Przykładowe struktury danych w komputerowych systemach wspomagających konstrukcję poprawnych programów

Janusz Jabłonowski*
Instytut Informatyki, Uniwersytet Warszawski

Streszczenie

Niniejsza praca dotyczy komputerowych systemów wspomagających tworzenie poprawnych programów. Omawiamy w niej zagadnienia związane z wprowadzeniem do takich systemów złożonych typów danych (np. rekordów czy tablic). Problemy te są dwojakiej natury. Po pierwsze, wprowadzenie nowych typów danych zwykle wymaga rozbudowania reguł systemu wnioskowania używanego do weryfikacji poprawności programów. Po drugie, dla wygodnego posługiwania się takimi typami zwykle jest konieczne także wzbogacenie języka asercji używanych do opisu stanów programu.

1. Wstęp

Obserwujemy obecnie niezwykle dynamiczny rozwój zastosowań informatyki. Nie tylko nieustannie rośnie liczba komputerów, ale także stale zwiększa się zakres ich stosowania (trudno jest dziś podać jakąś dziedzinę, w której nie używano by jeszcze komputerów). O ile ze wzrostem liczby komputerów dosyć łatwo sobie poradzić zwiększając po prostu ich produkcję, o tyle zaspokojenie popytu na coraz to nowe zastosowania komputerów napotyka na bardzo istotne

*Praca częściowo finansowana z grantu KBN nr 8 T11C 015 15

przeszkody. Są nimi: niezadowalająca jakość oprogramowania i duże koszty (także czasowe) związane z jego wytwarzaniem.

Ponieważ narzędzia do tworzenia oprogramowania - języki programowania - mają bardzo precyzyjnie, formalnie zdefiniowaną składnię i (zwykle) semantykę, narzuca się rozwiązanie powyższych problemów polegające na zastosowaniu przy tworzeniu oprogramowania metod formalnych. Wydaje się to tym bardziej oczywiste, że tworzący oprogramowanie programiści mają zwykle dość dobre przygotowanie matematyczne. Niestety, mimo tego panuje powszechne przekonanie, że zastosowanie metod formalnych jest możliwe jedynie teoretycznie. Wymienia się trzy przyczyny niemożności zastosowania metod formalnych na szeroką skalę:

- duży nakład pracy związany z poznaniem i stosowaniem tych metod,
- możliwość popełniania błędów podczas samej weryfikacji poprawności programów,
- konieczność tworzenia formalnych specyfikacji programów (tylko względem nich można formalnie weryfikować poprawność programu).

Aczkolwiek wszystkie wymienione zarzuty są prawdziwe, to jednak odnoszą się one przede wszystkim do sytuacji, w której metody formalne są stosowane bez żadnego wspomagania. W Instytucie Informatyki Uniwersytetu Warszawskiego od szeregu lat prowadzone są prace mające na celu stworzenie narzędzi pozwalających przezwyciężyć przedstawione problemy ([4], [6]). Żmudny, a przez to podatny na błędy, proces weryfikacji poprawności oprogramowania można efektywnie wspomóc komputerowo. Co więcej można to robić na wiele sposobów ([2], [8]). Można weryfikować poprawność gotowych programów. Jednak znacznie bardziej obiecującym podejściem jest budowanie systemów, które nie weryfikują poprawności gotowych programów, lecz na każdym etapie tworzenia oprogramowania wymuszają ich poprawność (metodologiczne podstawy takich systemów można znaleźć w pracach [3] i [5]). Wyniki przedstawione w niniejszej pracy mają zastosowanie w obu tych podejściach.

Zwróćmy uwagę, że współczesne produkcyjne środowiska programistyczne również zdają się podążać w tym drugim kierunku. Większość z nich ma wbudowane narzędzia do składniowej analizy programu na bieżąco (podpowiadanie składni, automatyczne poprawianie pisowni uprzednio zadeklarowanych identyfikatorów, itp.) Również sam rozwój inżynierii oprogramowania (budowanie programów z komponentów, metody obiektowe, programowanie rozproszone), powoduje że coraz większego znaczenia nabiera pisanie względnie niedużych fragmentów programów z dobrze zdefiniowaną specyfikacją. A w takich warunkach prezentowane w tej pracy podejście sprawdza się najlepiej.

