Sterowanie elastycznością i wydajnością aplikacji w procesie wytwarzania oprogramowania

Bartosz Krysztop*
Politechnika Gdańska
Wydział Elektroniki Telekomunikacji i Informatyki
Katedra Architektur Systemów Komputerowych
[krysztop@pg.gda.pl]

Streszczenie

W pracy zaprezentowano nowe podejście umożliwiające wytwarzanie oprogramowania w sposób zapewniający wysoką elastyczność oraz wydajność wykonania. Zaproponowane rozwiązanie opiera się na nowym środowisku programowania TL (*Transformation Language*), w którym na tym samym poziomie abstrakcji definiuje się algorytmy aplikacji oraz procedury optymalizacyjne zapewniające otrzymanie wysoce wydajnego kodu wykonywalnego. Wyniki przeprowadzonych eksperymentów potwierdzają skutecznosc zaproponowanego rozwiązania.

1. Wstęp

W procesie wytwarzania aplikacji, złożone problemy najczęściej rozwiązuje się poprzez ich dekompozycję na mniejsze podproblemy, które można rozwiązać niezależnie [2]. W konsekwencji dostrzeżono zalety metodologii polegającej na wielokrotnym wykorzystaniu raz napisanego kodu niezależnie od istniejących okoliczności (np. typu reprezentacji danych) [9, 8]. Powodem tego są zalety,

*Praca finansowana z grantu KBN 8 T11C 001 17

jakie niese za sobą wielokrotne wykorzystanie kodu: skrócenie czasu wytwarzania oprogramowania, zmniejszone prawdopodobieństwo pojawienia się błędów, zmniejszona ilość nowego kodu podlegającego testowaniu, niższy koszt wprowadzania modyfikacji itp. Modele przetwarzania danych takie jak *programowanie obiektowe* (C++, JAVA [11, 1]) czy *programowanie funkcjonalne* (LISP, Haskell [10, 4]) wspierają wielokrotne wykorzystanie algorytmów. Niestety metody wspierające modularyzację i *software reuse* znacznie obniżają efektywność wykonania aplikacji [5]. Przyczyną tego zjawiska jest m.in. koszt komunikacji pomiędzy odrębnymi jednostkami funkcjonalnymi (takimi jak procedury, obiekty, procesy aplikacji rozproszonej).

Jednym z istotnych parametrów jakościowych aplikacji jest elastyczność (ang. *flexibility*). Przyjmuje się, iż *elastyczność to potencjalna zdolność do szybkiej adaptacji do nowej specyfikacji funkcjonalnej.* Miarą elastyczności jest szeroko pojęty koszt wprowadzenia modyfikacji do istniejącej aplikacji mający na celu rozszerzenie bądź zmianę jej funkcjonalności. Trudno wskazać ogólnie przyjęte metryki umożliwiające ocenę elastyczności aplikacji [6]. Można to tłumaczyć m.in. tym, iż praktycznie nie jest możliwa ocena *potencjalnych* zdolności aplikacji do efektywnej modyfikacji, gdyż rzeczywisty koszt wprowadzenia modyfikacji zależy głownie od kierunku poczynionych zmian. W praktyce wszystkie istniejące metryki koncentrują się na elastyczności w odniesieniu do ustalnego kierunku zmian w aplikacji. Z tego względu można zaproponować następującą ogólną metrykę służącą do pomiaru elastyczności F:

$$F = \frac{F_s}{F_c} \tag{1}$$

gdzie:
F_s — rozmiar modyfikacji
F_c — koszt modyfikacji

Zarówno wielkość modyfikacji jak i koszt modyfikacji można mierzyć na wiele sposobów. Ponieważ rozważania ograniczono do aplikacji wytwarzanch w oparciu o kod wyrażony w formie pliku tekstowego, miarę kosztu wprowadzenia modyfikacji zdefiniowano następująco (zakładajac dodatnią korelację pomiędzy wysiłkiem programistycznym i rozmiarem zmian w pliku tekstowym):

$$F_c = \frac{\text{ilość linii kodu podlegających zmianie}}{\max\{\text{całk. ilość linii przed modyf.}, \text{całk. ilość linii po modyf.}\}} \tag{2}$$

Zaletą w/w metryki jest łatwość pomiaru. Rozmiar modyfikacji F_s można określać procentową zmianą w specyfikacji funkcjonalnej:

$$F_s = 1 - \frac{|f_A \cap f_B|}{|f_A \cup f_B|} \tag{3}$$

gdzie:
f_A — zbiór funkcjonalności zdefiniowany przez specyfikację A
f_B — zbiór funkcjonalności zdefiniowany przez specyfikację B

Jako kryterium oceny wydajności aplikacji przyjęto czas wykonania.

