic operations typed with complex kinds.

3.2 Pragmatics

The Dule language needs a lot of additional case studies, especially concerning its module system. For serious large-scale experiments, a serious set of libraries should be designed, which themselves will be substantial case studies of the Dule modular programming style. We should explore and identify all modularization methodologies favored by our module system and analyze the impact of Dule modularization style on the core language programming.

After a form of formal specifications is chosen for the module system, the categorical model should be extended accordingly and methodologies studied once again. The specification specialization operation of our module system, suggests that the programming language code will itself be a part of the logic used for specifying modules (so that some specialized specifications could be expressed as base specifications with axioms containing code, similar to the axioms of Extended ML [24]).

We hope the ideas underlying our module system can be applied to improve other systems, in particular the OCaml module system. However, using the original Dule module system to structure OCaml programs would require an extension to the OCaml core language itself, even after the recent addition of (almost) arbitrary fixpoints to OCaml.

The notion dual to the exponent, the coexponent [13], could be used to model co-variables or exceptions shared among all values in a module. Both rewriting of coexponent and the pragmatics of co-variables would be an original topic of research. One may also check if any other adjunction expressible in our formalism is interesting from the programming point of view, or if parameterization of any other adjunction than the exponent makes sense.

We would like to improve pattern matching in Dule. In particular we would also like to implement a "three dots" notation for records and this extension should be doable at the type reconstruction level. We wonder how Haskell monads would fit into our categorical framework, in particular when used according to the methodology of merging monads and folds [29]. With monads, the Haskell list comprehensions generalized to other datatypes could possibly also be added.

3.3 Implementation

The semantics of Dule is simple and constructive enough that the design of a Dule compiler straightforwardly implementing the semantics was possible. Main parts of the compiler have also been rewritten in an extremely modular and yet concise manner in Dule itself. However, the Dule version of the compiler lacks most boring, basic tools and technical components, and is not planned to bootstrap in the near future. The example programs currently can be type-checked, compiled and executed only in the OCaml version of the Dule compiler. To help in identifying programmer's (or compiler's) errors, some rudimentary error reporting facilities

are implemented, but many more improvements, programming tools and manuals are needed to make programming in Dule comfortable.

To enable efficient recompilation, Dule would require a version of the "`make`" utility that is not tied to files but recompiles only those of the many small modules contained in a file that are changed. The implementation should also take advantage of other aspects of separate compilation in Dule, such as the possibility to salvage compiled code of all unchanged modules of a set of mutually dependent modules. We would also like to find a way to provide low-level libraries, or even arbitrary C libraries (or OCaml libraries, for a start) as a fake compiled code with a Dule-expressible specification. To enable bootstrapping of the Dule compiler written in Dule we will also need to interface a parser generator (e.g., `ocamlyacc`) to Dule.

We should improve readability of compiler reports about typing errors and efficiency of type-checking, compilation and execution. In particular, the execution of code containing arbitrary fixpoints should be improved [21]. Generally, we would like to experiment some more with the evaluation mechanism in our programming language, taking ideas from other approaches. Staying with our conventional combinator reduction, we can modify reduction rules and, for a fixed set of rules, we can try to guess which evaluation strategy is the best. Our set of rewriting rules admits many more strategies than the standard eager/lazy options of λ-calculus. As a source for additional reduction rules, we can consider the portions of the developed equational theories that are not yet used, especially the equalities of exponents, or investigate the additional equalities of the theories of initial models. Currently, despite many attempts and even hash-consing of generated code, we find a generalization of the conventional weak head normal form reduction restriction to be necessary for efficient evaluation in the presence of fixpoints.

The definition of the product of module signatures should be simplified more, without affecting the overall semantics. We may also consider implementing the linking combinator in a more conventional manner, in which the compiled modules are not composed and reduced (with possible copying of some code), but instead the compiled bodies stay separated and their functions are called whenever necessary. Perhaps a version of this idea is implementable just by changing or restricting the reduction rules for core language records.

3.4 Type reconstruction algorithms

Despite the mandatory types of values listed in specifications, Dule programs require type reconstruction. First of all, there can be locally defined values with undeclared types, and secondly, there is significant ambiguity of typing introduced by the anonymous sum and record types, the explicit (co)inductive types, the built in (co)inductive combinators and mapping. Similarly, although the preferred coding style in the module language assumes explicit specifications of almost all basic modules, the module projections, which can be separately compiled, and the various grouping operations introduce a lot of ambiguity that has to be resolved by specification reconstruction.

Our type reconstruction algorithm for core language values, unlike the classical Hindley-Milner algorithm for the ML type system, features deferred unification and uses an additional kind of type variables representing indexed lists of types. The current, implemented version of the algorithm seems to be complete, but the proof has yet to be attempted. The efficiency of the algorithm, as well as of the nonstandard substitution and unification algorithms have to be improved. Then we would like to verify that the typing reconstructed by the algorithm is minimal and study the possible choices of minimal typings from the perspective of programming practice. We would also like to prove that the complexity of the reconstruction algorithm is polynomial, assuming a constant bound on the nesting of the mapping combinator. We also conjecture the unrestricted problem is exponential, due to the thorough state space search when unifying types generated by nested mapping.