Jednym z istotnych problemów związanych z poprawnością (bezpieczeństwem) pisanych programów jest zapewnienie poprawności operacji na strukturach danych używanych przez program. I tym właśnie zagadnieniem zajmiemy się w niniejszej pracy. Jest ona zorganizowana następująco. Najpierw omawiamy ewolucję stosowanych dotychczas sposobów zwiększania bezpieczeństwa tworzonych programów, zwracając uwagę na podobieństwa z proponowanymi w niniejszej pracy rozwiązaniami. Następnie przechodzimy do opisu typów danych. Zaczynamy do rekordów, podając dwa możliwe sposoby ich formalnego traktowania. Potem zajmujemy się problemem włączenia tablic do systemu wspomagającego tworzenie oprogramowania. Wreszcie w ostatnim rozdziale przedstawiamy kierunki dalszych badań.

2. Sposoby zwiększania bezpieczeństwa

Ponieważ błędy programistyczne związane np. z niepoprawnym używaniem tablic są często spotykane, wymyślono kilka sposobów zapobiegania im, bądź choćby wczesnego ich wykrywania i lokalizowania. Zapewne pierwszym jest *metoda wydruków kontrolnych*, polegająca na umieszczeniu w newralgicznych punktach programu instrukcji wypisujących wartości istotnych dla bezpiecznego wykonania programu wyrażeń. To oczywiście nie pozwala unikać błędów, a jedynie ułatwia ich lokalizowanie. Wady tej metody to trudność przewidzenia, które wyrażenia okażą się istotne dla zlokalizowania błędów w programie, zmniejszenie przejrzystości programu przez dodanie licznych instrukcji wypisujących oraz trudność przechodzenia z wersji testowej do produkcyjnej i z powrotem (konieczność usuwania i przywracania instrukcji generujących wydruki kontrolne). Istotne niedogodności tej metody starano się oczywiście usunąć i, co ciekawe, pojawiły się tu dwa różne podejścia.

Pierwsze polega na stwierdzeniu, że skoro trudno z góry ustalić, wartości jakich wyrażeń będą potrzebne programiście do zlokalizowania błędu, to w ogóle nie należy się nad tym zastanawiać dopóki nie pojawi się błąd. Dopiero wtedy gdy okaże się, że program źle działa należy go interaktywnie wykonywać przy użyciu systemu wspomagającego śledzenie wykonania (ang. debugger) i badać na bieżąco wartości zmiennych. To podejście oczywiście rozwiązuje problemy zarządzania instrukcjami generującymi wydruki kontrolne (w ogóle ich nie ma), ale na tym jego zalety się kończą. Ma natomiast poważne wady, gdyż zachęca do bardzo niefrasobliwego stylu programowania (napisz coś a potem zobacz czy przypadkiem nie działa).

Drugie podejście polega na zastosowaniu warunkowo kompilowanych operacji (zwykle o nazwie *assert*), które generują wydruki tylko wtedy, gdy określone przez programistę warunki nie są spełnione. Jest to oczywiście zdecydowanie lepsze rozwiązanie od metody wydruków kontrolnych: nie ma problemu przechodzenia między wersją testową a produkcyjną (wystarcza jedynie włączyć lub wyłączyć kompilowanie operacji assert), wersja produkcyjna nie jest obarczona żadnymi narzutami (tak pod względem szybkości działania jak i rozmiaru generowanego kodu programu). Ponadto program zyskuje na czytelności, bo operacje assert można traktować jako komentarze programisty (przy każdym wykonaniu program dochodząc do miejsca zapisania operacji assert wylicza takie wartościowanie zmiennych, w którym spełniony jest podany warunek). Zwróćmy uwagę, że stosowanie tego podejścia pozwala nie tyle śledzić wykonywanie programu ile wykrywać (zwykle z wyprzedzeniem) błędne sytuacje.

