Poprawność algorytmów

* Na początku lat 60. jeden z amerykańskich statków kosmicznych z serii Mariner wysłany na Wenus zaginął na zawsze kosztem milionów dolarów z powodu błędu w programie komputerowym kontroli lotu.

* W 1981 roku jedna ze stacji telewizyjnych relacjonujących wybory w prowincji Quebec w Kanadzie, przez swoje błędne programy komputerowe przekonała, że w rzeczywistości przewodniczy jej mała partia, początkowo uważana za pozbawioną jakichkolwiek szans. Te informacje i wynikające z nich odpowiedzi komentatorów zostały przekazane milionom widzów.

* W serii incydentów w latach 1985-1987 kilku pacjentów otrzymało ogromne przedawkowanie promieniowania z systemów radioterapii Therac-25; troje z nich zmarło z powodu powstałych komplikacji. Sprzętowe blokady bezpieczeństwa z poprzednich modeli zostały zastąpione testami bezpieczeństwa oprogramowania, ale wszystkie te incydenty wiązały się z błędami programistycznymi.

* Kilka lat temu pewna duńska dama otrzymała, około swoich 107 urodzin, skomputeryzowany list od lokalnych władz szkolnych z instrukcjami dotyczącymi procedury rejestracji w pierwszej klasie szkoły podstawowej. Okazało się, że na pole „wiek” w bazie danych przypisano tylko dwie cyfry.

* Na przełomie tysiącleci problemy z oprogramowaniem stały się nagłówkami wiadomości wraz z tak zwanym problemem roku 2000, czyli błędem Y2K. Obawiano się, że 1 stycznia 2000 r. rozpęta się piekło, ponieważ komputery używające dwóch cyfr do przechowywania lat błędnie zakładają, że rok podany jako 00 to 1900, podczas gdy w rzeczywistości był to rok 2000. z perspektywy czasu, całkiem udane) wysiłki mające na celu poprawienie tych programów musiały zostać podjęte przez firmy programistyczne na całym świecie.

To tylko kilka z licznych opowieści o błędach oprogramowania, z których wiele zakończyło się katastrofami, często z utratą życia. Nie sposób przecenić wagi kwestii poprawności. Przez cały czas naiwnie zakładaliśmy, że algorytmy i programy, które piszemy, robią dokładnie to, co zamierzamy zrobić. To nie ma żadnego uzasadnienia

Błędy językowe

Jeden z najczęstszych rodzajów błędów występujących przy przygotowywaniu programów komputerowych wynika z nadużywania składni języka programowania. Spotkaliśmy się z nimi w poprzednim rozdziale. Zapis:

for Y from 1 until N do

zamiast:

for Y from 1 to N do

jak wymaga tego język, jest zły, ale nie jest to błąd algorytmu. Błędy składniowe są jedynie uciążliwym przejawem tego, że algorytmy realizowane przez komputer muszą być prezentowane w stroju formalnym. Kompilatory i interpretery są tworzone w celu wykrywania błędów składniowych i powiadamiają programistę, który zazwyczaj będzie w stanie je poprawić przy niewielkim wysiłku. Co więcej, i tutaj kompilatory mają przewagę nad interpreterami, sprytny kompilator sam spróbuje poprawić pewne rodzaje błędów składniowych, umożliwiając pożądane tłumaczenie na język maszynowy. Kompilator nie jest ograniczony, tak jak interpreter, do przeglądania jednej linii lub jednej instrukcji programu na raz. Zwykle jest również zaprogramowany do wykrywania bardziej subtelnych

błędów, które zamiast naruszać lokalne reguły składniowe języka, powodują sprzeczności między prawdopodobnie odległymi częściami programu, zwykle między definicją a instrukcją operacyjną. Przykłady obejmują operacje arytmetyczne zastosowane do zmiennych nienumerycznych, odwołania do 150. elementu wektora, którego indeksy zostały zdefiniowane w zakresie od 1 do 100, oraz wywołania podprogramów z niewłaściwą liczbą parametrów. Wszystko to jednak oznacza również nieprawidłowe użycie języka. Niezależnie od tego, czy kompilator lub interpreter wykryje taki błąd z wyprzedzeniem, próba uruchomienia programu zakończy się niepowodzeniem, gdy zostanie osiągnięta obraźliwa część; program zostanie przerwany, to znaczy zostanie zatrzymany, a użytkownikowi wyświetli się odpowiedni komunikat. W przeciwieństwie do błędów omówionych w następnym podrozdziale i pomimo potencjalnie nieprzyjemnego charakteru awarii programu, błędy językowe nie są uważane za najpoważniejsze. Często są one wykrywane automatycznie i zazwyczaj można je stosunkowo łatwo skorygować.

