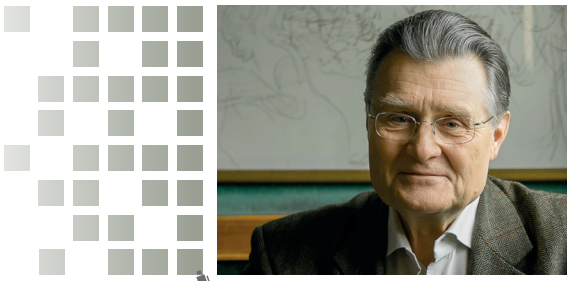
Zaproszenie do projektu

Przed laty rozpocząłem prace nad koncepcją języka programowania bazującego na metodzie inżynierii denotacyjnej. Postanowiłem ten niekomercyjny projekt, zupełnie odwracający kolej rzeczy w inżynierii oprogramowania, kontynuować – liczę na zainteresowanie i wsparcie polskiego środowiska programistów.

W każdej inżynierii, poza inżynierią oprogramowania, projektowanie nowego produktu rozpoczyna się od stworzenia opisu tego produktu w specyficznym dla danej inżynierii języku, który posługuje się z jednej strony wyspecjalizowanym aparatem pojęciowym, a z drugiej – aparatem matematycznym. Do tego dochodzi nierzadko rysunek techniczny. W tych trzech językach – branżowym, matematycznym i graficznym – powstaje opis przyszłego produktu uzupełniony o komentarze pisane w języku potocznym. Matematyczna część opisu, zwana zwykle „obliczeniami", stanowi gwarancję, że wytworzony na jej podstawie produkt będzie miał oczekiwane właściwości: most się nie zawali, skrzydło samolotu nie pęknie, instalacja chemiczna będzie prowadzić prawidłową syntezę. Oczywiście mogą zdarzyć się błędy na etapie projektu lub wykonania, więc przed oddaniem produktu do eksploatacji prowadzi się testy pozwalające na wykrycie ewentualnych zagrożeń. Później produkt jest oddawany do użytku i zwykle zachowuje się zgodnie z oczekiwaniami jego twórców.



prof. dr hab. Andrzej Jacek Blikle
Instytut Podstaw Informatyki PAN

W inżynierii oprogramowania inaczej

Przyszły program jest najczęściej opisywany w języku potocznym, co prawda wzbogaconym o terminologię techniczną, ale pozbawionym aparatu matematycznego, który dawałby gwarancję poprawności. To powoduje, że pierwsza wersja programu oddawana do testowania z reguły testów nie przechodzi. Wraca więc do programistów ze wskazaniami na błędy, którzy te błędy usuwają, ale też wprowadzają nowe. A nawet, gdyby nie wprowadzili, to w programie i tak pozostają błędy niewykryte podczas testowania, choćby z tego powodu, że niektóre ścieżki przebiegu programu zostały sprawdzone jedynie do miejsca, w którym wykryto błąd. Po usunięciu błędów program wraca więc do ponownego testowania, które wskazuje kolejne błędy. Ten proces powtarza się wielokrotnie, co generuje koszty stanowiące istotną część budżetu całego projektu. Ponadto, programiści wiedząc, że ich program będzie testowany, uznają nierzadko, że odpowiedzialność za poprawność programu ponoszą testerzy, a że pracują najczęściej pod silną presją czasu, mają tendencję do oddawania testerom programów napisanych niestarannie.

> *Warto też pamiętać o znanej prawdzie, że za pomocą testowania można jedynie wykryć błędy, ale nigdy nie można uzyskać pewności, że ich nie ma.*

Ten stan rzeczy powoduje, że użytkownik nabywający aplikację informatyczną jest z reguły zmuszany zaakceptować tzw. „wyłączenie odpowiedzialności" ze strony producenta aplikacji (ang. *disclaimer*). A oto typowy przykład takiego wyłączenia (nazwa firmy została zastąpiona słowem „Firma"):

O ile nie zaznaczono inaczej w „Warunkach dodatkowych", Usługi i Oprogramowanie są udostępniane w stanie, w jakim się znajdują („AS-IS"). W maksymalnym zakresie dozwolonym przez prawo, Firma wyłącza wszelkie gwarancje wyraźne lub domniemane, w tym domniemane gwarancje nienaruszania praw, przydatności handlowej i przydatności do określonego celu. Firma nie przyjmuje żadnych zobowiązań dotyczących treści zawartej w Usługach. Ponadto Firma nie gwarantuje, że:

a. Usługi lub Oprogramowanie spełnią wymagania Użytkownika, będą stale dostępne oraz że będą działały w sposób nieprzerwany, terminowy i bezbłędny;

b. efekty uzyskane w wyniku użycia Usług lub Oprogramowania będą skuteczne, dokładne i niezawodne;

c. jakość Usług i Oprogramowania spełni oczekiwania Użytkownika;

d. błędy lub usterki w Usługach lub Oprogramowaniu zostaną naprawione.