Zapisane w ten sposób warunki bardzo przypominają asercje omawiane w [7]. Są jednak istotne różnice między nimi. Po pierwsze, asercje są kompletnym opisem pożądanych stanów obliczeń, natomiast warunki podawane w operacjach assert mogą dotyczyć tylko niektórych elementów stanu programu (np. tylko jednej zmiennej). Po drugie, asercje są analizowane niezależnie od konkretnego wykonania programu (zwykle przed pierwszym wykonaniem programu), a warunki operacji assert są wyliczane podczas wykonywania programu.

Pomysł z umieszczaniem w programie operacji assert można starać się ulepszyć. Ponieważ asercje często dotyczą typowych błędów programistycznych (takich jak np. wyjście poza tablicę), można asercje tego rodzaju generować automatycznie (większość kompilatorów daje taką możliwość). Drugie ulepszenie, zastosowane w systemie ESC [2], polega na wykorzystaniu do analizy asercji mechanizmów automatycznego dowodzenia twierdzeń, tak by informacje o błędach w programie generować jeszcze zanim program rozpocznie swoje działanie.

Naturalnym dalszym krokiem w rozwoju metod zwiększających bezpieczeństwo pisanych programów, jest szersze zastosowanie asercji i narzędzi weryfikujących poprawność twierdzeń. System ESC koncentruje się jedynie na wykrywaniu typowych błędów programistycznych, nie badając poprawności programu względem zadanej specyfikacji. W pracy [8] opisano system pozwalający weryfikować poprawność prostych programów napisanych w podzbiorze Pascala względem zadanej specyfikacji. Ten system działa na gotowym tekście programu uzupełnionym o asercje. Kolejnym etapem rozwoju tego typu systemów są środowiska wspomagające nie tyle weryfikację gotowych programów, ile samo konstruowanie poprawnych programów.

Zarówno weryfikowanie gotowych programów jak i wspomaganie konstruowania poprawnych programów wymagają odpowiednich systemów logicznych do wnioskowania o poprawności programów oraz mechanizmów do dowodzenia generowanych twierdzeń. O ile zdefiniowanie takiego systemu dla prostych instrukcji i typów danych nie przedstawia większych trudności, o tyle bardziej złożone konstrukcje (rekurencja, różne tryby przekazywania parametrów) i typy danych (rekordy, tablice, wskaźniki) wymagają już istotnego rozbudowania takiego systemu.

3. Rekordy

Rekordy są stosunkowo prostym pojęciem, nie zwiększającym siły wyrazu języka, służącym jedynie do grupowania logicznie ze sobą powiązanych zmiennych w jedną całość. Skoro tak, to powstaje pokusa, by opis rekordów był wyrażony w kategoriach elementów już wykorzystywanych w opisie semantyki pozostałej części języka - jest to pierwsze proponowane podejście. Drugie z proponowanych podejść wymaga rozbudowania mechanizmu przeprowadzającego dowody. W niniejszej pracy omawiamy jedynie rekordy bez wariantów.

3.1. Pierwsze rozwiązanie

W tym podejściu traktujemy rekord jako paczkę zmiennych. Odwołania do pól rekordu traktujemy jako odwołania do zmiennych o odpowiednio kodowanych nazwach, np. poniższy fragment programu (zapisany w Pascalu):

```
var
    r : record
            x: integer;
            y: real;
        end;
{ ... }
r.x := 13;
{ ... }
```

zamieniamy na:

```
var
    r!x: integer;
    r!y: real;
{ ... }
r!x := 13;
{ ... }
```

Znak '!' służy jedynie temu, by nowe identyfikatory, wprowadzone na potrzeby tłumaczenia, nie pokrywały się z identyfikatorami występującymi w programie oryginalnym. Zamiast znaku '!' można użyć dowolnego innego znaku nie występującego w poprawnych identyfikatorach.