Błędy logiczne

Przypomnijmy algorytm liczenia „pieniądze" wprowadzony w Części 2. Problem polegał na liczeniu zdań zawierających wystąpienia słowa „pieniądze". Rozwiązanie polegało na przeprowadzeniu wyszukiwania słowa „pieniądze", a następnie wyszukania końca zdania, które umownie jest zawsze oznaczane w tekście wejściowym przez kombiację „. " , a mianowicie kropka, po której następuje spacja. Po pomyślnym przeprowadzeniu obu wyszukiwań początkowo wyzerowany licznik jest zwiększany, a wyszukiwanie „pieniądze" jest wznawiane od początku następnego zdania . Co by się stało, gdyby algorytm użył „." (bez spacji) zamiast „. "? Załóżmy na razie, że nie mówimy o algorytmie, ale o jego wersji formalnej, jako programie nie zawierającym błędów językowych. Oczywiste jest, że nowa wersja, która różni się od oryginału jedynie brakiem miejsca, również nie zawiera błędów językowych. Dla obserwatora, łącznie z kompilatorami i interpreterami, nowy program jest doskonały. Nie tylko nie ma dostrzegalnych błędów składniowych, ale za każdym razem, gdy zostanie uruchomiony na tekście wejściowym, program posłusznie zatrzymuje się i wyświetla liczbę jako końcową wartość swojego licznika. Oczywiście w nowym programie jest błąd. Wynika to z faktu, że kropki mogą występować w zdaniach. Rozważ następujące:

Całkowita suma pieniędzy na moim koncie bankowym to 322.56 dolarów, naprawdę niezwykła suma, biorąc pod uwagę mój talent do zarabiania pieniędzy. Jestem bogatym człowiekiem.

Po włączeniu tego dwuzdaniowego tekstu nasza zmodyfikowana wersja wygeneruje 2, mimo że słowo „pieniądze" pojawia się tylko w pierwszym zdaniu. Program jest oszukany przez przecinek dziesiętny pojawiający się w 322.56 USD. Nowy program jest poprawny pod względem językowym i faktycznie rozwiązuje problem algorytmiczny, ale niestety nie dokładnie ten, który zamierzaliśmy rozwiązać. Program zawiera to, co nazwiemy błędem logicznym, co skutkuje nie niepoprawnym składniowo lub bezsensownym programem, ale programem, który robi coś innego niż to, do czego był przeznaczony. Błędy logiczne mogą być notorycznie nieuchwytne. Chociaż może nie być zbyt trudno zauważyć, że spacja została pominięta w „. " w programie do liczenia pieniędzy, następujący błąd nie jest tak łatwy do znalezienia. Załóżmy, że różne wskaźniki do tekstu są używane do wyszukiwania „pieniądze" i „. " Szukaj. Oczywiście raz ". ", a licznik został zwiększony, pierwszy wskaźnik powinien zostać przekierowany na pozycję drugiego przed wznowieniem wyszukiwania słowa „pieniądze". Niezastosowanie się do tego stanowi błąd logiczny, który również da 2 po uruchomieniu w poprzednim przykładzie, ale z innego powodu; tym razem granice zdań są prawidłowo wykrywane, ale licznik jest zwiększany dwukrotnie w obrębie pierwszego zdania. (Dlaczego?) Takie błędy nie wskazują, że coś jest nie tak z programem per se, ale że coś jest nie tak z połączeniem programu i konkretnego problemu algorytmicznego; program, który sam w sobie jest w porządku, nie rozwiązuje poprawnie tego problemu. Błędy logiczne mogą być spowodowane niezrozumieniem semantyki języka programowania

(„Myślałem, że X*Y oznacza X podniesione do potęgi Y , a nie X razy Y ” lub „Byłem pewien, że kiedy pętla formy ponieważ Y od 1 do N jest uzupełnione, wartość Y to N, a nie N + 1”), w takim przypadku możemy nazwać je błędami semantycznymi. Jednak o wiele bardziej typowe jest napotkanie „prawdziwych” błędów logicznych; to znaczy błędy w procesie logicznym używanym przez projektanta algorytmu do rozwiązania problemu algorytmicznego. Nie mają one nic wspólnego z programem napisanym później w celu zaimplementowania algorytmu. Stanowią one wady samego algorytmu, gdy są rozważane jako proponowane rozwiązanie problemu. Są to błędy algorytmiczne i to właśnie one nas tutaj interesują. Nieprzekierowanie licznika „pieniądze” do następnego zdania jest błędem algorytmicznym. Czy zatem błędne powiązanie we wczesnej wersji Rysunku 2.4 przedstawionej na Rysunku 5.1 dotyczyło jednego z wyjść z „czy pensja P jest wyższa niż Q?” na „przejść P do następnego pracownika” zamiast „czy P jest na końcu listy?”