To wyłączenie pochodzi z 2018 r. i dotyczy nie małej, lokalnej firmy, ale dużej i międzynarodowej.

Czy ktoś mógłby sobie wyobrazić, że producent jakiegokolwiek nieinformatycznego produktu przemysłowego – np. samochodu, pralki, telewizora czy budynku –mógłby zażądać od swoich klientów zgody na podobne zrzeczenie się ich praw? Dlaczego więc w przemyśle IT jest to zjawisko powszechne?

Próba diagnozy

W mojej ocenie przyczyną tego stanu rzeczy jest brak dostatecznie rozwiniętych modeli i narzędzi matematycznych, które pozwalałyby uzyskiwać gwarancje funkcjonalnej poprawności programów. Zresztą ten brak ma wpływ nie tylko na produkty informatyczne, lecz także na podręczniki języków programowania, co znów pośrednio wpływa na jakość produktów.

Uważam, że na przykład wydany w 1960 r. podręcznik Algolu 60 – języka, który w dużej mierze wytyczył drogę rozwoju informatyki dla kilku pokoleń – znacznie przewyższał dzisiejsze podręczniki pod względem precyzji i kompletności. Po pierwsze, składnia Algolu 60 była opisana za pomocą gramatyk generacyjnych, a nie – jak to się czyni dziś – przez najczęściej niejasne przykłady. Po drugie, semantyka języka, choć nieodwołująca się do modeli matematycznych (nie były wówczas znane), była opisana przy użyciu dobrze zdefiniowanego aparatu pojęciowego: *zmienna, blok, widoczność zmiennej, procedura, parametr procedury, rekurencja*.

Podobnymi cechami charakteryzował się też powstały około dziesięciu lat później podręcznik języka Pascal. Niestety, nie można tego powiedzieć o dzisiejszych podręcznikach, których autorzy nie odróżniają wyrażenia od instrukcji, a instrukcji od deklaracji.

Zjawiska te są powszechne i dotyczą nie tylko języków programowania, lecz także narzędzi budowania systemów, takich jak np. Joomla! czy Drupal – świadczą o tym cieszące się niezmienną popularnością pomocowe internetowe fora, na których zdesperowani użytkownicy wymieniają się własnymi doświadczeniami. Do podręczników systemów mało kto zagląda, bo nie dość, że są nieprecyzyjne i niekompletne, to jeszcze dalece nieczytelne i to zarówno ze względu na swój język pozbawiony najczęściej aparatu pojęciowego, jak i na obszerność. Podczas gdy podręcznik Algolu 60 obejmował 237 str., a podręcznika Pascala – 166 str., podręcznik Pytona zawiera 696 str., systemu Access – 952 str., a prawie zapomnianego już dziś Delphi, który miał się stać uniwersalnym językiem programowania wszechczasów, przekracza 2000 stron. Fora samopomocy wypełniają się więc pytaniami typu „Hej, czy ktoś wie jak…?", na które zresztą – poza sytuacjami najprostszymi – najczęściej nikt nie udziela odpowiedzi (obserwacja własna).

Źródła problemów

W mojej ocenie niepowodzenie prób zbudowania formalnego i praktycznego zarazem systemu uzyskiwania gwarancji poprawności programów miało źródła w dwóch grupach problemów.

Pierwsza wiąże się z faktem, że zbudowanie takiego systemu dla konkretnego języka wymaga opisania jego semantyki

na gruncie matematyki. Począwszy od lat 70. XX w. dla matematyków było raczej oczywiste, że aby semantyka języka programowania miała praktyczny sens, musi być *kompozycyjna*, przez co rozumiemy, że znaczenie całości jest funkcją znaczeń jej części.

Semantyki kompozycyjne nazwano *denotacyjnymi* (znaczenie programu to jego denotacja) i przez wiele lat badano ich własności teoretyczne oraz podejmowano próby budowania takich semantyk dla istniejących języków. Wśród najbardziej znanych dokonań praktycznych należy wymienić formalne semantyki dla języków ADA oraz CHILL opisane w metajęzyku VDM. Inny eksperyment z tej grupy, choć na znacznie mniejszą skalę, to definicja podzbioru Pascala napisana przeze mnie w metajęzyku MetaSoft zbliżonym do VDM.

