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

Istotnym etapem w procesie wytwarzania aplikacji lub systemu informatycznego jest faza jego testowania, wdrażania i rozwoju. Ponieważ większość oprogramowania jest wytwarzana przez zespoły informatyków istotny jest również koszt łączenia kodu napisanego przez różnych programistów, czas wymagany do zrozumienia kodu jednego programisty przez innego oraz specyfikowania

*Praca finansowana z grantu KBN 8 T11C 001 17

interfejsów funkcjonalnych. Na bazie tych problemów wykształciło się wiele metodologii wytwarzania oprogramowania, które skracają drogę od specyfikacji do implementacji, ułatwiają również zrozumienie, modyfikację i rozszerzanie istniejącego kodu. Aby rozwiązać złożony problem najczęściej stosuje się podejście polegające na dekompozycji danego problemu na mniejsze podproblemy, które można rozwiązać niezależnie [1]. Niezależnie od tego dostrzeżono zalety metodologii polegającej na wielokrotnym wykorzystaniu raz napisanego kodu niezależnie od istniejących okoliczności (np. typu reprezentacji danych) [8, 7]. Powodem tego są zalety, 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,
- niższy koszt zmiany platformy programistycznej.

Modele przetwarzania danych takie jak *programowanie obiektowe* (C++, JAVA [10, 2]) czy *programowanie funkcjonalne* (LISP, Haskell [9, 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 komunikacju pomiędzy odrębnymi jednostkami funkcjonalnymi (takimi jak procedury, obiekty, procesy aplikacji rozproszonej).

Jednym z istotnych parametrów jakościowych aplikacji jest elastyczność (ang. *flexbility*). 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. Elastyczność jest utożsamiana z modyfikowalnością (ang. *modifiability*) oraz zdolnością do wielokrotnego wykorzystana kodu (ang. *reusability*) a także przenośnością (ang. *portability*) pomiędzy różnymi platformami. 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 głownie zależy od kierunku poczynionych zmian. W praktyce wszystkie istniejące metryki koncentrują się elastyczności w odniesieniu do ustalnego kierunku zmian w aplikacji. Z tego względu można zaproponować następującą ogólną metrykę służącą do pomiaru elastyczności F:

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

Tablica 1: Porównanie istniejących metodologii wytwarzania oprogramowania

Wymagania użytkownika	Metodologie i narzędzia
niska elastyczność, niska wydajność	nie do przyjęcia z praktycznego punktu widzenia
niska elastyczność, wysoka wydajność	języki niskego pozimu, assemblery, podejścia zorientowane na sprzęt
wysoka elastyczność, niska wydajność	komponenty, języki funkcjonalne, programowanie obiektowe, OMT, narzędzia CASE
wysoka elastyczność, wysoka wydajność	**akceleratory, kompilacja *just-in-time*, techniki optymalizacyjne**

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:

$$F_c = \text{ilość linii kodu podlegających zmianie} \tag{2}$$

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

$$F_s = \left(1 - \frac{|f_A \cap f_B|}{|f_A \cup f_B|}\right) \times 100\% \tag{3}$$

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

Wydajność wykonania aplikacji można oceniać za pomocą takich metryk jako średni czas przetwarzania danych wejściowych, czas reakcji systemu, stopień wykorzystania zasobów systemowych, czy jako złożoność obliczeniową. Oceniając zaproponowane podejście przyjęto definicję wydajności $E = 1/t$, gdzie t to czas wykonania aplikacji. Tabela 1 zestawia różne metodologie wytwarzania programowania ze względu na interesujące nas parametry jakościowe.

Należy zauważyć, iż wymagania jakościowe w stosunku do aplikacji zmieniają się w zależności od fazy cyklu wytwarzania oprogramowania. O ile w fazie projektowania, implementacji i testowania istotnym staje się elastyczność aplikacji, łatwość wprowadzenia poprawek i zmian (również architektonicznych), o tyle w fazie eksploatacji ważniejszym staje się parametr wydajnościowy. Z tego względu nie jest konieczne uzyskanie kodu posiadającego jednocześnie wysoką elastyczność oraz wydajność wykonania, lecz w zamian zastosować 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, a funkcję przejścia od etapu pierwszego do drugiego najczęściej realizują kompilatory optymalizujące. Niestety ten proces optymalizacji może być w większości przypadków niewystarczający. Bezpośrednią przyczyną tego faktu jest ograniczona możliwość intgerencji w charakter optymalizacji (*co* optymalizować, *kiedy* optymalizować i *jak* optymalizować), a także lokalny zakres optymalizacji operujący na niezależnych jednostkkach funkcjonalnych bez uwzględnienia zachodzących pomiędzy nimi interakcji.