Komputery nie błądzą

Analogia między algorytmami a recepturami zawodzi, jeśli chodzi o kwestie poprawności. Kiedy próba gotowania lub pieczenia nie powiedzie się, mogą być dwa powody:

1. winę ponosi „sprzęt” lub

2. przepis jest nieprecyzyjny i niejasny.

W przeważającej części pierwszy z nich jest tak naprawdę powodem, zwłaszcza po tym, jak zdecydowaliśmy, jak rzeczywiście zrobiliśmy, że kucharze i piekarze są częścią sprzętu. Ale jeśli są problemy z przepisem, a nie z piekarzem, piekarnikiem czy sztućcami, to zazwyczaj mają one do czynienia z założeniami autora na temat kompetencji jego użytkowników. „Ubijaj białka na pieniste” wymaga od piekarza pewnej wiedzy na temat piany jajecznej, bez której efektem pieczenia może nie być mus czekoladowy, ale czekoladowy bałagan! W przeciwieństwie do przepisów, algorytmy napisane do wykonania przez komputer kończą w formalnym, jednoznacznym języku programowania, który prawie całkowicie eliminuje przyczynę (2). Co więcej, powód (1) można również odrzucić. Ogólnie rzecz biorąc, komputery nie popełniają błędów! Błąd sprzętowy to taka rzadkość we współczesnych komputerach, że gdy nasz wyciąg bankowy jest błędny i bankier mamrocze coś w tym sensie, że komputer się pomylił, możemy być pewni, że to nie komputer popełnił błąd - to chyba jeden pracowników banku. Albo w jednym z programów wprowadzono nieprawidłowe dane, albo sam program, oczywiście napisany przez człowieka, zawierał błąd. Nieprawidłowo działający program nie jest wynikiem problemu z komputerem. Jeśli dane wejściowe zostaną sprawdzone i okaże się, że są poprawne, problem dotyczy programu i jego podstawowego algorytmu.

Testowanie i debugowanie

Błędy algorytmiczne mogą pozostać niewykryte przez wieki. Czasami nigdy nie są wykrywane. Jest całkiem możliwe, że dane wejściowe, dla których błąd generuje nieprawidłowe dane wyjściowe, po prostu nie wystąpią w okresie życia algorytmu. Ewentualnie takie dane wejściowe mogą się rzeczywiście pojawić, ale nieprawidłowe dane wyjściowe mogą nigdy nie zostać zauważone. Niektóre błędy logiczne pojawiają się, gdy procesor nie może wykonać instrukcji z jakiegoś nieoczekiwanego powodu. Na przykład próba podzielenia X przez Y nie powiedzie się, jeśli Y będzie w tym czasie równe 0. Podobnie niepowodzenie będzie wynikać z próby zejścia w dół drzewa z węzła, który akurat jest liściem (czyli nie ma dokąd zejść). Nie są to błędy językowe, a kompilator generalnie nie będzie w stanie ich wcześniej wykryć. Nazywa się je błędami czasu wykonania i wynikają z błędów logicznych w projekcie algorytmu. Często wada polega po prostu na zapomnieniu o odrębnym traktowaniu specjalnych przypadków „z pogranicza”, takich jak zera (w liczbach) i liście (w drzewach). Projektant może wypróbować algorytm na kilku typowych i nietypowych danych wejściowych i nie znaleźć błędu.