Niestety, żadna z tych prób nie doprowadziła do wprowadzenia modeli denotacyjnych do praktyki inżynierii oprogramowania. W moim przekonaniu nie mogło być inaczej, gdyż dla istniejących języków sensownych semantyk denotacyjnych zbudować się raczej nie da (bliższe uzasadnienie tego poglądu mogę przedstawić zainteresowanym). Przyczyną tego stanu rzeczy jest fakt, że języki programowania budowano rozpoczynając od składni, a nie od denotacji, czyli najpierw decydowano, jak będziemy wyrażać treści, a dopiero później, czym te treści będą. To oczywiście prosta konsekwencja historycznego faktu, że gdy zaczęto zastanawiać się nad tworzeniem semantyk, składnie języków już istniały.

W tym miejscu należy wyjaśnić, że przy projektowaniu składni budowniczowie języków mają zawsze na myśli jakieś jej znaczenie, jednakże myślą o nim w terminach raczej operacyjnych (sekwencja operacji maszynowych) niż denotacyjnych (kompozycja funkcjonalności). Dla przykładu, semantyka operacyjna bankowej procedury przelewu z jednego konta na drugie opisuje zmiany w pamięci operacyjnej i stałej komputera realizowane przez skompilowany kod, natomiast semantyka denotacyjna opisuje zmiany stanu bazy danych banku zrozumiałe dla bankowca lub właściciela konta.

Druga grupa problemów wiąże się z milczącym założeniem, że budowanie programów matematycznie poprawnych powinno przebiegać w kolejności – najpierw program, a później dowód jego poprawności. Ta kolejność jest oczywista w matematyce – najpierw hipoteza, a później jej dowód – jednak dla każdego inżyniera absurdalny musi być pomysł, aby najpierw zbudować most, a dopiero później wykonać jego projekt wraz z obliczeniami. Zasada „najpierw program, a później dowód poprawności" jest jednak nie tylko irracjonalna, lecz i praktycznie raczej niewykonalna, i to aż z dwóch powodów.

Po pierwsze, dowód jest z reguły dłuższy od twierdzenia, a twierdzenie o poprawności programu jest oczywiście dłuższe od samego programu. Skoro więc „praktyczne" programy liczą od dziesiątków tysięcy do milionów linii kodu, to „ręczne" dowodzenie ich poprawności nie wchodzi w grę. Z kolei systemów automatycznego dowodzenia poprawności programów, które weszłyby do powszechnego użycia w przemyśle IT, nie udało się zbudować choćby z tego powodu, że dla istniejących języków programowania nie stworzono matematycznych semantyk.

Sądzę jednak, że praktycznie ważniejszy jest powód drugi. Otóż programy, których poprawność staralibyśmy się udowodnić, z reguły poprawne nie są! Stąd metody dowodzenia poprawności miały w zasadzie funkcjonować jako narzędzia wykrywania błędów. Gdy dowód poprawności się „zacina", poprawiamy program, a następnie „uruchamiamy" dowód od nowa. Innymi słowy, najpierw robimy coś byle jak, by później to poprawiać. To chyba niezbyt racjonalne podejście.

Mój projekt

Podjąłem więc próbę zaproponowania narzędzi matematycznych pozwalających na zbudowanie języka programowania opartego na modelu denotacyjnym, a następnie zdefiniowania dla tego języka takich reguł programowania, które gwarantowałyby poprawność tworzonych programów. Zamiast więc najpierw pisać program „intuicyjnie", by później uzyskiwać (ograniczone) gwarancje jego poprawności za pomocą testowania, budujemy program w sposób, który tę poprawność gwarantuje. Tym ideom poświęciłem moje badania prowadzone w latach 70. i 80. XX w., nie doprowadziłem jednak wtedy do poziomu na tyle praktycznego, aby można było podjąć eksperymenty w obszarze inżynierii oprogramowania.

Dziś powracam do moich dawnych planów i badań, proponując dwie techniki odwracające tradycyjne sposoby postępowania.

