Streszczenie rozprawy doktorskiej pod tytułem:

Application of Category-Theory Methods to the Design of a System of Modules for a Functional Programming Language

(Wykorzystanie metod teorii kategorii do opracowania systemu modułów dla języka programowana funkcyjnego)

Mikołaj Konarski

Systemy modułów wbudowują do wewnątrz samych języków programowania elementy metodologii, formalizmów i narzędzi, służących programowaniu w dużej skali. W naszej rozprawie projektujemy taki system modułów, jego kategoryjny model i konstruktywną semantykę owego systemu w tym modelu. Nasz kategoryjny model jest prosty i ogólny, zbudowany używając elementarnych pojęć teorii kategorii, dopuszczający wiele wariantów i rozszerzeń systemu modułów. Zaprojektowany system modułów charakteryzuje się mechanizmami programistycznymi inspirowanymi przez teorię kategorii i umożliwia drobnoziarniste programowanie modularne, jak świadczą zaprezentowane eksperymenty z użyciem opartego na nim języka programowania Dule.

Pokazujemy, jak rozszerzyć nasz kategoryjny model (i w konsekwencji cały język programowania) o typy danych wyrażalne jako sprzężenia w 2-kategoriach, jak również o konstrukcje (ko)indukcyjne i w pełni sparametryzowane obiekty wykładnicze. Analizujemy i porównujemy alternatywne aksjomatyzacje tych rozszerzeń i pokazujemy szereg własności przepisywania termów oznaczających morfizmy 2-kategorii; w szczególności termów wyrażających ogólne sprzężenia.

Na podstawie rozszerzonego modelu i bez wprowadzania rekurencyjnych typów definiujemy mechanizm wzajemnej zależności pomiędzy modułami w postaci konstrukcji (ko)indukcyjnego modułu. Rozszerzony system modułów zawiera zasadnicze mechanizmy modularnego programowania, takie jak dzielenie typów, przezroczystą jak i nieprzezroczystą aplikację funktorów oraz grupowanie powiązanych modułów. Popularne modułowe konstrukcje są wyrażalne na nowe i oryginalne sposoby, umożliwiając modularne programowanie bez nagłówków, bez aplikacji modułów i bez globalnych modułowych błędów.

Cel

Celem naszych badan było znalezienie dobrego matematycznego modelu dla systemu modułów w stylu Standard ML [28]. Taki model powinien prowadzić do intuicyjnej semantyki i w konsekwencji umożliwiać programiście stosunkowo łatwe

rozumienie i przewidywanie działania systemu modułów. Powinien on wspierać drobnoziarnistą modularyzację, bez nieakceptowalnego narzutu programistycznej pracy. W szczególności, operowanie wymaganiami dzielenia typów [24] powinno być łatwe i znajdowanie błędów w modularnych kodzie powinno być szybkie.

Matematyczny model powinien odzwierciedlać wszystkie ważne aspekty modularnego programowania: konstrukcję modułów z wyrażeń języka programowania i różnorodne postacie grupowania, instancjonowania i dostępu do definiowanych modułów. Semantyka powinna być dość prosta [31], by zapobiegać pomyłkom i paradoksom w rozumowaniu o modułach, szczególnie tych z wieloma współzależnymi parametrami. Zestaw operacji powinien zapewniać pełną kontrolę nad subtelnościami hierarchii modułów, jednocześnie umożliwiając zwięzły zapis najczęstszych konstrukcji. Własności modelu powinny gwarantować adekwatność, abstrakcję, rozłączną kompilację i dobrze umiejscowione zgłaszanie błędów [21].

Model systemu modułów powinien również dawać wgląd w naturę fundamentalnych modułowych mechanizmów, takich jak przezroczysta i nieprzezroczysta aplikacja funktorów [25], grupowanie powiązanych modułów i wzajemna zależność modułów [10]. Płytkie, obliczeniowe rozumienie popularnych modułowych operacji powinno być wzbogacone i usystematyzowane przez odtworzenie tych operacji wewnątrz abstrakcyjnego semantycznego modelu. Ten model nie powinien być syntaktycznych rachunkiem, imitującym obliczeniowe zachowanie konwencjonalnych operacji modułowych, ale niezależną, abstrakcyjną, matematyczną konstrukcją. Z drugiej strony, taki model powinien być klarowny, prosty i wyposażony w naturalną obliczeniową interpretację, tak aby rozumowanie o modularnych programach i projektowanie wyspecjalizowanych technik kompilacji dla nich było łatwe.