W rzeczywistości programista zwykle testuje program na wielu wejściach, czasami nazywanych zestawami testowymi, i stopniowo usuwa z niego błędy językowe i większość błędów logicznych. Nie może być jednak pewien, że program (i leżący u jego podstaw algorytm) jest całkowicie wolny od błędów, po prostu dlatego, że większość problemów algorytmicznych ma nieskończone zestawy danych wejściowych, a zatem nieskończenie wiele kandydujących zestawów testowych, z których każdy może ujawnić nowy błąd. Błędy logiczne, jak ktoś kiedyś powiedział, są jak syreny. Sam fakt, że ich nie widziałeś, nie oznacza, że nie istnieją. Proces wielokrotnego wykonywania algorytmu lub uruchamiania programu w celu znalezienia i wyeliminowania błędów nazywa się debugowaniem. Nazwa ma ciekawą historię. Jeden z pierwszych komputerów, które zbudowano, pewnego dnia przestał działać, a później okazało się, że w kluczowej części obwodu zablokował się duży owad. Od tego czasu błędy, zwykle błędy logiczne, są pieszczotliwie nazywane błędami. Debugowanie programu, zwłaszcza złożonego i długiego, może być dość dużym przedsięwzięciem. Nawet jeśli zaobserwuje się, że program generuje nieprawidłowe dane wyjściowe w określonym przypadku testowym, może nie być dostępnych informacji o źródle błędu. Istnieje jednak kilka technik zawężania możliwości. Wersję programu można uruchomić ze sztucznie wstawionymi instrukcjami dla wydruków pośrednich. Pokazują one częściowe wyniki i wartości debuggera podczas wykonywania. Alternatywnie, jeśli program jest zinterpretowany, a nie skompilowany, jego wykonanie można śledzić wiersz po wierszu, umożliwiając wykrycie podejrzanych wartości pośrednich, zwłaszcza podczas pracy z wyświetlaczem interaktywnym. Wspomnieliśmy również o środowiskach programistycznych. Obsługują one wiele rodzajów narzędzi testujących i symulacyjnych, aby pomóc projektantowi algorytmów i programiście w uzyskaniu właściwego działania. Należy jednak ponownie podkreślić, że żadna z tych metod nie gwarantuje wolnych od błędów algorytmów, które generują prawidłowe dane wyjściowe na dowolnym legalnym wejściu. Jak ktoś kiedyś to ujął, testowanie i debugowanie nie może służyć do wykazania braku błędów w oprogramowaniu, a jedynie ich obecności.

Nieskończone pętle

Nieprawidłowość algorytmu jako rozwiązania problemu algorytmicznego może zatem objawiać się albo wykonaniem, które kończy się normalnie, ale z nieprawidłowymi danymi wyjściowymi, albo przerwaniem wykonania. Jakby co gorsza, algorytm uruchomiony na jednym z legalnych danych wejściowych problemu może w ogóle się nie zakończyć! Oczywiście jest to również błąd. Nieskończone obliczenia lub nieskończone pętle, jak się je czasami nazywa, mogą mieć charakter oscylacyjny, powtarzający w kółko pewną liczbę instrukcji na tych samych wartościach, lub nieoscylacyjny, ale rozbieżny charakter, co skutkuje coraz większym lub coraz mniejszym wartości. Przykładem oscylującej pętli jest procedura wyszukiwania, w której nie ma instrukcji do przekazania odpowiedniego wskaźnika; algorytm kontynuuje wyszukiwanie w tym samym obszarze struktury danych. Przykładem rozbieżności jest pętla, która zwiększa X o 1 w każdym przejściu i która jest instruowana, aby zakończyć, gdy X osiągnie 100. Jeśli pętla zostanie błędnie osiągnięta z początkową wartością 17,6 w X, pętla „chybi” 100 i zwiększaj X w nieskończoność. Oczywiście, prawdziwe komputery mają ograniczoną pamięć i generalnie przerywają programy drugiego rodzaju, gdy wartość X przekroczy pewne ostateczne maksimum, ale algorytm, na którym oparty jest program, mimo to dopuszcza nieskończoną pętlę. Testowanie i debugowanie może również pomóc w wykrywaniu potencjalnych nieskończonych pętli. Wypisując wartości pośrednie, debugger może zauważyć podejrzane oscylacje lub nienormalny wzrost lub spadek wartości, które pozostawione bez zmian mogą prowadzić do braku zakończenia. Tak jak poprzednio, zawsze jest więcej danych wejściowych niż możemy przetestować, dlatego żadna taka metoda nie gwarantuje znalezienia wszystkich potencjalnych nieskończonych pętli. Dla nich brak zakończenia jest błogosławieństwem, a zakończenie wskazuje na obecność błędu. Na razie jednak poprawny algorytm musi kończyć się normalnie na wszystkich legalnych danych wejściowych i za każdym razem generować właściwe dane wyjściowe.

Poprawność częściowa i całkowita

Jak omówiono w Części 1, problem algorytmiczny można zwięźle podzielić na dwie części:

1. specyfikację zestawu danych wejściowych prawnych; oraz

2. związek między wejściami a pożądanymi wyjściami.