Pierwsza z nich, którą nazwałem inżynierią denotacyjną, polega na projektowaniu języka programowania poczynając od denotacji (znaczeń), a nie od składni. Po opisaniu denotacji przyszłego języka w metajęzyku MetaSoft generuje się z jej opisu – w dużej mierze algorytmicznie – składnię, a ściśle mówiąc jej gramatykę. Okazuje się, że dla tak wygenerowanej składni mamy nie tylko zagwarantowane istnienie (jednoznacznie wyznaczonej) semantyki denotacyjnej, lecz także możemy algorytmicznie wygenerować jej definicję w języku MetaSoft. Ta ostatnia z kolei może być podstawą do wygenerowania kodu interpretera lub kompilatora.

Proponowaną technikę wraz z jej matematycznym uzasadnieniem opisałem w roboczej wersji mojej książki *A Denotational Engineering of Programming Languages*, dostępnej obecnie w wersji cyfrowej https://moznainaczej.com.pl/what-has-been-done/the-book. Moją metodę ilustruję przykładem projektowania języka programowania Lingua, oferującego techniki programowania sekwencyjnego, takie jak konstruktory strukturalne oraz procedury funkcyjne i imperatywne wraz z rekursją. Opisałem też zarys środowiska dla SQL tzw. API. Dla Lingua w wersji bez SQL studenci mojego kursu prowadzonego na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego (w 2020 r.) napisali prosty interpreter w języku OCaml.

Podjąłem też próbę pokazania, jak mógłby wyglądać podręcznik programisty napisany dla języka wyposażonego w semantykę denotacyjną (patrz https://moznainaczej.com.pl/what-has-been-done/on-writing-user-manuals). Co warte podkreślenia, w tym podręczniku nie zakładam znajomości technik denotacyjnych przez czytelnika, pokazuję jednak, jak można ustrukturalizować sam podręcznik, a także uczynić go zarówno pełnym, jak i zwartym dzięki temu, że te techniki znane są jego autorowi. Język Lingua, poza tym że ma semantykę denotacyjną, jest wyposażony w konstruktory programów gwarantujące poprawność tych ostatnich względem zawartej w nich (składniowo) specyfikacji.

Druga z moich technik, którą nazwałem programowaniem walidacyjnym, polega na budowaniu programów za pomocą konstruktorów gwarantujących poprawność. Rozpoczynamy od programów bardzo prostych, których poprawność jest łatwa do udowodnienia, a następnie, krok za krokiem, budujemy z nich programy złożone. We wspomnianej książce opisałem matematyczne podstawy programowania walidującego oraz podałem dwa przykłady budowania bardzo prostych, ale dość nieoczywistych, programów poprawnych. Jeden z nich to program napisany w 1970 r. przez informatyka norweskiego Ole-Johana Dahla, współtwórcę idei programowania obiektowego:

```
q := 1;
while q ≤ n do q:=4*q od
y := n;
p := 0;
while q > 1
  do
    q:=q/4;
    if p+q≤y
      then p:=p/2+q; y:=y-p-q else p:=p/2
    fi
  od
```

Na pierwszy rzut oka raczej nie widać, że końcowa wartość zmiennej p jest równa części całkowitej pierwiastka kwadratowego z n. Nie jest też łatwo ten fakt udowodnić, bo trzeba znaleźć wcale nieoczywisty niezmiennik drugiej z pętli. Natomiast techniką programowania walidacyjnego wyprowadza się program Dahla w kilku dość oczywistych krokach, a te kroki stanowią jednocześnie dowód poprawności programu.

Co dalej?

Obecnie prowadzę prace badawcze z kilkoma moimi kolegami z Wydziału MIM UW nad wyposażeniem Lingua w narzędzia do programowania obiektowego i (niezależnie) współbieżnego.

Dalsze kierunki rozwoju projektu widzę w dwóch obszarach:

1. **W obszarze inżynierskim:**
   - Zbudowanie środowiska projektanta języków programowania metodą inżynierii denotacyjnej. Takie środowisko mogłoby obejmować nie tylko inteligentny edytor, lecz także generatory parserów, interpreterów, a może i kompilatorów.
   - Zbudowanie środowiska programisty dla języków z grupy Lingua. Ono również obejmowałoby nie tylko inteligentny edytor, lecz także narzędzia wspierające budowanie programów poprawnych na bazie zadanych konstruktorów.
   - Tworzenie języków dziedzinowych (ang. *Domain Specific Languages*) dla różnych zastosowań, np. do pisania mikroprogramów sterujących maszynami autonomicznymi czy też dla operacji bankowych.
   - Eksperymenty z budowaniem praktycznych programów ze zweryfikowaną poprawnością. Myślę, że na początek mogłyby to być mikroprogramy ze względu na ich stosunkowo niewielki wolumen przy wysoce krytycznym problemie poprawności.
2. **W obszarze teoretycznym:**
   - Rozwój denotacyjnych modeli języków programowania obejmujących zaawansowane narzędzia programistyczne, takie jak np. obiektowość, współbieżność czy typy polimorficzne.
   - Tworzenie modeli denotacyjnych dla specyficznych języków dziedzinowych, np. do pisania mikroprogramów.
   - Budowanie reguł programowania walidującego dla tak rozbudowanych lub specjalistycznych języków.

