Obiektowy język programowania oparty na mixinach Prawdziwa modularność w językach obiektowych

Autoreferat pracy doktorskiej

Jarosław Dominik Mateusz Kuśmierek

15 lutego 2010

1 Sformułowanie problemu

W dzisiejszych czasach, większość tworzonego oprogramowania (szczególnie tego tworzonego komercyjnie) powstaje z użyciem języków programowania zaliczanych do kategorii zwanej językami obiektowymi. Dla języków tych stworzono także bardzo duży zestaw narzędzi wspomagających sam proces wytwarzania programowania.

Języki te wciąż jednak posiadają pewne ograniczenia, które są tematem wielu prowadzonych badań.

Jednym z ważniejszych obszarów badań nad rozwojem języków obiektowych jest tzw *modularyzacja* i *kompozycja*. Przez *modularyzację* rozumiemy tu możliwość podziału programu na mniejsze komponenty, które umożliwiają później elastyczne budowanie większych części programu. Przez *kompozycję* rozumiemy tu wspomnianą przed chwilą możliwość budowania nowych elementów programu poprzez składanie ze sobą wcześniej stworzonych komponentów. Podstawowym elementem modularyzacji w wiodących[1] językach obiektowych jest pojęcie klasy. Jednak klasa jako narzędzie modularycji posiada wiele ograniczeń (wspomniane poniżej).

Ze względu na ograniczenia w późniejszym wykorzystaniu klas jako komponentów programu powstało wiele rozszerzeń i wariantów obiektowych języków programowania. Jednak żadne z tych rozszerzeń nie uzyskało szerokiej akceptacji. Moim zdaniem, jest to zazwyczaj spowodowane znacznym wzrostem komplikacji języka wprowadzanym przez takie rozszerzenie, co z kolei powoduje opór programistów przed sprawnym przyswajaniem takich rozszerzeń.

Pierwsza część rozprawy doktorskiej zawiera opis wspomnianych ograniczeń w językach obiektowych. Następnie zawiera prezentację nowego języka programowania zwanego Magda, który rozwiązuje postawione problemy. Język ten jest opisany poprzez przedstawienie szeregu przykładowych programów, od najprostszych do wykorzystujących bardziej zaawan-

[1] Jako wiodące rozumiemy te, z których korzysta największa liczby programistów.

sowane elementy języka. Następnie zaprezentowana jest kompletna składnia języka z opisem znaczenia wszystkich konstrukcji. Kolejna część zawiera formalny opis semantyki dużych kroków języka. Następna część zawiera system przypisywania typów dla wszystkich elementów języka Magda. W kolejnej części dowodzi się że system typów jest poprawny. Dowód ten jest przeprowadzony dla silniejszej definicji poprawności niż tej, która stosowana jest zazwyczaj dla języków opisanych semantyką dużych kroków. Ostatnia część zawiera krótki opis implementacji języka oraz porównanie tej pracy z wynikami istniejącymi w literaturze.

2 Mixiny

Kluczowym pojęciem w języku programowania Magda jest *mixin*. Pojęcie to zostało wprowadzone ponad 20 lat temu w językach takich jak Flavors [56], oraz Jigsaw [20] i nastepnie w wielu innych językach takich jak rozszerzenia Javy (MixedJava [36], MixGen [6]). Mixin jest elementem modularyzacji kodu, który może zawierać zestaw deklaracji metod, pól (a w niektórych implementacjach także konstruktorów) w sposób podobny do deklaracji klasy w klasycznych językach obiektowych.

Jednak mixin rózni sie od klasy sposobem w jaki określana jest nadklasa deklarowanej właśnie klasy. Deklaracja zwykłej klasy C posiada jednoznaczne wskazanie na jedną nadklasę Y, określające że klasa C powstaje przez dodanie występujących w niej deklaracji do tych istniejących w zadeklarowanej nadklasie Y.

Natomiast deklaracja mixinu nie określa jednoznacznie klasy, którą rozszerza. Dlatego też, jeden mixin M zawierający pewne deklaracje może być stosowany wielokrotnie, przez aplikowanie do wielu klas, uzyskując w efekcie podklasy istniejących klas rozszerzone o deklaracje metod i pól wystepujące w M. W ten sposób mixin może być interpretowany jako funkcja wyższego rzędu. Co więcej, deklaracje mixinów mogą być ze sobą składane tworząc wiele nowych mixinów i klas. W efekcie mixiny dają dużo większą elastyczność modularyzacji i kompozycji elementów programu niż klasy.