Na przykład może być wymagane, aby każdy wpis prawny składał się z listy L słów w języku angielskim. Relacja między danymi wejściowymi a pożądanymi wynikami może określać, że wynik musi być listą zawierającą słowa w L posortowane w rosnącym porządku leksykograficznym. W ten sposób określiliśmy problem algorytmiczny, który wymaga algorytmu A, który sortuje każdą listę legalnych danych wejściowych L. Aby ułatwić precyzyjne potraktowanie problemu poprawności dla algorytmów, badacze rozróżniają dwa rodzaje poprawności, w zależności od tego, czy zakończenie jest, czy nie jest zawarty. W jednym przypadku zakłada się a priori, że program się kończy, a w drugim tak nie jest. Mówiąc dokładniej, mówi się, że algorytm A jest częściowo poprawny (w odniesieniu do definicji legalnych danych wejściowych i pożądanego związku z wynikami), jeśli dla każdego legalnego wejścia X, jeśli A kończy się po uruchomieniu na X, to określona relacja zachodzi między X i wynikowy zestaw wyjściowy. Zatem częściowo poprawny algorytm sortowania może nie kończyć się na wszystkich dozwolonych listach, ale zawsze, gdy tak się dzieje, wynikiem jest poprawnie posortowana lista. Mówimy, że A kończy działanie, jeśli zatrzyma się po uruchomieniu na jednym z dozwolonych danych wejściowych. Oba te pojęcia wzięte razem - częściowa poprawność i zakończenie - dają całkowicie poprawny algorytm, który poprawnie rozwiązuje problem algorytmiczny dla każdego legalnego wejścia: proces uruchamiania A na dowolnym takim wejściu X rzeczywiście kończy się i daje wyniki spełniające pożądaną zależność

Konieczność udowodnienia poprawności

Teraz wiemy dokładnie, co chcielibyśmy ustalić w obliczu problemu algorytmicznego i proponowanego rozwiązania, i dysponujemy różnymi technikami testowania i debugowania, które mogą nam w tym pomóc. Jednak żadna z tych technik nie jest niezawodna i podobnie jak przykłady przytoczone na początku części, folklor informatyczny pełen jest opowieści o katastrofach, niektóre z nich śmiertelne, inne powodujące utratę niewiarygodnych ilości pieniędzy, a wszystkie następujące od błędów algorytmicznych, zwykle w dużych i złożonych systemach oprogramowania. Twierdzenie, że dany algorytm jest poprawny w odniesieniu do problemu algorytmicznego, jest być może mniej głębokie niż twierdzenie, że syreny są wyimaginowane, ale może być tak ważne, że zależy od tego wiele żyć lub fortun. Wystarczy pomyśleć o systemach komputerowych, które kontrolują broń nuklearną lub wielomilionowe transakcje, aby docenić ten punkt. Niepokój budzą jednak nie tylko liczne nagłaśniane i niepublikowane przypadki błędów. Powszechnie uważa się, że ponad 70% (!) wysiłku i kosztów opracowania złożonego systemu oprogramowania jest w taki czy inny sposób przeznaczane na korygowanie błędów. Obejmuje to opóźnienia spowodowane błędnie przyjętymi specyfikacjami (tj. niejasnymi i nieprecyzyjnymi definicjami problemów algorytmicznych), szeroko zakrojone testowanie i debugowanie samych algorytmów, a co najgorsze, zmiany i przepisywanie już działających systemów (ogólnie określane jako konserwacja), ponieważ wynik nowo odkrytych błędów. Odnosząc się do dużych systemów oprogramowania używanych komercyjnie, sytuację opisano ładnie, mówiąc, że oprogramowanie jest udostępniane do użytku nie wtedy, gdy wiadomo, że jest poprawne, ale gdy tempo odkrywania nowych błędów spada do poziomu, który menedżerowie uznają za akceptowalny. Ta sytuacja jest wyraźnie zła. Potrzebujemy sposobów na udowodnienie, że algorytm jest bez wątpienia poprawny. Nikt nie pyta, czy może istnieć jakiś „nieodkryty" trójkąt równoboczny o nierównych kątach. Ktoś udowodnił raz na zawsze, że wszystkie trójkąty równoboczne mają jednakowe kąty i od tej pory wątpliwości nie pozostały. Czy można coś zrobić, aby ułatwić takie dowody? Czy sam

komputer może pomóc zweryfikować poprawność naszych algorytmów? Właściwie to, czego najbardziej chcielibyśmy, to automatyczny weryfikator; mianowicie jakiś rodzaj superalgorytmu, który przyjąłby jako dane wejściowe opis problemu algorytmicznego P i algorytm A, który jest proponowany jako rozwiązanie, i określałby, czy rzeczywiście A rozwiązuje P. Być może wskazałby również błędy, jeśli odpowiedź brzmiała nie. Nie można skonstruować takiego weryfikatora. Na razie jednak zignorujmy kwestię skorzystania z pomocy komputera. Czy możemy sami udowodnić poprawność naszych algorytmów? Czy jest jakiś sposób, w jaki możemy wykorzystać formalne, matematyczne techniki do realizacji tego celu? Tutaj mamy lepsze wieści.