Zalety tego podejścia:

- nie wymaga wzbogacania języka opisu semantyki o nowe konstrukcje, po prostu tłumaczymy program z rekordami na program bez nich,

- jest zgodne z intuicją, że rekord nie jest niczym więcej niż paczką zmiennych.

Wady:

- w przypadku bardziej złożonych typów danych to tłumaczenia staje się niewygodne, np. z tablicy rekordów robi się kilka tablic,

- nie daje się zastosować do zmiennych dynamicznych (które zwykle są rekordami). Można by wprawdzie (analogicznie jak w przypadku tablic) zastąpić polecenie utworzenia dynamicznej zmiennej rekordowej kilkoma instrukcjami **new**, ale kłopot polega na tym, że potrzebujemy wtedy tyle wskaźników, ile było pól w rekordzie. Dla pojedynczych zmiennych dynamicznych to nie jest problem, ale w przypadku struktur rekurencyjnych (np. listy) uniemożliwia to ich realizację (dla zapamiętania jednego elementu listy należałoby przeznaczyć więcej wskaźników niż jest w tym elemencie wszystkich pól).

3.2. Drugie rozwiązanie

Drugi pomysł polega na wprowadzeniu do mechanizmu przeprowadzającego dowody twierdzeń obiektów odpowiadających rekordom. Ponieważ w większości języków programowania wszystkie typy danych używane w programie dają się wyznaczyć podczas statycznej analizy jego tekstu (nie ma typów tworzonych dynamicznie podczas wykonywania), można do każdego programu dobrać odpowiedni zestaw takich obiektów. Na tych obiektach należy zdefiniować operacje pobierania wartości pola oraz tworzenia nowych obiektów na podstawie wartości poszczególnych składowych. Niektóre systemy dowodzenia twierdzeń mają wbudowane wsparcie dla takich obiektów (np. PVS [9]).

Zalety tego podejścia:

- daje się łatwo opisać,

- daje się zastosować we wszystkich kontekstach w jakich używa się rekordów,

Wady:

- do zdefiniowania znaczenia rekordów wymaga silniejszego języka opisu.

Wprawdzie w niniejszej pracy nie rozważamy wskaźników, ale wady pierwszego podejścia uniemożliwiają (planowane) rozszerzenie tej pracy o następne konstrukcje językowe, zatem wybieramy drugie rozwiązanie.

W samym systemie wnioskowania wprowadzenie rekordów wymaga jedynie kosmetycznych zmian (w instrukcjach przypisania i wywołania procedury). Dużo trudniejszy problem stanowią tablice.

4. Tablice w ujęciu formalnym

Problemy z wprowadzeniem tablic do formalnego systemu wnioskowania o programach są dwojakiego rodzaju. Po pierwsze, reguły wnioskowania wystarczające dla prostych zmiennych nie wystarczają w przypadku użycia tablic. Po drugie, samo zmodyfikowanie reguł wnioskowania pozwala wprawdzie udowadniać poprawność programów korzystających z tablic, ale nie rozwiązuje drugiego istotnego problemu związanego z używaniem tablic, polegającego na niemożności opisywania interesujących programistę własności programów. Rozwiązanie tego drugiego problemu wymaga rozbudowania języka asercji o konstrukcje pozwalające na wyrażanie właściwości całych tablic, a nie tylko ich pojedynczych elementów.