Niestety, mechanizm kompozycji mixinów ma pewne słabe punkty. Jednym z nich jest zachowanie w przypadku, w którym mixin zawierający deklarację nowej metody m, jest aplikowany do klasy, która zupełnie niezależnie zawiera także metodę o nazwie m. Taka kolizja nazw jest rozstrzygana przez różne implementacje na różne sposoby. Niektóre implementacje zgłaszają błąd, niektóre przyjmują deklarację m z aplikowanego mixina jako tę obowiązującą. Niestety żadne z podejść do tego problemu nie jest dobrym rozwiązaniem w ogólnym przypadku. Dlatego też zaczęto szukać innych rozwiązań (takich jak mechanizm *traits*), które jednak zwiększają stopień komplikacji języka.

Dlatego też w języku Magda wprowadziliśmy rozwiązania (opisane w skrócie w rozdziale 4) które powodują że zjawisko konfliktów deklaracji nie istnieje.

3 Inicjalizacja obiektów

Dodatkowym ograniczeniem istniejących języków programowania jest podejście do inicjalizacji obiektów. W klasycznych językach sposób inicjalizowania obiektu (w tym lista danych potrzebnych do poprawnego zainicjowania obiektu) są podawane w postaci monolitycznych konstruktorów. Każdy konstruktor zawiera pełną listę parametrów, które muszą zostać dostarczone przez użytkownika danej klasy aby stworzyć obiekt. Dodatkowo każdy konstruktor musi zawierać jednoznaczne wskazanie konstruktora nadklasy, który zostanie wykonany gdy dany konstruktor podklasy zakończy swoją pracę.

W sytuacji, w której do języka dodawana jest konstrukcja mixina, takie podejście dodatkowo ogranicza użycie mixinów, gdyż konstruktor w mixinie musi wprost znać listę parametrów konstruktora klasy, do której będzie stosowany. Często nie jest to możliwe i dlatego niektóre implementacje mixinów odrzuciły w ogóle możliwość deklarowania konstruktorów.

Monolityczna budowa konstruktorów posiada jeszcze jedną wadę. Jeśli stworzenie obiektu z pewnej klasy wymaga dostarczenia kilku niezależnych zbiorów informacji i każdy z nich zawiera kilka akceptowalnych formatów lub może być dostarczony opcjonalnie, to liczba wszystkich kombinacji tych danych rośnie wykładniczo a wraz z nią liczba konstruktorów klasy i w efekcie rozmiar programu.

Dlatego też w języku Magda pojęcie konstruktora zostało zastąpione bardziej elastycznym pojęciem modułu inicjalizacyjnego (opisanym w rozdziale 4 autoreferatu).

4 Język programowania Magda

W języku Magda podstawowym elementem programu jest mixin. Podobnie jak w innych językach zawierających tę konstrukcję, mixiny zawierają deklaracje nowych metod, jak i redefinicje istniejących. Jednak, w odróżnieniu od klasycznych języków programowania stosuje inny sposób odwoływania się do raz zadeklarowanych identyfikatorów (nazw metod i pól obiektu). Każde odwołanie do metody jest prefiksowane mixinem z którego ta metoda pochodzi. Przykład takiego prefiksowania jest widoczny na rysunku 1. Przykład ten zawiera deklaracje dwóch mixinów: `Mixin1` i `Mixin2`, obu zawierających metody o tej samej nazwie: `Method`. Następnie, w ostatniej instrukcji, program tworzy nowy obiekt z tych dwóch mixinów i wywołuje metodę zadeklarowaną w pierwszym z nich.

W ten sposób, mimo że oba mixiny zawierają metody o tej samej nazwie, nie powoduje to konfliktu i obie metody są dostępne niezależnie. Analogicznie, jeśli mixin zawiera redefinicję metody zdefiniowanej w innym mixinie, redefinicja ta odwołuje się nie tylko do samej nazwy metody, ale także do nazwy mixinu — tak jak to zostało przedstawione na rysunku 2.