Niezmienniki i zbieżności

Rzeczywiście istnieją metody weryfikacji programu. W rzeczywistości, w pewnym sensie technicznym, każdy poprawny algorytm można rygorystycznie wykazać, że jest poprawny! Zanim zaczniemy ilustrować przykładem, powiedzmy coś o samych metodach dowodowych. Próbując ustalić częściową poprawność, nie jesteśmy zainteresowani wykazaniem, że pewne pożądane rzeczy się zdarzają, ale wykazaniem, że pewne niepożądane rzeczy się nie zdarzają. Nie obchodzi nas, czy wykonanie kiedykolwiek osiągnie punkt końcowy, ale jeśli tak się stanie, nie będziemy w sytuacji, w której wyniki będą różnić się od oczekiwanych. W związku z tym chcemy uchwycić zachowanie algorytmu poprzez dokładne stwierdzenie, co robi w określonych momentach. Aby udowodnić częściową poprawność, dołączamy zatem twierdzenia pośrednie do różnych punktów kontrolnych w tekście algorytmu. Dołączenie asercji do punktu kontrolnego oznacza, że wierzymy, że ilekroć wykonanie osiągnie dany punkt, w każdym wykonaniu algorytmu na dowolnym legalnym wejściu, asercja będzie prawdziwa. Obejmuje to oczywiście punkty, które są osiągane wiele razy w ramach jednego wykonania, w szczególności te w pętlach. Z tego powodu takie twierdzenia są powszechnie nazywane niezmiennikami; pozostają prawdziwe bez względu na to, jak często są osiągane. Na przykład algorytm sortowania może być taki, że w pewnym momencie tekstu sortowana jest dokładnie połowa listy wejściowej, w którym to przypadku możemy dołączyć do tego punktu twierdzenie „połowa listy jest posortowana”. Bardziej typowe jest jednak to, że niezmienniki zależą od wartości zmiennych dostępnych w punkcie kontrolnym. W związku z tym do pewnego momentu możemy dołączyć twierdzenie, że „częściowa lista od pierwszej lokalizacji do X jest posortowana”, gdzie X jest zmienną, która zwiększa się w miarę sortowania większej ilości listy. Początkowe stwierdzenie, a mianowicie to, które jest dołączone do punktu początkowego algorytmu, jest zwykle formułowane w celu uchwycenia wymagań dotyczących legalnych danych wejściowych, i podobnie, końcowe stwierdzenie, związane z punktem końcowym, ujmuje pożądaną relację wyników do wejść . Załóżmy teraz, że możemy ustalić, że wszystkie załączone przez nas twierdzenia są rzeczywiście niezmiennikami, co oznacza, że są prawdziwe, gdy tylko zostaną osiągnięte. Wtedy w szczególności ostateczne stwierdzenie jest również niezmiennikiem. Ale to oznacza, że algorytm jest częściowo poprawny. Dlatego wszystko, co musimy zrobić, to ustalić niezmienność naszych twierdzeń. Odbywa się to poprzez ustalenie pewnych lokalnych własności naszych stwierdzeń, czasami nazywanych warunkami weryfikacji, tak że przechodzenie lokalnie od punktu kontrolnego do punktu kontrolnego nie powoduje żadnych naruszeń własności niezmienności. Takie podejście do udowadniania poprawności jest czasami nazywane niezmienną metodą asercji lub metodą Floyda, od nazwiska jednego z jej wynalazców. Jak zabieramy się za wybór punktów kontrolnych i pośrednich asercji oraz jak ustalamy warunki weryfikacji? Przykład podany w następnej sekcji powinien rzucić nieco światła na te pytania. Przechodząc od częściowej poprawności do zakończenia, naszym głównym interesem jest pokazanie, że w końcu wydarza się coś dobrego (nie, że złe rzeczy się nie zdarzają); mianowicie, że algorytm rzeczywiście osiąga swój punkt końcowy i kończy pomyślnie. Aby udowodnić takie stwierdzenie, używamy punktów kontrolnych jak poprzednio, ale teraz znajdujemy pewną ilość zależną od zmiennych algorytmu i struktur danych i pokazujemy, że jest ona zbieżna. Rozumiemy przez to, że ilość spada w miarę postępu egzekucji z jednego punktu