Należy zauważyć, iż wymagania jakościowe w stosunku do aplikacji zmieniają się w zależności od fazy cyklu wytwarzania oprogramowania. W fazie projektowania, implementacji i testowania istotna staje się elastyczność aplikacji, łatwość wprowadzenia poprawek i zmian. Natomiast, w fazie eksploatacji dodatkowo istotnym staje się parametr wydajnościowy. Z tego względu najczęściej stosuje się podejście dwuetapowe:

1. etap — implementacja aplikacji zorientowana na wysoką elastyczność,
2. etap — optymalizacja w/w aplikacji w celu otrzymania wersji o wysokich parametrach wydajnościowych.

Tego typu podejście jest powszechnie spotykanie, przy czym w drugim etapie dokonuje się optymalizacji automatycznie (np. kompilatory optymalizujące). Niestety taki proces optymalizacji może być w większości przypadków niewystarczający. Bezpośrednią przyczyną tego faktu jest ograniczona możliwość ingerencji w charakter optymalizacji (co, kiedy i jak optymalizować), a także lokalny zakres optymalizacji operujący na niezależnych jednostkach funkcjonalnych bez uwzględnienia zachodzących pomiędzy nimi interakcji.

Aby rozwiązać ten problem zaprojektowano oraz zaimplementowano język TL (*Transformation Language*) [7]. Jest to język interpretowalny, bez silnego powiązania typów (ang. *weak typed*), w którym kod wykonywalny jest określony za pomocą struktur danych tego języka. Dzięki temu można na tym samym poziomie abstrakcji definiować algorytmy aplikacji oraz reguły transformacji (optymalizacji) kodu.

W drugim rozdziale opisano zaprojektowane środowisko programistyczne oraz zcharakteryzowano jego podstawowe własności. W trzecim rozdziale przedstawiono algorytm specjalizacji kodu TL mający na celu utworzenie kodu o wysokiej wydajności wykonania. Czwarty rozdział przedstawia eksperyment mający na celu weryfikację użyteczności zaproponowanego środowiska. W ostatnim rozdziale przedstawiono wnioski końcowe.

2. Środowisko TL

Zaproponowane środowisko ma w założeniu przeciwdziałać negatywnemu zjawisku opisanym we wstępie. W tym celu zaprojektowano interpretowalny język TL, w którym można wyrażać procedury transformacji wykonywalnego kodu a przez to m.in. scalać jednostki funkcjonalne. Jakkolwiek mechanizm umożliwiający auto-transformację kodu istnieje w niektórych językach funkcjonalnych (LISP [10], Scheme [3]) to jednak nie jest spotykany w językach obiektowych czy strukturalnych, gdzie występuje operator przypisania i które

Tablica 1: Wyrażenia języka TL

typ wyrażenia	przykład
wyrażenia proste	
dostęp do zmiennej	`a`
odczyt stałej	`4`
wywołanie operatora wbudowanego	`a+4`
wyrażenia złożone	
przypisanie	`a=4`
konstrukcja rekordu	`point=[ x=3 y=z ]`
sekwencja	`a=b; b=c`
dostęp do pola rekordu	`point.x=tab[i]`
wywołanie funkcji	`print(data=2+3)`, `insert(set: S value=v)`

w naturalny sposób przekładają się na język maszynowy. Wyjątkiem od tej reguły jest język TL.

W pełni funkcjonalne środowisko powstało w oparciu o zaledwie 8 wbudowanych typów wyrażeń (patrz Tab. 1). Dzięki temu procedury optymalizacji kodu są nie bardziej złożone, niż kod który podlega optymalizacji (nie pojawia się klasa *meta-języka*).

Kompletna definicja języka wykracza poza zakres tego opracowania. Poniżej opisano najbardziej charakterystyczne cechy tego środowiska wystarczające do prezentacji algorytmu optymalizacji. Język operuje tylko na dwóch typach danych: *symbol* (typ prosty) i *rekord* (typ złożony). Symbole reprezentują sekwencje bajtów (znaków). Wartości tego typu przechowują dowolny rodzaj danych np. łańcuchy tekstowe, binarne reprezentacje liczb itp. Rekordy reprezentują zbiory wartości (dowolnego typu) i posiadają dowolną ilość *pól* (w szczególności 0). Każde pole posiada unikatową *etykietę* (etykiety są symbolami). Kolejność pól jest tu nieistotna; w tym sensie rekordy zachowują się jak asocjacyjne mapy z etykiet na wartości. Na przykład: `[ x=1 y=3 ]` jest równe `[ y=3 x=1 ]`. Te dwa typy umożliwiają definiowanie struktur danych o dowolnej złożoności, na przykład: `moja_lista = [ head=4 tail=[ head=6 tail=nil ] ]`, `moja_tablica = [ 0="jacek" 1="placek" 2="gacek" ]`.