Deklaracje mixinów w Magdzie mogą zawierać także opis sposobu inicjalizacji obiektów. Opis ten nie jest jednak realizowany klasycznie z użyciem konstruktorów. Zamiast tego w Magdzie wprowadzone zostało pojęcie modułu inicjalizacyjnego (patrz rysunek 3). Każdy taki moduł opisuje tylko fragment procesu inicjalizacji obiektu.

Każdy moduł zawiera listę parametrów wejściowych zbudowaną z ich nazw i typów.

```
mixin Mixin1 of Object=
  new void Method()
  begin
    "Method from Mixin1 ".String.print();
  end;
end;

mixin Mixin2 of Object =
  new void Method()
  begin
    "Method from Mixin2 ".String.print();
  end;
end;

(new Mixin1, Mixin2[]).Mixin1.Method();
```

Rysunek 1: Odwołania do metod w Magdzie

```
mixin Mixin3 of Mixin1=
  override void Mixin1.Method()
  begin
    "Redefinition of method from Mixin1 ".String.print();
  end;
end;
```

Rysunek 2: Redefinicje metod w Magdzie

Podczas tworzenia nowego obiektu (patrz ostatnia linia przykładu na rysunku 3) użytkownik podaje listę nazw parametrów wraz z ich wartościami. Na tej podstawie wybierana jest lista modułów, które mogą pochodzić z różnych mixinów.

Dodatkowo, moduł inicjalizacyjny może wpływać na parametry przekazywane do innych modułów. Jednak wtedy moduł odwołuje się jedynie do tych parametrów które modyfikuje i nie zawiera żadnych odwołań do pozostałych parametrów. Moduł, który inicjalizuje parametry innego modułu jest widoczny na rysunku 4. Na przykładzie tym można zauważyć, że moduł ten odnosi się jedynie do parametrów `x` i `y` mixina `CPoint`. Moduł ten wprowadzając dwa nowe parametry zastępujące dwa istniejące, pozostawia pozostałe bez zmian.

Dodatkowo, podział pracy związanej z inicjalizacją obiektu na mniejsze części jakimi są moduły inicjalizacyjne powoduje uniknięcie problemu wykładniczej eksplozji rozmiaru kodu występującego przy klasycznych konstruktorach.

W efekcie, zastosowany mechanizm odwoływania się do metod powoduje, że podczas składania ze sobą wielu mixinów nie może dojść do konfliktów nazw. Deklaracje modułów inicjalizacyjnych nie powodują także takich ograniczeń jak klasyczne konstruktory. Dlatego też uważam, że mixiny w wersji dostępnej w Magdzie są elastycznymi i uniwersalnymi

```
mixin CPoint of Object =
  fx:Integer; fy:Integer;
  fr:Integer; fg:Integer; fb:Integer;

  required CPoint(x:Integer; y:Integer) initializes ()          // moduł 1
  begin
    this.Point2D.fx := x;
    this.Point2D.fy := y;
    super[];
  end;

  required CPoint(r:Integer; g:Integer; b:Integer) initalizes() // moduł 2
  begin
    this.Point2D.fr := r;
    this.Point2D.fg := g;
    this.Point2D.fb := b;
  end;
end;

new ColorPoint [CPoint.x:=1, CPoint.y:=0, CPoint.r:=255,
                CPoint.g:=255, CPoint.b:=255];
```

Rysunek 3: Inicjalizacja obiektów w Magdzie

```
mixin PolarPoint of CPoint =
  fx:Integer; fy:Integer;
  fr:Integer; fg:Integer; fb:Integer;

  required PolarPoint(rad:Float, angle:Float) initializes (CPoint.x, CPoint.y)
  begin
    super[CPoint.x := cos(angle) * rad, CPoint.y := sin(angle) * rad];
  end;
end;

new PolarPoint [PolarPoint.rad:=20, PolarPoint.angler := 45, CPoint.r:=255,
                CPoint.g:=255, CPoint.b:=255];
```

Rysunek 4: Inicjalizacja obiektów w Magdzie - część 2

narzędziami modularyzacji i kompozycji kodu.

5 Semantyka, system typów i ich poprawność

W rozprawie doktorskiej semantyka języka Magda jest opisana jako semantyka dużych kroków. Daje to czytelny opis mechanizmów rekurencyjnych wywołań i innych elementów programu.