- Opracowanie zasad pisania zwięzłych i kompletnych podręczników języków programowania.
- Opracowanie nowych metod uczenia programowania i to nie tylko specjalistów, lecz także nieinformatyków, np. użytkowników języków dziedzinowych, czy wręcz młodzieży szkolnej.

To oczywiście nie wszystkie możliwe kierunki rozwoju projektu, gdyż każde jego wzbogacenie generuje kolejne problemy do rozwiązania. Jest tu więc wiele miejsca na prace naukowe, a w tym licencjackie, magisterskie i doktorskie.

Przesłanie Urbi et Orbi

Proponowane przeze mnie metody mogą spotkać się z różnymi ocenami, jednakże jedno jest chyba pewne – dość daleko odbiegają od dzisiejszej praktyki. To, co proponuję, to poważna zmiana, która siłą rzeczy rodzi zarówno przeciwników, jak i zwolenników. Przeciwników należy przekonać, a zwolenników – zdobyć. I oczywiście trzeba zacząć od drugiego z tych zadań.

Jednakże, aby je zrealizować, trzeba potencjalnym zwolennikom albo dać do ręki choćby najprostszą, ale praktyczną, wersję Lingua, albo też zachęcić ich do zbudowania własnej. Pierwsza droga wydaje się mało prawdopodobna, bo wymagałaby znalezienia inwestora, który sfinansowałby budowę prototypu czegoś dziwnego i nieznanego, na co raczej liczyć nie można. Pozostaje więc druga droga, co oznacza budowę Lingua przez ochotników dla ochotników, a więc założenie, że będzie to produkt bezpłatny i ogólnie dostępny, jak np. Linux, Joomla! czy Drupal. „Ogólnie dostępny" nie może jednak w tym przypadku oznaczać dostępności do jego tworzenia (open source), bo to mogłoby prowadzić do matematycznie niepoprawnych konstrukcji i w rezultacie do niepoprawnych reguł budowania programów.

> *Społeczność budowniczych Lingua – gdyby powstała – powinna wypracować reguły przyjmowania nowych członków i udzielania im prawa do włączenia się w tok prac implementacyjnych.*

Ta zasada nie oznacza oczywiście, że należałoby jakkolwiek ograniczać badania naukowe, które z istoty rzeczy powinny podlegać naturalnej ocenie przez środowisko akademickie.

Co mogę zaoferować?

Gdyby znalazła się grupa osób zainteresowanych rozwijaniem projektu, mogę obiecać pomoc merytoryczną w tym obszarze, a także – być może – jakąś pomoc organizacyjną. Mogę się podjąć promotorstwa prac magisterskich i doktorskich, a także przeprowadzenia kursów na różnych poziomach szczegółowości w zależności od tego, czy słuchaczami byliby teoretycy zamierzający dalej rozwijać denotacyjny model dla Lingua, czy też praktycy, zainteresowani budowaniem narzędzi wspomagających pracę projektanta języka lub programisty.

W dalszej perspektywie można by też pomyśleć o powołaniu start-upu, który oferowałby wdrożenia środowisk dla projektantów języków i dla programistów oraz opiekę na tymi narzędziami. Analogicznie jak w przypadkach np. Linuksa, same narzędzia pozostawałyby bezpłatne, natomiast komercjalizacji podlegałoby wsparcie w ich stosowaniu i aktualizowaniu. Firma mogłaby też oferować niezawodne produkty IT zbudowane zgodnie z technikami programowania walidującego.

Osoby zainteresowane porozmawianiem o szczegółach projektu proszę o kontakt ze mną:

andrzej.blikle@moznainaczej.com.pl

Ponieważ zdaję sobie sprawę, że mój projekt może rodzić wiele znaków zapytania, deklaruję, że taki kontakt nie będzie rodzić żadnych zobowiązań po stronie mojego rozmówcy.