4.1. Zmodyfikowane reguły wnioskowania

Skoro tablica (jednowymiarowa) jest uporządkowanym ciągiem zmiennych typu prostego, a najczęściej stosowaną operacją na tablicy jest przypisywanie lub odczytywanie wartości jej pojedynczych elementów, to na pierwszy rzut oka wydawałoby się, że do wnioskowania o tablicach wystarczą reguły opracowane dla języka tylko z typami prostymi. Tak jednak nie jest. Problem wynika z tego, że (czasami) analizując program z odwołaniami do tablic nie jesteśmy w stanie stwierdzić, które elementy tablicy będą przez ten program używane (ta elastyczność typu tablicowego stanowi zresztą o jego sile). Oto prosty przykład tej sytuacji (zapisany w Pascalu):

read(i);
$tab[i] := 0$;

A oto klasyczny [1] przykład problemów na jakie napotykamy próbując podać reguły wnioskowania dla programów z tablicami. Następująca reguła wnioskowania wydaje się być bezpieczna:

Po wykonaniu instrukcji
$tab[e] := 0$
zachodzi
$tab[e] = 0$
(gdzie e jest wyrażeniem dającym w wyniku poprawny indeks w tablicy tab).

Niestety poniższy przykład pokazuje, że ta reguła jest błędna. Załóżmy, że wartością elementu tablicy *tab* o indeksie 1 jest 1 i wykonajmy poniższe przypisanie:

$tab[tab[1]] := 0$;

Jego efektem jest przypisanie wartości 0 elementowi tablicy o indeksie 1. Zatem po wykonaniu tego przypisania nie musi być prawdą, że:

$$tab[tab[1]] = 0$$

(zależy to od tego jaką wartość przed wykonaniem omawianego przypisania miał element tablicy *tab* o indeksie 0 i czy w ogóle istniał).

Chyba najelegantszym rozwiązaniem powyższych problemów jest traktowanie tablic w systemie dowodzenia jako pojedynczych obiektów. Wówczas przypisanie na element tablicy możemy traktować jako operację dającą w wyniku nową tablicę, identyczną z tablicą sprzed przypisania we wszystkich, prócz jednego, elementach.

Kłopoty z wnioskowaniem o tablicach nie kończą się na instrukcji przypisania, co najmniej tak samo niebezpieczna jest reguła dla wywołania procedury. W tym przypadku problemy są związane z przekazywaniem parametrów przez zmienną. Weźmy jako przykład poniższą procedurę (zapisaną w Pascalu):

```
procedure Zwiększ(var x,y: integer);
    begin
        x := y + 1;
    end
```

Oczekujemy, że po wykonaniu tej procedury wartość argumentu odpowiadającego parametrowi x będzie o jeden większa od wartości drugiego argumentu (pomijając możliwość przekroczenia zakresu zmiennych). Niestety nie musi to być prawdą, jak to pokazuje poniższe wywołanie tej procedury:

Zwiększ($tab[i]$, $tab[j]$);

Jeśli i będzie równe j to oczywiście pierwszy argument procedury nie będzie mógł być większy od drugiego. Z czego wynika ten problem? Przede wszystkim z tego, że znaczenie procedury, której niektóre parametry przekazywane przez zmienną odpowiadają temu samemu argumentowi, zwykle będzie zupełnie inne niż znaczenie tej samej procedury przy zachowaniu rozłączności argumentów. Stwierdzenie czy argumenty procedury są rozłączne jest znacznie bardziej skomplikowane w przypadku języków, w których występują tablice i zwykle wymaga udowadniania dodatkowych warunków (jak np. $i \neq j$ w powyższym przykładzie).

Reguła wnioskowania (nieco uproszczona) dla instrukcji przypisania na element tablicy ma następującą postać:

$$\frac{\begin{array}{c} Ax \vdash \alpha \Rightarrow \mathrm{valid}(e_1) \\ Ax \vdash \alpha \Rightarrow \mathrm{valid}(e_2) \\ Ax \vdash \alpha \Rightarrow \beta[t/t(e_1 : e_2)] \\ Ax \vdash \alpha \Rightarrow e_1 \in domain(t) \end{array}}{\overset{Ax}{\vdash}_H \{\alpha\}\ t[e_1]\ := e_2\ \{\beta\}}$$