Niestety, semantyka dużych kroków posiada ograniczoną możliwość wyrażenia jej poprawności względem systemu typów. Zazwyczaj stosowane w takich sytuacjach definicje poprawności są ograniczone do stwierdzenia, że jeśli program przejdzie poprawnie kontrolę typów i zakończy swoje działanie (czyli istnieje wyprowadzenie dla formuły zdaniowej stwierdzającej, ze program X ewaluuje się do stanu i wartości Y) to jego wynik i stan na koniec działania spełnia pewne warunki (np. jest zgodny z deklaracją typu).

Taka własność jest dosyć słaba, ponieważ nie chroni przed tym, że program może w dowolnym momencie swojego działania ugrzęznąć, czyli dojść do punktu w którym nie da się zastosować żadnej reguły semantyki. Takie grzęźnięcie w semantyce dużych kroków sprowadza się do tego, że brak jakiegokolwiek wyprowadzenia dla formuły mówiącej że uruchomienie programu prowadzi do jakiejś wartości.

Z drugiej strony fakt, że brak wyprowadzenia dla formuły "program X kończy swoje wykonanie w stanie Y" nie musi oznaczać grzęźnięcia lub błędu. Brak takiego wyprowadzenia może także oznaczać zapętlenie programu, które jest w większości języków akceptowalnym zachowaniem programu i nie świadczy o błędzie w systemie typów.

Dlatego w pracy wprowadziliśmy pojęcie *śladu programu* (w oryginale: *trace*), który jest ciągiem konfiguracji programu. Kolejne konfiguracje w śladzie odpowiadają w pewnym sensie kolejnym stanom programu, lub inaczej: przechodzeniu po kolejnych węzłach drzewa wyprowadzenia w semantyce dużych kroków. Zaleta analizowania ciągu konfiguracji jest taka, że program, dla którego nie istnieje drzewo wyprowadzenia posiada ślad, który od śladu programu wykonującego się w całości różni się tylko tym, że ten pierwszy ślad kończy się w stanie nie reprezentującym zakończenia wykonywania programu. W tym sensie, ślad odpowiada sekwencji ewaluacji programu w semantyce małych kroków.

Dzięki temu, poprawność systemu typów została dla Magdy sformalizowana w następującej postaci: Jeżeli dany program przechodzi poprawnie kontrolę typów, to zachodzą dla niego dwie właśności. Po pierwsze, każdy stan w jego śladzie jest zgodny z zadeklarowanymi typami zmiennych i pól obiektów. Po drugie, jeśli ślad tego programu nie jest ciągiem nieskończonym to albo zatrzymuje się na odwołaniu do wskaźnika, którego wartość jest niezainicjowana (`null`) albo zatrzymuje się na końcu treści programu. Tej drugiej własności, nie dało się wyrazić w klasycznym podejściu i gwarantuje ona że program nie zatrzyma się z powodu żadnego innego błędu niż odwołanie do wartości `null`. Dodatkowo, pojęcie śladu pozwala dodatkowo wnioskować o programach, które z definicji nie kończą nigdy swojego działania.

6 Publikacje

Część z wyników prac nad tematem rozprawy doktorskiej została opublikowana niezależnie.

Dla opisanego języka Magda powstał kompilator, który dostępny jest w [47]. Kompilator ten zawiera mechanizm kontroli typów wg opisanego w rozprawie mechanizmu przypisywania typów. Po poprawnym przejściu weryfikacji, program ten przetwarzany jest na program w Javie, który następnie jest kompilowany standardowym kompilatorem języka Java.

Dodatkowo, częściowe wyniki badań zostały opublikowane na konferencjach zajmujących się tematyką obiektowych języków programowania. Dotyczy to między innymi mechanizmu odwołań do metod bez wieloznaczności i konfliktów, zwanego "higienicznymi identyfikatorami". W pracy [48] mechanizm ten został zaprezentowany w oderwaniu od języka Magda — w formie rozszerzenia języka Java zwanego HygJava.

Opis mechanizmu modularnej inicjalizacji obiektów także został umieszczony w kilku publikacjach. W pracy [18] mechanizm ten został zaprezentowany w formie rozszerzenia języka Java zwanego JavaMIP. Dla języka tego zbudowany został także kompilator [55]. Opublikowana została także praca na temat fomalnego rachunku FJMIP rozszerzającego rachunek FJ (Featherweight Java [42]) o mechanizm modularnej inicjalizacji obiektów. Dowód poprawności tego rachunku został zawarty w pracy [17].