Środowisko wykonania składa się ze zbioru *nazw* (N) które są referencjami do *zmiennych* (zbiór M), które zawierają wartości (elementy zbioru V). Dodatkowo definiuje się podzbiór $V_e \subset V$ zawierający wszystkie wartości, które można interpretować jako wykonywalny kod TL. W danej chwili wiele nazw może wskazywać na te same zmienne, funkcjonują one na podobnej zasadzie jak referencje w języku C++ [11]. Chwilowy globalny stan S aplikacji jest opisany przez parę:

$$S = \langle \mathcal{E}, \mathcal{V} \rangle \tag{4}$$

kontekst wejściowy: `a=[ 0=4 1=8 2=3 ]`, `i=1`, `b=3`

$$\begin{array}{lcl} eval(\texttt{a}) & \to & \&a \\ eval(\texttt{i}) & \to & \&i \\ eval(\texttt{a[i]}) & \to & \&a[1] \\ eval(\texttt{b}) & \to & \&b \\ eval(\texttt{a[i]=b}) & \to & \&a[1] \end{array}$$

kontekst wyjściowy: `a=[ 0=4 1=3 2=3 ]`, `i=1`, `b=3`

Rysunek 1: Interpretacja (ewaluacja) wyrażenia `a[i]=b`

gdzie:
$\mathcal{E}: N \to M$ — funkcja odwzorowująca nazwy na zmienne
$\mathcal{V}: M \to V$ — funkcja odwzorowująca zmienne na wartości

Argumenty do funkcji są przekazywane za pośrednictwem *nazw*. Funkcje nie wymagają deklarowania w sposób jawny jakichkolwiek argumentów, gdyż z założenia operują na określonym zbiorze nazw, które powinny być odpowiednio przygotowane na czas wykonywania funkcji. Podejście to redukuje koszt komunikacji pomiędzy funkcjami. Na przykład `f={ x=t[i] }` przypisuje kod `x=t[i]` (t.j. jego reprezentację w formie rekordu) do zmiennej `f`. Następnie taki kod można wykonać: `f(x: a)` wiąże nazwę `x` ze zmienną wskazywaną przez `a` i wykonuje kod zawarty w `f`. Taka sekwencja operacji jest równoważna z wykonaniem kodu: `a=t[i]`.

Procedura 1 $eval(\texttt{\{}e_l\texttt{=}e_r\texttt{\}}) \to l$

1: $l_l \leftarrow eval(\texttt{\{}e_l\texttt{\}})$,
2: $l_r \leftarrow eval(\texttt{\{}e_r\texttt{\}})$,
3: przypisz wartość $\mathcal{V}(l_r)$ do zmiennej l_l, (zmianie ulega funkcja $\mathcal{V}$)
4: **return** l_l

Formalnie interpretację programu można zamodelować jako wykonanie funkcji ewaluacji kodu, którą można zdefiniować dla każdego typu wyrażenia e (Tab. 1):

$$eval(e) \to l \tag{5}$$

gdzie:
$e \in V_e$ — wyrażenie TL
$l \in M$ — zmienna zwrócona przez wyrażenie (tzw. l-wartość)

Efektem ubocznym tej procedury są zmiany zawartości zmiennych (zmiana funkcji $\mathcal{V}$ w S). Procedura 1 przedstawia semantykę *eval* dla operatora przypisania. Na Rys. 1 przedstawiono interpretację wyrażenia `a[i]=b` zakładając pewnien wejściowy stan S.