Ax to zestaw aksjomatów, opisujących rozważaną dziedzinę algorytmiczną. Wynikiem operacji *valid* jest warunek gwarantujący poprawność wyliczenia zadanego jako parametr wyrażenia (np. może to być warunek zapewniający, że dzielnik jest różny od zera). Operacja $domain(t)$ daje jako wynik zakres dozwolonych indeksów tablicy t, zatem zwykle wyrażenie $e_1 \in domain(t)$ będzie się tłumaczyć na koniunkcję dwu nierówności: $(e_1 \geq d) \wedge (e_1 \leq g)$, gdzie d i g oznaczają, odpowiednio, dolny i górny zakres elementów tablicy t. Zapis $t(e_1 : e_2)$ oznacza tablicę t zmienioną na pozycji e_1 tak, by miała tam wartość e_2. Konkluzja reguły stwierdza, iż w proponowanym systemie wnioskowania, można udowodnić, że rozpoczęcie wykonanie instrukcji przypisania w stanie spełniającym warunek α gwarantuje zakończenie jej wykonywania w stanie spełniającym warunek β.

Ze względu na brak miejsca nie zamieszczamy tu reguły dla instrukcji wywołania procedury. Można ją znaleźć, wraz z pozostałymi regułami i ich pełnym opisem, w przygotowywanej obecnie rozprawie doktorskiej autora. Jest tam też pokazane jak przy pomocy tych reguł można wyliczyć poprawny wynik podanego tu wcześniej przykładu dla przypisania.

4.1.1. Wzbogacony język asercji

Do zapisywania asercji w programach ze zmiennymi prostymi doskonale nadaje się język logiki pierwszego rzędu wzbogacony o równość i liczby całkowite. W przypadku tablic to już nie wystarcza. Jeden z najprostszych przykładów związanych z tablicami to sumowanie ich zawartości. Do opisania wyniku obliczeń należałoby użyć specjalnej operacji *suma(zakres, co)*, podobnej do kwantyfikatorów, której wynikiem jest suma elementów wskazanych parametrami (np. *suma(i:1..n, tab[i])*). Niestety drobna modyfikacja podanego zadania (np. sumowanie tylko nieparzystych elementów tablicy) powoduje, że znów nie da się opisać wyniku. Można temu zaradzić wprowadzając uogólnioną operację sumowania *suma(zakres, warunek, co)* sumującą tylko te ze wskazanych elementów, dla których zachodzi warunek (np. sumę nieparzystych elementów tablicy o indeksach 1..n można zapisać tak: *suma(i:1..n, odd(i), tab[i])*, gdzie predykat *odd* oznacza nieparzystość argumentu). Gdyby zamiast sumy elementów potrzebny był ich iloczyn, bądź liczba elementów tablicy spełniających jakiś zadany warunek, podana operacja okazałaby się za słaba. Dlatego postulujemy wprowadzenie ogólniejszej operacji iteracji, mającej następującą postać

iter(zakres, warunek, wart_pocz, operacja, co)

gdzie *zakres*, *warunek* i *co* mają znaczenie takie jak poprzednio, zaś *wart_pocz* jest początkową wartością, od której zaczyna się wykonywanie zadanej przedostatnim parametrem operacji. Na przykład liczbę wszystkich dodatnich elementów tablicy *tab* można wyrazić następująco:

iter(i:1..n, tab[i]>0, 0, +, 1).