Pożytek

Zwrot w kierunku modularnego programowania jest konieczny z powodu rosnących rozmiarów, komplikacji i wymogu modyfikowalności programów komputerowych. Najbardziej użytecznym podejściem wydaje się być połączenie rygorystycznego systemu modułów z językiem programowania wysokiego poziomu, jak to jest w językach Standard ML lub OCaml [26]. Jednakże, ściśle modularny styl programowania może być utrzymany jedynie w małych projektach programistycznych pisanych w którymkolwiek z tych dwóch języków. W dużych projektach, operowanie wieloma wprowadzonymi przez hierarchię modułów poziomami abstrakcji okazuje się trudniejsze, niż choćby ręczne śledzenie każdej poszczególnej zależności w nieuporządkowanym kodzie [32].

Naszym zdaniem, aby mieć szansę użyteczności w każdej skali, system modułów musi posiadać dobrze zaprojektowany i zwarty model. Taki model powinien

być bardzo prosty, aby nie akumulował złożoności modularnych zależności w rosnącym programie. Jednocześnie, powinien być on wystarczająco silny, aby ta upraszczająca abstrakcja mogła być ominięta w kontrolowany sposób, jeśli jest to konieczne. Dobry lukier syntaktyczny, domyślne konwencje i narzędzia programistyczne nie wystarczają, by system modułów był użyteczny. Przystępny i sugestywny model jest konieczny, tak aby wykształceni programiści mogli myśleć w terminach modelu. Ma to zasadnicze znaczenie — co najmniej w procesie znajdowania błędów w kodzie.

Trudności

Programowanie wewnątrz modułu jest dobrze znanym i zazwyczaj łatwym zadaniem. Jednak abstrakcyjne i drobnoziarniste modularne programowanie jest ciężką i niewdzięczną pracą, ponieważ nagłówki, które trzeba pisać, są skomplikowane, a aplikacje modułów często wywołują globalne błędy dzielenia typów. Kiedy moduł ma n parametrów, $O(n^2)$ potencjalnych konfliktów typowania czycha przy każdej aplikacji, czyniąc nie tylko tworzenie modułów, ale również ich używanie i pielęgnację, bardzo kosztownymi.

Kolejnym problemem jest napięcie pomiędzy abstrakcyjnością, wyrażalnością i użytecznością modułu. W uproszczeniu: im bardziej abstrakcyjna jest specyfikacja modułu, tym łatwiej go zaimplementować, ale tym trudniej go używać. Bez wyspecjalizowanych mechanizmów systemu modułów, to globalne napięcie może być rozwiązane poprzez uczynienie każdego modułu przesadnie silnym i ogólnym. Jednakże, najczęstszym praktycznym podejściem jest stopniowe rezygnowanie z abstrakcji, w miarę jak program rośnie, podczas gdy, szczególnie dla zapewnienia poprawności dużych programów, abstrakcja jest kluczowa [5].

Olbrzymie kolekcje współzależnych modułów same wymagają modularyzacji, gdyż w przeciwnym razie zarządzanie modułami staje się tak nużące, jak doglądanie mrowia indywidualnych bytów w niezmodularyzowanych programach. Grupowanie modułów przeprowadzane używając mechanizmu podmodułów często przytłacza programistę biurokracją dzielenia typów, albo wymaga pogwałcenia gwarancji abstrakcyjności. Inne rozwiązania, niektóre z nich używające zewnętrznych narzędzi, są zwykle toporne i wprowadzają ryzyko pomieszania zakresów nazw lub ignorują abstrakcję.