Our specification reconstruction is general and light-weight, by not using the semantics of the core language. Only the completely reconstructed least detailed specifications (if reconstruction does not fail) are checked for soundness using core language type-checking. We suspect that any specification reconstruction based on our typing rules for modules must be partial, because it has to approximate unification modulo the semantic equality (very nontrivial on the module level) by a unification with two kinds of signature variables, performed modulo a simple syntactic equality. We wonder if the algorithm can be made complete without sacrificing compositionality that is crucial for separate compilation. The main problems with the unification are that the module product is not injective and that the conditions for interchange of signature specialization with signature product are not always met. However, for natural examples the algorithm succeeds and only in very special cases the reconstruction has to be aided with spurious specification annotations.

References

[1] Anya Helene Bagge, Martin Bravenboer, Karl Trygve Kalleberg, Koen Muilwijk, and Eelco Visser. Adaptive Code Reuse by Aspects, Cloning and Renaming. Technical Report UU-CS-2005-031, Institute of Information and Computing Sciences, Utrecht University, 2005.

[2] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, January 1990.

[3] Dave Berry. Lessons from the design of a Standard ML library. *Journal of Functional Programming*, 3(4):527–552, October 1993.

[4] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273, 2002.

[5] Robin Cockett. Introduction to Distributive Categories. *Math. Struct. in Comp. Science*, pages 1–20, 1991.

[6] Robin Cockett. Charitable Thoughts, 1996. (draft lecture notes, `http://pll.cpsc.ucalgary.ca/charity1/www/home.html`).

[7] G. Cousineau, P. L. Curien, and M. Mauny. The Categorial Abstract Machine. *Science of Computer Programming*, 8:173–202, 1987.

[8] Karl Crary, Robert Harper, and Sidd Puri. What is a Recursive Module? In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, 1999.

[9] P.-L. Curien. Categorical Combinators. *Information and Control*, 69(1-3):189–254, 1986.

[10] Daniel de Rauglaudre. Camlp4 — Reference Manual, 2003. `http://caml.inria.fr/camlp4/manual`.

[11] Daniel J. Dougherty. Some lambda calculi with categorical sums and products. In Claude Kirchner, editor, *Proceedings of the 5th International Conference on Rewriting Techniques and Applications (RTA-93)*, volume 690 of *Lecture Notes in Computer Science*, pages 137–151, Berlin, June 16–18 1993. Springer-Verlag.

[12] Leonidas Fegaras and Tim Sheard. Revisiting Catamorphisms over Datatypes with Embedded Functions (or, Programs from Outer Space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

[13] Andrzej Filinski. Declarative Continuations and Categorical Duality. Technical Report 89/11, DIKU, University of Copenhagen, Denmark, 1989. Masters Thesis.

[14] Robert Bruce Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. *ACM SIGPLAN Notices*, 34(1):94–104, January 1999.

[15] Marcelo P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. PhD thesis, University of Edinburgh, 1994.

[16] Jacques Garrigue. Labeled and optional arguments for Objective Caml. In JSSST Workshop on Programming and Programming Languages, Kameoka, Japan, March 2001.

[17] Joseph A. Goguen and Will Tracz. An Implementation-Oriented Semantics for Module Composition. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 231–263. Cambridge University Press, New York, NY, 2000.

[18] Tatsuya Hagino. *A Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh, Department of Computer Science, 1987. CST-47-87 (also published as ECS-LFCS-87-38).

[19] T. Hardin. From Categorical Combinators to Lambda Sigma-Calculi, a Quest for Confluence. Technical Report No. 1777, INRIA, Le Chesnay, France, 1992.

[20] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-Order Modules and the Phase Distinction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990.

[21] Tom Hirschowitz, Xavier Leroy, and J. B. Wells. Call-by-Value Mixin Modules. In *13th European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 64–78. Springer-Verlag, 2004.

[22] C. B. Jay. An introduction to categories in computing. Technical Report UTS-SOCS-93.9, University of Technology, Sydney, 1993.

[23] C. B. Jay. Covariant types. *Theoretical Computer Science*, 185:237–258, 1997.

[24] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445–484, 28 February 1997.

[25] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st symp. Principles of Programming Languages*, pages 109–122. ACM press, 1994.

[26] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd symp. Principles of Programming Languages*, pages 142–153. ACM Press, 1995.

[27] Xavier Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

[28] Erik Meijer and Graham Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Conf. Record of 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 324–333. ACM Press, New York, 1995.

[29] Erik Meijer and Johan Jeuring. Merging Monads and Folds for Functional Programming. In J. Jeuring and E. Meijer, editors, *Tutorial Text 1st Int. Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, 24–30 May 1995*, volume 925, pages 228–266. Springer-Verlag, Berlin, 1995.

[30] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[31] John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156, Berlin, 1984. Springer-Verlag.

[32] Claudio V. Russo. Non-dependent Types for Standard ML Modules. In *Principles and Practice of Declarative Programming*, pages 80–97, 1999.

[33] Perdita Stevens. Experiences with the ML Module System, or, Why I Hate ML. Transparencies for a talk given to the Edinburgh ML Club and Glasgow Functional Programming group. `http://www.dcs.ed.ac.uk/home/pxs/talksEtc.html`, 1998.