W kontekście komputerowego wspomagania konstruowania programów rozszerzanie języka asercji musi być robione bardzo ostrożnie, tak by nie stracić możliwości automatycznego weryfikowania poprawności programu. Oczywiście systemy automatyczne nigdy nie będą w stanie w pełni weryfikować poprawności dowolnego programu z asercjami zapisanymi w logice pierwszego rzędu (wynika to z nierozstrzygalności tej logiki), ale im mniej rozszerzeń wprowadzimy do logiki pierwszego rzędu, tym więcej możemy oczekiwać po środowisku wspomagającym tworzenie programów. Na szczęście przedstawiona powyżej operacja *iter* daje się łatwo zdefiniować:

```
iter(i:d..g, warunek, wart_pocz, operacja, co) =
    Jeśli d > g
        wart_pocz
    w przeciwnym przypadku
        jeśli warunek nie zachodzi dla i = d
            iter(i:d+1..g, warunek, wart_pocz, operacja, co)
        w przeciwnym przypadku (d<=g i warunek zachodzi dla i = d)
            iter(i:d+1..g, warunek, wart_pocz, operacja, co) operacja co[i/d],
```

gdzie zapis $co[i/d]$ oznacza jednoczesne, tekstowe podstawienie wartości wyrażenia d za wszystkie wystąpienia zmiennej i w wyrażeniu co.

5. Konkluzje

Niniejsza praca dotyczy komputerowych systemów wspomagających tworzenie poprawnych programów. Omówiliśmy w niej zagadnienia związane z wprowadzeniem do takich systemów złożonych typów danych, takich jak tablice. Nasze rozważania ograniczyliśmy jedynie do typów wbudowanych w język programowania. Te typy są trudniejsze do opisania (gdyż są specjalnie traktowane w samej definicji języka). W przypadku typów zdefiniowanych przez użytkownika wystarczy wykorzystać formalizm opisujący wywołania procedur [7] (bo to one będą stanowiły zasadniczy element definicji typu użytkownika).

W pracy pokazaliśmy na przykładzie rekordów i tablic jakie problemy można napotkać przy włączaniu strukturalnych typów danych do komputerowych środowisk wspomagających tworzenie programów. Pokazaliśmy też jak można te problemy rozwiązać na każdym z trzech poziomów: semantyki języka, wnioskowania o programach i specyfikowania programów. Przedstawiając operację *iter* chcieliśmy zwrócić uwagę na (często niedocenianą) rolę odpowiednich narzędzi językowych umożliwiających opisywanie struktur danych.

Bardzo interesującym i obiecującym kierunkiem dalszych badań, jest zastosowanie klas (wraz z dziedziczeniem i metodami wirtualnymi) do definicji typów użytkownika. Te zagadnienia będą tematem dalszych prac autora.

Literatura

[1] K.R. Apt. Ten Years of Hoare's Logic: A Survey - Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[2] D.L. Detlefs, K.R.M. Leino, G. Nelson, and J.B. Saxe. Extended static checking. Technical Report SRC 159, Compaq Systems Research Center, 1998.

[3] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, 1976.

[4] B. Dunin-Kęplicz, J. Jabłonowski, W. Łukaszewicz, and E. Madalińska-Bugaj. CAProDel: A system for computer aided program development. In *Proceedings of the Sixth International Conference on Software Engineering and Knowledge Engineering, SEKE'94, Jurmala, Latvia*, 1994.

[5] D. Gries. *The Science of Programming.* Springer, Berlin, 1981.

[6] M. Iglewski, M. Kubica, J. Madey, J. Mincer-Daszkiewicz, and K. Stencel. TAM'97: the Trace Assertion Method of Module Interface Specification. Reference Manual. Technical Report TR 97-01 (238), Uniwersytet Warszawski, Instytut Informatyki, 1997.

[7] J. Jabłonowski. Wspomagane komputerowo konstruowanie poprawnych programów. In *I Krajowa Konferencja Inżynierii Oprogramowania, KKIO'99, str. 301-310*, Kazimierz Dolny, 1999.

[8] J. Jabłonowski and M. Lipczyński. Praktyczne wykorzystanie formalnych systemów dowodzenia poprawności programów. In *II Krajowa Konferencja Inżynierii Oprogramowania, KKIO'2000, str. 47-54*, Zakopane, 2000.

[9] S. Owre, N. Shankar, and J.M. Rushby. PVS System Guide. PVS Prover Guide. PVS Language Reference. Version 2.3. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1999.