Rezultaty

Konstrukcja głównego rezultatu rozprawy — matematycznego modelu modularnego języka programowania — przebiega w dwóch etapach. Po pierwsze, proponujemy pewną abstrakcję języka jądra (języka bez modułów), używając elementarnych pojęć produktu i 2-kategorii. Potem, nad tą abstrakcją (2-kategoryjnym modelem języka jądra) definiujemy model systemu modułów: Prostą Kategorię

Modułów. Dowodzimy jej własności i podajemy semantykę podstawowych modularnych operacji. Taki układ powoduje, że nasz system modułów korzysta ze wszystkich prowadzonych następnie badań nad 2-kategoryjnym modelem języka jądra, jego obliczeniowym znaczeniem i jego rozszerzeniami.

W pierwszym etapie rozwijamy pojęcie 2-kartezjańskiej kategorii, jako podstawy naszego kategoryjnego warsztatu. To pojęcie leży u podstaw każdego modelu języka jądra i systemu modułów rozwijanych w naszej rozprawie. Używając wyłącznie produktów, modelujemy zmienne oznaczające typy i zmienne oznaczające wartości. Pozwalają one wyrażać zależności typów języka programowania od typów, wartości od typów i wartości od wartości.

W drugim etapie wyrażamy moduły programistyczne w naszym podstawowym 2-kategoryjnym modelu i pokazujemy, że zbiór modułów wraz ze złożeniem modułów tworzą kategorię — Prostą Kategorię Modułów (SCM). Potem, zauważamy, że SCM jest kartezjańska i dzięki temu możemy uchwycić zależności między wieloma modułami na raz. Co więcej, istnieje dość equalizatorów (granic) w SCM, aby modelować specyfikacje dzielenia typów. Do naszego systemu modułów wybieramy taki mechanizm dzielenia typów, że wraz z ogólną prostotą kategoryjnych modułowych operacji eliminuje on biurokrację dzielenia typów.

Kompozycjonalność semantyki naszego systemu modułów w SCM zapewnia rozłączną kompilowalność modułów. Brak jakichkolwiek odniesień do środowisk w naszych definicjach semantyki powoduje całkowitą abstrakcyjność parametrów modułów. Jednakże, kiedy moduł jest złożony ze swoimi argumentami, abstrakcja może zostać przekroczona przy użyciu odpowiednich modularnych operacji. Konstrukcja SCM nad bardzo ogólną klasą 2-kategorii pokazuje, że nasz system modułów ma teorio-mnogościowy model [29], i zapewnia, że konstrukcja naszego systemu modułów jest stosunkowo niezależna od języka jądra, w szczególności nie wymaga typów funkcyjnych, ani polimorfizmu drugiego rzędu [3].

W naszej rozprawie konstruujemy teorie równościowe i projektujemy konfluentne i silnie normalizowalne typowane systemy redukcyjne dla szeregu rozszerzeń naszego postawowego 2-kategoryjnego modelu. W szczególności, podajemy aksjomatyzację dla kategorii 2-kartezjańskich, konstruujemy dla nich system redukcyjny i pokazujemy poprawność systemu względem tej aksjomatyzacji. Również dla 2-kartezjańskich kategorii rozszerzonych o typy sumowe [12] konstruujemy system redukcyjny i dowodzimy jego konfluencji i silnej normalizacji. Podajemy dwie różne aksjomatyzacje dla tej klasy kategorii. Pokazujemy poprawność i pełność pierwszej spośród aksjomatyzacji względem opisywanej klasy kategorii, poprawność systemu redukcyjnego względem drugiej aksjomatyzacji oraz, co wymaga najtrudniejszego dowodu, równoważność obu aksjomatyzacji.

Konstruujemy również system redukcyjny dla naszych 2-kategorii rozszerzonych o typy (ko)indukcyjne [18] z kombinatorami rekurencji strukturalnej [6], które mogą służyć jako główny obliczeniowy mechanizm języka jądra. Dowo-

dzimy wielu własności tego systemu, ale na niektóre fundamentalne pytania znajdujemy tylko częściowe odpowiedzi. Rozszerzenie o typ funkcyjny (obiekt wykładniczy parametryzowalny typami zarówno na kowariantnej, jak i na kontrawariantnej pozycji) jest jeszcze trudniejsze [16], więc najpierw rozwijamy pojęcie sparametryzowanego sprzężenia, które dopuszcza jako szczególne przypadki oraz uogólnia operacje naszego postawowego 2-kategoryjnego modelu. Przy pomocy najbardziej wyrafinowanego spośród naszych formalizmów dla sprzężeń jesteśmy w stanie uchwycić w pełni sparametryzowane obiekty wykładnicze, jak również określić i odizolować miejsca, w których ich kontrawariancja kłóci się [22] z (ko)indukcyjnymi typami, powodując problemy z redukcją kombinatora `map` [15].