3. Specjalizacja kodu

Podstawową cechą zaproponowanego środowiska jest możliwość operowania na wykonywalnym kodzie w trakcie jego działania. Dzięki temu można definiować i kontrolować m.in. techniki optymalizacji, które zakładają transformację kodu aplikacji. Podejście polegające na specjalizacji kodu pojawia się w badaniach naukowaych już od wczesnych lat 70-tych. Jednakże większość zaproponowanych rozwiązań koncentrowała się na dziedzinie języków funkcjonalnych, gdzie takie podejście jest w miarę naturalne. Zaproponowane rozwiązanie umożliwia specjalizację programów strukturalnych, operujących na dowolnych typach danych, również na wskaźnikach. Optymalizacja polega na redukcji wszystkich dających się określić statycznie (a więc nadmiarowych) obliczeń ze specjalizowanego kodu. Podejście takie umożliwia uzyskanie optymalizacji lepszej niż w tradycyjnych kompilatorach. Przyjęto założenie, iż w każdym momencie analizy można jednoznacznie określić, do której zmiennej ma miejsce przepływ danych. Z tego względu ta wersja algorytmu może być tylko stosowana do wąskiej klasy aplikacji, w których przepływ danych nie jest uwarunkowany ich zawartością (np. zaproponowany algorytm specjalizacji nie może być zastosowany do algorytmu sortowania).

Algorytm specjalizacji wykorzystuje podejście „dziel i zwyciężaj”. Dla każdego wyrażenia złożonego w pierwszej kolejności specjalizowane są jego podwyrażenia i uzyskane wyniki służą do konstrukcji wyspecjalizowanego kodu dla e. Dzięki takiemu podejściu algorytm specjalizacji analizyje kod w takiej samej kolejności jak procedura *eval* co zapewnia prawidłową kolejność przypisań i takie same efekty uboczne.

Procedura 2 $Spec(\{e_l{=}e_r\}) \rightarrow \langle l, le, re, ne\rangle$

$\langle l_l, le_l, re_l, ne_l\rangle \leftarrow Spec(\{e_l\})$,
$\langle l_r, le_r, re_r, ne_r\rangle \leftarrow Spec(\{e_r\})$,
przypisz wartość $\mathcal{V}(l_r)$ do zmiennej l_l, (w wyniku zmianie ulega funkcja $\mathcal{V}$)
$le_{out} \leftarrow \{le_l{=}re_r\}$,
if $\mathcal{V}(l_r)$ jest zdefiniowane **then**
    $re_{out} \leftarrow \{ne_l;re_r\}$,
    $ne_{out} \leftarrow \{ne_l;ne_r\}$
else
    $re_{out} \leftarrow \{le_l{=}re_r\}$,
    $ne_{out} \leftarrow \{le_l{=}re_r\}$
end if
return $\langle l_l, le_{out}, re_{out}, ne_{out}\rangle$.

Algorytm specjalizacji jest wykonywany w takim samym środowisku w jakim działa interpreter. Jego implementacja opiera się na funkcji *Spec*:

kontekst wejściowy: `a=[ 0=? 1=8 2=4 ]`, `i=1`, `b=3`

$Spec(\texttt{a})$	$\rightarrow$	$\langle \&a, \texttt{a}, \texttt{a}, \lambda \rangle$
$Spec(\texttt{i})$	$\rightarrow$	$\langle \&i, \texttt{i}, \texttt{1}, \lambda \rangle$
$Spec(\texttt{a[i]})$	$\rightarrow$	$\langle \&a[1], \texttt{a[1]}, \texttt{8}, \lambda \rangle$
$Spec(\texttt{b})$	$\rightarrow$	$\langle \&b, \texttt{b}, \texttt{3}, \lambda \rangle$
$Spec(\texttt{a[i]=b})$	$\rightarrow$	$\langle \&a[1], \texttt{a[1]}, \texttt{3}, \lambda \rangle$

kontekst wyjściowy: `a=[ 0=? 1=3 2=4 ]`, `i=1`, `b=3`

l-wyrażenie:	`a[1]`
r-wyrażenie:	`3`
n-wyrażenie:	λ

Rysunek 2: Specjalizacja wyrażenia `a[i]=b`

$$Spec(e) \rightarrow \langle l, le, re, ne \rangle \tag{6}$$

gdzie:

$e \in V_e$	— kod wyrażenia
$l \in M$	— adres zmiennej zwrócony przez wyrażenie
$le \in V_e$	— opt. wersja e (zakładając, że e jest l-wyrażeniem)
$re \in V_e$	— opt. wersja e (zakładając, że e jest r-wyrażeniem)
$ne \in V_e \cup \{\lambda\}$	— opt. wersja e (zakładając, że e jest n-wyrażeniem)

W zależności od kontekstu użycia wyrażenie można zaliczyć do jednej z trzech grup:

l-wyrażenie	— istotny jest adres zmiennej określonej przez wyrażenie (np. gdy znajduje się po lewej stronie operatora przypisania)
r-wyrażenie	— istotna jest tylko wartość określona przez wyrażanie (np. gdy znajduje się po prawej stronie operatora przypisania)
n-wyrażenie	— bezpośredni wynik interpretacji wyrażenia jest ignorowany, a istotne są tylko efekty uboczne takiej interpretacji.