Dodatkowo, nowe podejście do polimorfizmu możliwe dzięki mechanizmom dostępnym w Magdzie zostało opisane w pracy [46].

Literatura

[1] *JavaCC*: A Java Compiler Compiler. Available at `https://javacc.dev.java.net/`.

[2] *Visual Basic .NET Language Reference*. Microsoft Press, 2002.

[3] *Delphi Language Guide*. Borland Software Corporation, 2004.

[4] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., NJ, USA, 1996.

[5] M. S. Ager. From natural semantics to abstract machines. In *Logic Based Program Synthesis and Transformation*, volume 3573 of *LNCS*, pages 245–261. Springer-Verlag, 2005.

[6] E. Allen, J. Bannet, and R. Cartwright. A first-class approach to genericity. In *Proc. OOPSLA'03*, pages 96–114, 2003.

[7] E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, and G. L. S. Jr. Project fortress: A multicore language for multicore processors. *Linux Magazine*, September:38–43, 2007.

[8] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification. Technical report, Sun Microsystems, 2008.

[9] E. Allen, J. J. Hallett, V. Luchangco, S. Ryu, and G. L. Steele, Jr. Modular multiple dispatch with multiple inheritance. In *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1117–1121, New York, NY, USA, 2007. ACM.

[10] D. Ancona, G. Lagorio, and E. Zucca. Jam — a smooth extension of Java with mixins. In *Proc. ECOOP '00*, volume 1850 of *LNCS*, pages 145–178. Springer-Verlag, 2000.

[11] D. Ancona and E. Zucca. An algebra of mixin modules. In *Proc. Workshop on Algebraic Development Techniques'97*, volume 1376 of *LNCS*, pages 92–106. Springer-Verlag, 1997.

[12] D. W. Barron, editor. *Pascal: The Language and its Implementation*. John Wiley, 1981.

[13] D. Beazley and G. V. Rossum. *Python. Essential Reference*. New Riders Publishing, Thousand Oaks, CA, USA, 1999.

[14] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008.

[15] L. Bettini, V. Bono, and S. Likavec. A Core Calculus of Higher-Order Mixins and Classes. In *Proc. TYPES '03 (Selected Papers)*, volume 3085 of *LNCS*. Springer-Verlag, 2004.

[16] V. Bono, F. Damiani, and E. Giachino. On traits and types in a java-like setting. In *Fifth Ifip International Conference On Theoretical Computer Science - Tcs 2008*, IFIP International Federation for Information Processing, pages 367–382. Springer-Verlag, 2008.

[17] V. Bono and J. D. M. Kuśmierek. FJMIP: A calculus for a modular object initialization. In *Proc. FCT 2007*, volume 4639 of *LNCS*, pages 100–112. Springer-Verlag, 2007.

[18] V. Bono and J. D. M. Kuśmierek. Modularizing constructors. *Journal of Object Technology*, vol. 6 no. 9. Special Issue: TOOLS EUROPE 2007:297–317, October 2007.

[19] V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. ECOOP '99*, volume 1628 of *LNCS*, pages 43–66. Springer-Verlag, 1999.

[20] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, The University of Utah, 1992.

[21] G. Bracha, P. Ahe, V. Bykov, Y. Kashai, and E. Mirand. The newspeak programming platform. Technical report, Cadence Design Systems, 2008.

[22] G. Bracha and W. Cook. Mixin-based Inheritance. In *Proc. OOPSLA '90*, pages 303–311. ACM Press, 1990.

[23] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. *SIGPLAN Not.*, 33(10):183–200, 1998.

[24] G. Bracha, M. Odersky, D. Stuotamire, and P. Wadler. *GJ Specification.* May 1998.

[25] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[26] R. J. Cardone. *Language and Compiler Support for Mixin Programming.* PhD thesis, The University of Texas at Austin, 2002.

[27] W. J. Chun. Python 2.2. Q&A with Guido van Rossum, creator of Python. *Linux J.*, 2002(98):4, 2002.

[28] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, (5(2)):56–68, 1940.

[29] T. Cohen and J. Gil. Better construction with factories. *Journal of Object Technology*, vol. 6 no. 6 July/August 2007:109–129, 2007.