Używamy sparametryzowanych sprzężeń również do systematyzacji i analizy naszych teorii równościowych i systemów redukcyjnych. W szczególności, proponujemy reguły spokrewnione z η-równoważnością i wyprowadzamy zestaw reguł do redukcji przemnażania przez funktory sprzężone. Algebraiczna sygnatura rozszerzonego 2-kategoryjnego modelu (widzianego jako algebra) wraz z systemem redukcyjnym opartym na jego teorii równościowej jest sama z siebie interesującym językiem programowania. Aby ułatwić programowanie w tym języku, proponujemy nieco lukru syntaktycznego i konwencji notacyjnych, podajemy syntaktyczne reguły typowania i dowodzimy ich własności; na przykład, że każdy typ wyprowadzony tymi regułami jest wyrażalny przez programistę jako zamknięte wyrażenie typowe.

Na postawie naszego rozszerzonego 2-kategoryjnego modelu dodajemy wzajemnie zależne moduły do SCM. W ich konstrukcji jesteśmy w stanie uniknąć mechanizmu typów rekurencyjnych, używając typów (ko)indukcyjnych do rekurencyjnego domknięcia typowej części modułów. Użycie typów (ko)indukcyjnych komplikuje konstrukcję, ale umożliwia jej weryfikację we względnie bezpiecznym formalizmie typowym oraz otwiera drogę do strukturalnej rekurencji [27] względem typów zdefiniowanych w modułach. Aby uczynić system modułów modelowany przez SCM używalnym przez programistę, dodefiniowujemy dodatkowe modułowe operacje używając języka SCM, umożliwiamy notację pseudo-wyższegorzędu, wprowadzamy rekurencyjne sygnatury i podajemy reguły wyprowadzania sygnatur dla modułów, dowodząc, że wszystkie wyprowadzone sygnatury są wyrażalne przez programistę. Przez połączenie języka jądra opartego na rozszerzonym 2-kategoryjnym modelu oraz języka modułów opartego na rozszerzonej SCM powstaje język programowania Dule.

Implementacja

Modularny język programowania Dule, wyłaniający się z naszych badań, okazuje się być dość bogatym językiem programowania, dość różnym od innych języków. Z powodu swojej oryginalności, Dule wciąż ma wiele niewygładzonych krawędzi.

Jednakże, jesteśmy przekonani, że pokonaliśmy to, co uważamy za główne przeszkody praktycznego modularnego programowania. Operacje dostępne w naszym systemie modułów dostarczają środków do modularnego programowania bez obowiązkowych nagłówków modułów, bez jawnych aplikacji modułów i bez trudnych do zlokalizowania błędów modułowych. Model naszego systemu modułów umożliwia widzenie tego samego modułu z perspektywy różnych poziomów abstrakcji. Nowa metodologia, wynikająca z naszych badań nad typami indukcyjnymi, daje precyzyjną kontrolę nad konkretnością sygnatur modułów. Grupowanie modułów może być wykonywane na kilka silnych i wyspecjalizowanych sposobów, bez nadmiernie szczegółowej notacji i bez rezygnowania z abstrakcji.