Aby rozwiązać ten problem zaprojektowano oraz zaimplementowano język TL (*Transofrmation language*). 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 elastycznego na kod wydajny, przez co można w potencjalnie nieograniczonym stopniu sterować elastycznościa i wydajnością aplikacji. Zaproponowane podejście należy zaliczyć do ostatniej kategorii wymienionej w Tabeli 1.

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 opisanymu we wstępie. W tym celu zaprojektowano interpetowalny język TL, w którym można wyrażać procedury transformacji wykonywalnego kodu a przez to m.in. scalać jednostki funkcjonalne. Jakkolwiek mechanizm

Tablica 2: Wyrażenia języka TL

typ wyrażenia	przykład
	wyrażenia proste
dostęp do zmiennej	`a`
odczyt stałej	`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)`

umożliwiający auto-transformację kodu istnieje w niektórych językach funkcjonalnych (LISP [9], Scheme [3]) to jednak nie jest spotykany w językach obiektowych czy strukturalnych, gdzie występuje operator przypisania i które w naturalny sposób przekładają się na język maszynowy. Wyjątkiem od tej reguły jest język TL.

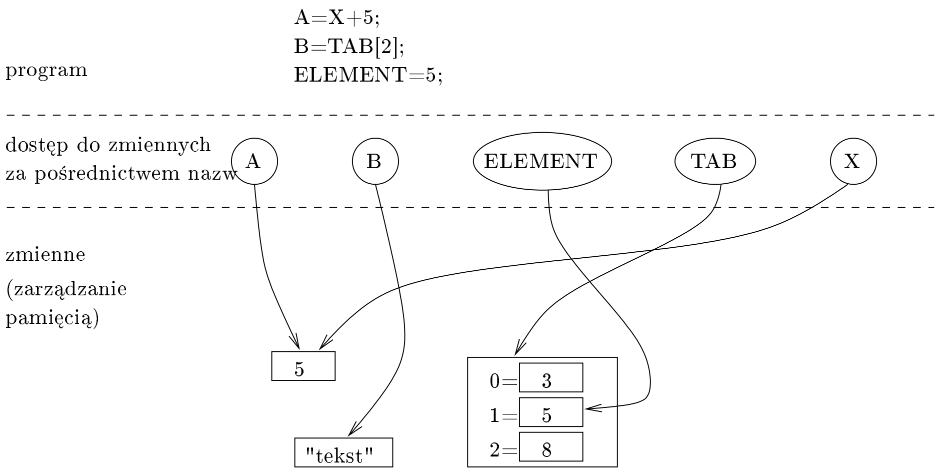
Jednym z warunków stawianym przed projektowanym narzędziem była możliwie minimalna ilość wbudowanych mechanizmów. W rezultacie w pełni funkcjonalne środowisko powstało w oparciu o zaledwie 8 wbudowanych typów wyrażeń (patrz Tab. 2).

Zaletą takiego podejścia jest dużo łatwiejsza analiza kodu języka. Na przykład procedury optymalizacji kodu są nie bardziej złożone, niż kod który podlega optymalizacji (nie pojawia się klasa *meta-języka*). Miało to również na celu umożliwienie potencjalnym algorytmom analizujących kod samemu „dostrzec" własności danych konstrukcji (np. jak zachowuje się pętla `while`).