Ponieważ w ogólności nie jest możliwe ustalenie w jakim kontekscie wyspecjalizowany kod będzie użyty, procedura *Spec* tworzy *trzy* wersje kodu e (le, re, ne). Dodatkowo ne może być pustym wyrażeniem (oznaczonym jako λ).

Procedura *Spec* jest osobno zdefiniowana dla każdego typu wyrażenia. Procedura 2 zawiera definicję *Spec* dla wyrażenia przypisania (por. z Procedurą 1). Procedura usuwa z wyspecjalizowanego kodu instrukcje przypisania, dla których przypisywana wartość jest znana w trakcie specjalizacji. Rys. 2 przedstawia przykładową specjalizaję wyrażenia `a[i]=b`. Kontekst wejściowy definiuje zawartość wszystkich widzialnych zmiennych na wejściu specjalizowanego wyrażenia. Niektóre z nich mają niezdefiniowaną zawartość (oznaczoną jako ?). Procedura *Spec* analizuje kod i przeprowadza jego specjalizację. W wyniku otrzymujemy *kontekst wyjściowy* (uaktualniony stan S), który zawiera całko-

Tablica 2: Porównanie elastyczności oraz wydajności wykonania aplikacji testowych

Aplikacja	ilość linii kodu		F_c	F	czas wyk. [s]	
	A	B			A	B
C_F	103	105	7,62%	13,1	19,52	51,17
C_E	63	79	32,91%	3,0	0,7	3,96
TL	57	57	7,02%	14.2	1,69	8,57

witą wiedzę o zawartości zmiennych po wykonaniu oryginalnego kodu. Jednocześnie otrzymano trzy wersje wyspecjalizowanego kodu (jak widać mogą się one od siebie różnić). Należy zauważyć iż, pomiomo tego, że specjalizacja usunęła instrukcję przypisania, nie utracono żadnej informacji, gdyż jest ona odwzorowana w kontekście wyjściowym.

Złożoność obliczeniowa procedury *Spec* jest liniowa $O(n)$ w funkcji ilości instrukcji wykonywanych podczas interpretacji oryginalnego kodu n. Poprawne zachowanie procedury *Spec* (tzn. generowanie kodu semantycznie tożsamego z oryginalnym) można łatwo udowodnić dla wyrażeń prostych, i uogólnić przez indukcję (wyrażenia złożone) na cały program.

4. Eksperyment

Aby zweryfikować użyteczność zaproponowanego podejścia przeprowadzono eksperyment w którym porównano elastyczność oraz wydajność aplikacji wytworzonych w środowisku TL oraz w języku C. W tym celu dodatkowo zaimplementowano w TL translator TL2C, który dla zadanego kodu w TL drukuje jego odpowiednik w języku C. Aby uzyskać porównywalne dane, wszystkie pomiary wydajności wykonano dla aplikacji skompilowanych w języku C.

Eksperyment składał się z dwóch etapów. W pierwszym etapie zaimplementowano funkcjonalność f_A oraz zmierzono efektywny czas wykonania $t(f_A)$. W drugim etapie dokonano modyfikacji w celu otrzymania funkcjonalności f_B i zmierzono czas wykonania $t(f_B)$. Wykorzystując metrykę (1) zmierzono koszt wprowadzenia modyfikacji (z f_A do f_B). Dla potrzeb eksperymentu wybrano następujące funkcjonalności:

f_A — mnożenie macierzy liczb zmiennoprzecinkowych o rozmiarach 2×2,

f_B — mnożenie macierzy zero-jedynkowych o rozmiarach $(2\times3)\cdot(3\times4)$, dodawanie elementów realizuje operator logiczny OR, mnożenie elementów realizuje operator logiczny AND (np. macierze incydencji wierzchołków grafu)

Jedynym wspólnym elementem obu funkcjonalności jest samo mnożenie macierzy (w sensie ogólnym). Jednakże kontekst ich użycia znacznie się różni, co wyklucza m.in. zamienne stosowanie (nawet w ograniczonym zakresie), co oznacza $f_A \cap f_B = \emptyset$ i w konsekwencji $F_s = 1$ (równanie (3)).