Semantyka Dule okazała się wystarczająco prosta i konstruktywna, aby umożliwić zaprojektowanie kompilatora Dule, bezpośrednio implementującego tę semantykę: zobacz stronę Dule pod adresem `http://www.mimuw.edu.pl/~mikon/dule.html`. W rezultacie, nasza rozprawa może być postrzegana jako dowód poprawności kompilatora (poprawności typowej generowanego kodu i innych fundamentalnych własności). Z drugiej strony, język Dule okazał się wystarczająco silny, by umożliwić napisanie, w ekstremalnie modularny a jednak zwięzły sposób, różnorakich przykładowych programów, włączając w to główne części samego kompilatora Dule. Wersja kodu kompilatora, przetestowana względem przykładów w Dule zawartych w naszej rozprawie, jest utrzymywana pod adresem `http://www.mimuw.edu.pl/~mikon/Dule/dule-phd/`. Dalsze wielkoskalowe eksperymenty przy użyciu drobnoziarnistego modularnego programowania, umożliwione przez nasz system modułów, powinny odsłonić jakiekolwiek pozostałe problemy wynikające z naszej metodologii i zasugerować usprawnienia w praktycznej użyteczności. Elastyczności i rozszerzalność naszego systemu modułów dają nadzieję, że jego przyszłe warianty będą dojrzewać, używane w praktycznych wielkoskalowych projektach.

W naszej rozprawie nie opisujemy kompilatora Dule, ale większość jego kodu jest bezpośrednim zakodowaniem konstruktywnej semantyki Dule w abstrakcyjnym kategoryjnym modelu, rozwiniętym w naszej rozprawie. Nasz kompilator, po rekonstrukcji typów i sygnatur, oblicza semantykę modularnych programów w kategoryjnym modelu i weryfikuję jej poprawność typową, prawie dosłownie naśladując formalne semantyczne definicje. Wszystkie przykładowe programy podane w naszej rozprawie kompilują się z powodzeniem do języka naszego 2-kategoryjnego modelu i dają oczekiwane rezultaty, przez redukcję kombinatorów [11] zaprojektowaną w naszej rozprawie i zaimplementowaną w kompilatorze. Fascynujący wydaje nam się natychmiastowy praktyczny użytek, jaki nasze teoretyczne rezultaty znajdują w konstrukcji kompilatora Dule, oraz pomoc, jaką zakodowanie w języku programowania, testowanie i eksperymenty, stanowią w weryfikacji matematycznych modeli i upraszczaniu oraz doregulowywaniu semantyki.

Literatura

[1] J. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Processes. *Communications of the ACM*, 21(8):613–641, August 1978.

[2] 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.

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

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

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

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

[7] Robin Cockett and Dwight Spencer. Strong Categorical Datatypes I. In R. A. G. Seely, editor, *Proc. of Int. Summer Category Theory Meeting, Montréal, Québec, 23–30 June 1991*, volume 13 of *Canadian Mathematical Society Conf. Proceedings*, pages 141–169. American Mathematical Society, Providence, RI, 1992.

[8] Robin Cockett and Dwight Spencer. Strong Categorical Datatypes II. A term logic for categorical programming. *Theoretical Computer Science*, 139(1–2):69–113, March 1995.

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

[10] 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.

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

[12] 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.

[13] Derek Dreyer. Understanding and Evolving the ML Module System. PhD thesis, CMU Technical Report CMU-CS-05-131, May 2005.

[14] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation.* PhD thesis, DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark, December 1998. Available as Tech. Report 99/3.

[15] 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.

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

[17] N. Ghani. Beta-Eta-Equality for Coproducts. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Typed Lambda Calculi and Applications: Proc. of the 2nd International Conference on Typed Lambda Calculi and Applications (TLCA-95)*, pages 171–185. Springer, Berlin, Heidelberg, 1995.

[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. Confluence results for the pure strong categorical logic CCL lambda-calculi as subsystems of CCL. *Theoretical Computer Science*, 65(3):291–342, July 1989.

[20] Robert Harper and Mark Lillibridge. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[21] 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.

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

[23] C. B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

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

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

[26] 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`.

[27] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In John Hughes, editor, *Proceedings of Functional Programming Languages an Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Berlin, Germany, August 1991. Springer.

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

[29] 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.

[30] Claudio V. Russo. Types for Modules. PhD thesis, Univ. of Edinburgh, 1998.

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

[32] 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.

[33] Niklaus Wirth. *Programming in Modula-2 (3rd corrected edition)*. Springer-Verlag, New York, NY, 1985.