Kompletna definicja języka wykracza poza zakres tego opracowania. Poniżej opisano najbardziej charakterystyczne cechy tego środowiska wystarczające do prezentacji algorytmu optymalizacji. 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++ [10]. Zależności pomiędzy nazwami, zmiennymi i wartościami zilustrowano na Rys. 1. Chwilowy globalny stan S aplikacji jest opisany przez parę:

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

gdzie:



Rysunek 1: Mechanizm dostępu do zmiennych

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

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). Wartości w rekordzie są dostępne za pośrednictwem etykiet. 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 umożliwiją 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" ]
```

W języku TL argumenty do funkcji są przekazywane za pośrednictwem *nazw*. Aktualny stan dowiązań nazw do zmiennych nazywamy *środowiskiem wykonania* (funkcja $\mathcal{E}$),

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 $eval(\{e_l=e_r\}) \rightarrow l$

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

O efektywności zaproponowanego modelu funckjonalnego świadczy fakt, iż podstawowe instrukcje przepływu sterowania zostały zaimplementwane na bazie wbudowanych operatorów. Istrukcję sterowanie typu *if... else* zaimplementowano jako funkcję `ifelse`, która wykonuje kod zawarty w zmiennej `cond` i w zależności od wyniku (0 lub 1) wykonuje kod zawarty w zmiennej `then` lub `else`. Warunkowe wykonanie instrukcji (sterowanie typu *if*) to w/w funkcja z pustą częścią *else*:

```
if={ ifelse( else = {} ) }
```

Z kolei `while` dla zadanego warunku `cond` oraz kodu `do` to[1]:

```
while={ if( then = { do(); while() } ) }
```

Formalnie interpretację programu można zamodelować jako wykonani funkcji ewaluacji kodu, którą można zdefiniować dla każdego typu wyrażenia e (Tab. 2):

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

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

Efektem ubocznym tej procedury są zmiany zawartości zmiannych (zmiana funkcji $\mathcal{V}$ w S). Procedura 1 przedstawia semantykę *eval* dla operatora przypisania. Na Rys. 2 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. Dzieki 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

[1] Warto zwrócić uwagę na rekursywne wywołanie funkcji `while`. Kod ilustruje związek pomiędzy rekurencją i iteracją.

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

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

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

Rysunek 2: Interpretacja (ewaluacja) wyrażenia `a[i]=b`

takie podejście jest w miarę naturalne. Zaproponowane rozwiązanie umożliwia specjalizację programów strukturalnych, operujących na dowolnych typach danych, również na wskaźnikach. Podstawowym założeniem optymalizacji jest redukacja wszystkich dających się określić statycznie (a więc nadmiarowych) obliczeń ze specjalizowanego kodu. Podejście takie umożliwia uzyskanie optymalizacji lepszej niż w tradycyjnych kompilatorach. Zaproponowany algorytm optymalizacji zakłada, iż w każdym momencie analizy można jednoznacznie określić, do której zmiennej ma miejsce przepływ danych. Z tego względu ta wesja 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. Dzieki takiemu podejściu algorytm specjalizacji analizyje kod w takiej samej kolejności jak procedura *eval* co zapewnia prawidłową kolejność przypisań i takie same efekty uboczne.

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

$$
Spec(e) \rightarrow \langle l, lc, rc, nc \rangle \tag{6}
$$

gdzie:

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

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

l-wyrażenie — kiedy znajduje się po lewej stronie operatora przypisania (t.j. istotna jest adres zmiennej określona przez to wyrażenie)

r-wyrażenie — kiedy istotna jest tylko wartość określona przez to wyrażanie

n-wyrażenie — kiedy 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). Podstawową własnością *Spec* jest usuwanie z wyspecjalizowanego kodu instrukcji przypisania, dla których przypisywana wartość jest znana w trakcie specjalizacji.

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

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

Rys. 3 przedstawia przykładową specjalizaję wyrażenia `a[i]=b`. Kontekst wejściowy definiuje zawartość wszystkich widzialnych zmiennych na wejściu specjalizowanego wyrażenia. Niektóre zmają niezdefiniowaną zawartość (oznaczoną jako ?). Procedura *Spec* analizuje kod przeprowadza jego specjalizację. W wyniku otrzymujemy *kontekst wyjściowy* (uaktualniony stan S), który zawiera całkowitą 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 kontekscie wyjściowym.

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

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

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

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

Rysunek 3: Specjalizacja wyrażenia `a[i]=b`

Tablica 3: Modyfikowalność aplikacji testowych

Aplikacja	ilość linii kodu		F_c	F
	A	B		
C_F	103	105	8	12,5%
C_E	63	79	26	3,5%
TL	57	57	5	20%

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. Konstrukcja translatora okazała się stosunkowo skomplikowana ze względu na konieczność implementacji systemu wnioskowania typów, które w odróżnieniu od języka C nie są jawnie deklarowane w TL. 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 takiej procedury $t(f_B)$. Następnie dokonano modyfikacji w implementacji w celu otrzymania funkcjonalności f_B i również zmierzono czas wykonania otrzymanej implementacji $t(f_B)$. 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 $(2 \times 3) \cdot (3 \times 4)$, dodawanie elementów realizuje operator logiczny OR, mnożenie elementów realizuje operator logiczny AND (np. macierze incydencji wierzchołków grafu)

Jak widać jedynym wspólnym mianownikiem obu implementacji jest samo mnożenie macierzy w sensie ogólnym. Jednakże kontekst ich użycia znacznie się różni.

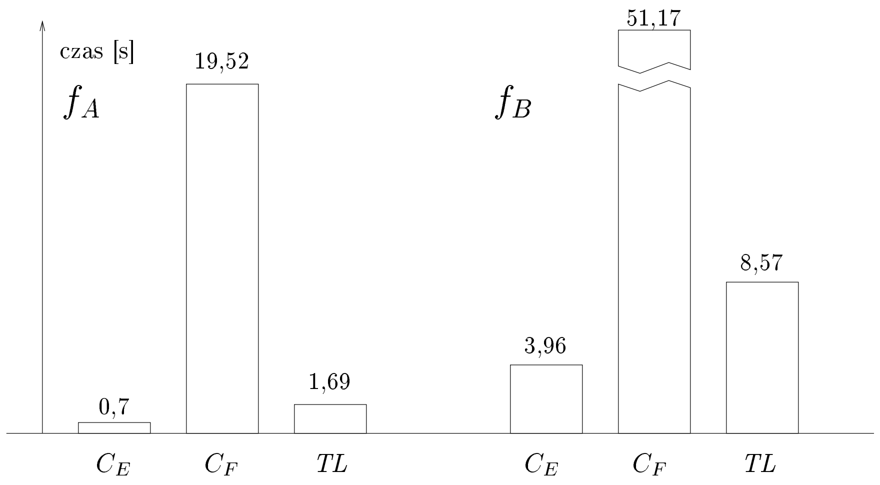
Eksperyment powtórzono w trzech środowiskach programistycznych:

C_E — język programowania C, implementacja zorientowana na wydajność wykonania,

C_F — język programowania 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. Wszystkie eksperymenty przeprowadzono dla takich samych danych wejściowych, pomiary wykonano dla 10 milionów wywołań funkcji mnożącej (procesor Intel 80486DX4 100MHz). Tabela 3 zestawia pomiary elastyczności wszystkich trzech implementacji (dla $F_s = 100\%$). Jak widać najwyższą elastyczność utrzymuje 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. Tutaj w przypadku modyfikacji kodu dodatkowy koszt pojawił się w związku z koniecznością zmiany deklaracji typu elementów macierzy (z `double` na `bool`). Rys. 4 przedstawia pomierzone czasy wykonania aplikacji. Jak widać najwyższą wydajnością wykonania charakteryzowała się aplikacja C_E, zaś najniższą aplikacja C_F. Warto odnotować, iż



Rysunek 4: Czas wykonania oraz rozmiar kodu aplikacji testowych

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 (niespotykany np. w języku C).

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. Dzięki temu nie istnieje potrzeba tworzenia dodatkowych narzędzi przetwarzających kod aplikacji. Opisane rozwiązanie można traktować jako alternatywę dla rozbudowanych narzędzi wspierających optymalizację kodu. 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] Component-based systems: A classification of issues. *Computer*, pages 54–62.

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

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

[4] Paul [editor] Hudak, Simon [editor] Peyton Jones, Philip [editor] Wadler, Brian Boutel, Jon Fairbairn, Joeseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell — A Non-strict, Purely Functional Language. Research Report 1.2, Yale University, New Haven, Conneticut, U.S.A. and University of Glasgow, Glasgow, U.K., 1992.

[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] 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.

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

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

[10] Bjarne Stroustrup. *Język C++.* WNT, 1994.