[30] S. Cook. OOPSLA'87 panel P2 varietes on inheritance. In *Proc. OOPSLA'87 Addendum to Proceedings*, pages 35–40. ACM Press, 1987.

[31] I. D. Craig. *Programming in Dylan.* Springer-Verlag New York, Inc., NJ, USA, 1996.

[32] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: Fickle$_{II}$. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(2):153–191, 2002.

[33] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 2:331–388, 2006.

[34] S. Ducasse, R. Wuyts, A. Bergel, and O. Nierstrasz. User-changeable visibility: resolving unanticipated name clashes in traits. In *Proc. OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 171–190. ACM, 2007.

[35] U. W. Eisenecker, F. Blinn, and K. Czarnecki. A solution to the constructor-problem of mixin-based programming in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, 10 2000.

[36] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. POPL '98*, pages 171–183. ACM, 1998.

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995.

[38] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Longman, 1983.

[39] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java^{TM} Language Specification.* Addison-Wesley, Sun Microsystems, 2005.

[40] A. Hejlsberg, P. Golde, and S. Wiltamuth. *C♯ Language Specification.* Addison-Wesley, 2003.

[41] H. Ibraheem and D. A. Schmidt. Adapting big-step semantics to small-step style: Coinductive interpretations and "higher-order" derivations. In *Second Workshop on Higher-Order Techniques in Operational Semantics (HOOTS2).* Elsevier, 1997.

[42] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions in Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[43] G. Kahn. Natural semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *LNCS*, pages 22–39. Springer-Verlag, 1987.

[44] S. Kochan. *Programming in Objective-C.* Sams, 2004.

[45] A. Kreczmar, A. Salwicki, and M. Warpechowski. *LOGLAN '88—report on the programming language.* Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[46] J. D. M. Kuśmierek. Implicit first class genericity. In *Proc. Software Composition 2009*, volume 5634 of *LNCS*, pages 107–124, 2009.

[47] J. D. M. Kuśmierek. MTJ: Magda Language Compiler. Available at `http://duch.mimuw.edu.pl/~jdk/`, 2009.

[48] J. D. M. Kuśmierek and V. Bono. Hygienic methods, Introducing HygJava. *Journal of Object Technology*, vol. 6 no. 9. Special Issue: TOOLS EUROPE 2007:209–229, October 2007.

[49] G. Leavens and Y. Cheon. *Design by Contract with JML.* 2003.

[50] X. Leroy. *The Objective Caml System Release 3.09.* Institut National de Recherche en Informatique et en Automatique, 2005.

[51] X. Leroy and H. Grall. Coinductive big-step operational semantics. *CoRR*, abs/0808.0586, 2008.

[52] B. Meyer. An Eiffel Tutorial. Technical Report TR-EI-66/TU, ISE, 2001.

[53] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *Proc. ECOOP '98*, volume 1445 of *LNCS*, pages 355–382. Springer-Verlag, 1998.

[54] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[55] G. Monteferrante. *javamip2java*: A preprocessor for the JavaMIP language. Available at `http://www.di.unito.it/~bono/papers/javamip/`, 2006.

[56] D. A. Moon. Object-oriented programming with flavors. In *Proc. OOPSLA '86*, pages 1–8. ACM Press, 1986.

[57] O. Nierstrasz, S. Ducasse, S. Reichhart, and N. Schärli. Adding Traits to (Statically Typed) Languages. Technical Report IAM-05-006, Institut für Informatik, University of Bern, Switzerland, 2005.

[58] M. Odersky, P. Altherr, V. Creme, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The Scala Language Specification, version 1.0. Technical report, Programming Methods Laboratory, EPFL, 2006.

[59] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, University of Aarhus, 1981.

[60] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behaviour. In *Proc. ECOOP '03*, volume 2743 of *LNCS*, pages 248–274. Springer-Verlag, 2003.

[61] Y. Smaragdakis and D. S. Batory. Mixin-Based Programming in C++. In *Proc. GCSE'00*, volume 2177 of *LNCS*, pages 163–177. Springer-Verlag, 2001.

[62] C. Smith and S. Drossopoulou. Chai: Traits for Java-like Languages. In *Proc. ECOOP'05*, volume 3586 of *LNCS*, pages 453–478. Springer-Verlag, 2005.

[63] B. Stroustrup. *The C++ programming language*. AT&T, 1997. Third edition.

[64] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.