Eksperyment powtórzono w trzech środowiskach programistycznych:

C_E — język C, implementacja zorientowana na wydajność wykonania,

C_F — język C, implementacja zorientowana na wysoką elastyczność,

TL — język TL, implementacja zorientowana na wysoką elastyczność, następnie przetworzona przez funkcję *Spec* oraz translator TL2C,

Aplikacje o wysokiej elastyczności (C_F i TL) otrzymano poprzez wyodrębnienie ogólnej funkcji mnożącej macierze, parametryzowanej przez następujące definicje: element neutralny ze względu na dodawanie (`zero_element`), funkcję dodawnia (`add_element`), funkcję mnożenia (`mul_element`) oraz wymiary macierzy. Pomiary wykonano dla 10 milionów wywołań funkcji mnożącej (procesor Intel 80486DX4 100MHz) dla takich samych danych wejściowych. Tabela 2 zestawia pomiary elastyczności wszystkich trzech implementacji. Jak widać najwyższą elastyczność posiada aplikacja zaimplementowana w środowisku TL. W tym przypadku modyfikacja ograniczyła się do zmiany implementacji funkcji `add_element`, `mul_element` i `zero_element` a także podania innych wymiarów macierzy. Porównywalny wynik otrzymano dla aplikacji C_F skonstruowanej w/g takiego samego schematu. Tabela 2 przedstawia pomierzone czasy wykonania aplikacji. Najwyższą wydajnością wykonania charakteryzowała się aplikacja C_E, zaś najniższą aplikacja C_F. Należy odnotować, iż stosunek $t(C_F)/t(C_E)$ waha się w granicach 13–28 co podkreśla, jak istotny wpływ na ostateczną wydajność aplikacji ma jej konstrukcja architektoniczna. Aplikacja utworzona w oparciu o język TL posiada wydajność wykonania porównywalną z aplikacją C_E (choć nadal jest ponad dwa razy wolniejsza). Jednak biorąc pod uwagę fakt, iż elastyczność tej aplikacji była największa ze wszystkich rozważanych, należy uznać, iż uzyskano tu rozsądny kompromis pomiędzy elastycznością a wydajnością wykonania.

5. Wnioski końcowe

Celem powyższego badania była próba weryfikacji alternatywnego podejścia do wytwarzania oprogramowania, zorientowanego na wysoką wydajność wykonania z zachowaniem wysokiej elastyczności w fazie projektowania, implementacji i testowania. Zaproponowane podejście umożliwia specyfikację algorytmów aplikacji oraz automatycznych procedur optymalizacji na tym samym poziomie abstrakcji. Należy zauważyć, iż specjalizacja kodu może się odbywać przed wykonaniem aplikacji (tak jak to miało miejsce w opisanym eksperymencie), ale również w trakcie interpretacji aplikcji TL. Dalsze badania należy skoncentrować na takiej modyfikacji algorytmów specjalizacji, które umożliwiałyby ich zastosowanie dla szerszej klasy aplikacji.

Bibliografia

[1] Java language specification. http://www.java.sun.com/docs/.

[2] P. Brereton and D. Budgen. Component-based systems: A classification of issues. *Computer*, 33(11):54–62, November 2000.

[3] W. Clinger and I. Rees. Revised 4 report on the algorithmic language scheme. Technical report, Eugene (Oregon), 1992.

[4] Simon Peyton Jones (editor). Report on the programming language haskell 98, a non-strict purely functional language. Tech Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, February 1999.

[5] Stephen H. Edwards. Representation inheritance: A safe form of "white box" code inhertance. *IEEE Transactions on Software Engineering*, 23(2):83–92, February 1997.

[6] H. Krawczyk and B. Wiszniewski. *Analysis and Testing of Distributed Software Applications*. Research Studies Press Ltd. Baldock, Hertfordshire, England, 1998.

[7] Bartosz Krysztop. Environment for processing transformative programs. Technical report, Technical University of Gdańsk, 2000.

[8] Gordon S. Novak. Creation of views for reuse of software with different data representations. *IEEE Transactions on Software Engineering*, 21(12):993–1005, December 1995.

[9] Gordon S. Novak. Software reuse by specialization of generic procedures through views. *IEEE Transactions on Software Engineering*, 23(7):401–417, July 1997.

[10] Guy, Jr. Steele. *Common LISP: The Language*. Digital Press, Bedford, Massachusetts, 1990.

[11] Bjarne Stroustrup. *Język C++*. WNT, 2000. wyd. 5.