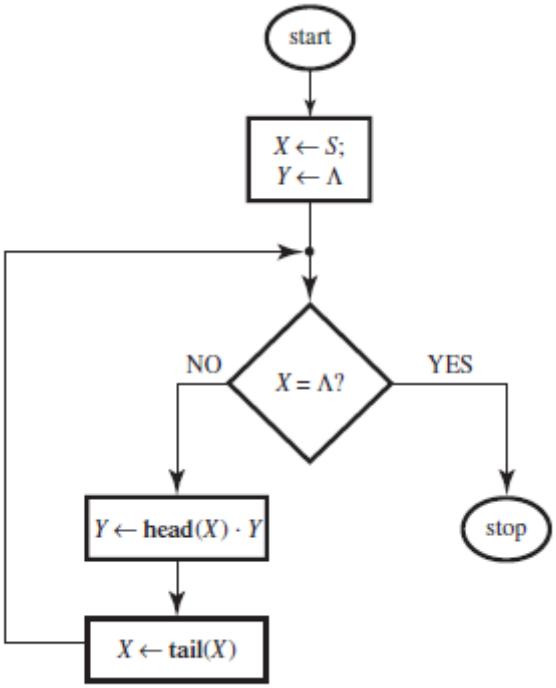
kontrolnego do drugiego, ale nie może się zmniejszać w nieskończoność – musimy pokazać, że istnieje pewna granica, poniżej której nigdy nie może zejść. Nie ma więc możliwości, aby algorytm działał w nieskończoność, ponieważ zbieżność, jak to się czasem nazywa, musiałaby wtedy zmniejszać się w nieskończoność, przecząc temu ograniczeniu. Na przykład w algorytmie sortowania liczba elementów, które nie znajdują się jeszcze na swoich końcowych pozycjach na posortowanej liście, może się zmniejszać w miarę postępu wykonywania, ale nigdy nie jest mniejsza niż 0. Gdy liczba ta osiągnie 0, algorytm przypuszczalnie się kończy. Jak wybrać takie zbieżności i jak pokazano, że są zbieżne? Ponownie przykład pomoże odpowiedzieć na te pytania.

Odwracanie ciągu symboli: Przykład

Prawnym wejściem do poniższego problemu jest ciąg znaków S składający się z symboli, powiedz słowo lub tekst w języku angielskim. Celem jest wytworzenie odwróconego obrazu S, oznaczonego jako reverse(S), składającego się z symboli S w odwrotnej kolejności. Na przykład:

reverse(“ajj$dt8”) = “8td$jja”

Rysunek 1 przedstawia prosty schemat blokowy algorytmu A, który rozwiązuje problem.



Używa unikalnego pustego ciągu , który nie zawiera żadnych symboli (i który możemy oznaczyć pustymi cudzysłowami „”), oraz funkcji head(X) i tail(X), które dla dowolnego ciągu X oznaczają , odpowiednio, pierwszy symbol X i ciąg X z usuniętą głową. Mamy więc:

head(“ajj$dt8”) = “a”

i :

tail(“ajj$dt8”) = “jj$dt8”

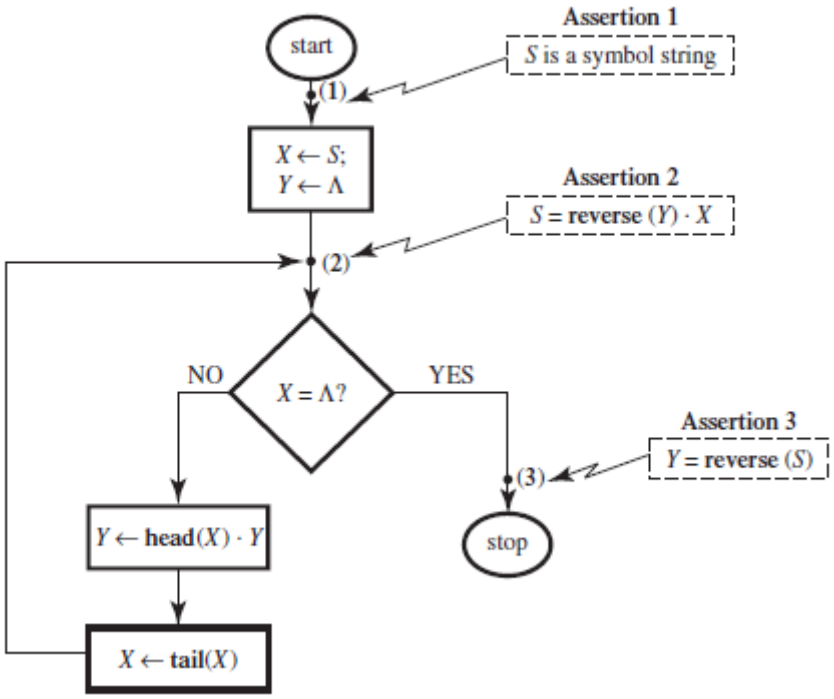
Używamy również specjalnego symbolu „·” dla konkatenacji ciągów lub załącznika. Zatem:

„ajj$dt8” · „tdd9tr” = „ajj$dt8tdd9tr”

W ten sposób będziesz w stanie łatwo zweryfikować, że przyczepienie głowy do ogona dowolnego sznurka daje sam sznurek. W symbolach:

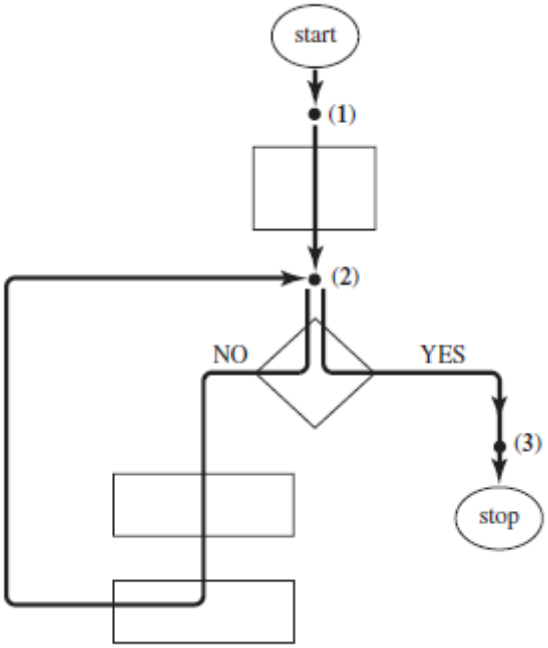
head(S) · tail(S) = S

Algorytm odwracania S polega na wielokrotnym „odklejaniu” symboli S jeden po drugim i dołączaniu każdego po kolei do początku nowo skonstruowanego ciągu Y . Nowy ciąg Y zaczyna się początkowo jako pusty. Procedura kończy się, gdy nie ma już nic do odklejenia S. Aby nie zniszczyć S w procesie, odklejanie odbywa się w zmiennej X, która jest inicjalizowana na wartość S. Twierdzi się, że ten algorytm poprawnie generuje odwrotność (S) w zmiennej Y . Oznacza to, że algorytm A z rysunku 1 jest całkowicie poprawny w odniesieniu do problemu algorytmicznego, który wymaga, aby wynik Y był odwróconym obrazem ciągu wejściowego S. Najpierw ustalimy, że A jest częściowo poprawny, używając metody asercji pośredniej , a następnie osobno, że również się kończy. Dlatego najpierw pokazujemy, że jeśli zdarzy się, że A zakończy się na łańcuchu wejściowym S, to wytworzy odwrotność(S) w Y . W tym celu rozważmy rysunek 2, na którym zidentyfikowano trzy punkty kontrolne.

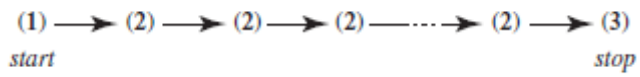


Jak już wyjaśniono, twierdzenie 1 ujmuje wymagania dotyczące zbioru danych wejściowych, a twierdzenie 3 zawiera pożądaną zależność między łańcuchem wejściowym S i wyjściem Y , a mianowicie, że Y ma być równe odwrotności (S). Jednak znaczenie rysunku 2 jest w asercji 2, która ma uchwycić sytuację tuż przed ponownym obejściem pętli lub zakończeniem. Twierdzenie 2 stwierdza, że w punkcie kontrolnym (2) bieżące wartości X i Y razem tworzą oryginalny łańcuch S, w tym sensie, że Y zawiera pewną początkową część S w odwrotnej kolejności, a X zawiera resztę S nieodwróconą, co jest dokładnie to samo, co powiedzenie, że konkatenacja reverse(Y) z X daje S. Chcemy pokazać, że wszystkie trzy twierdzenia są niezmiennikami, co oznacza, że w każdym wykonaniu A na dowolnych legalnych danych wejściowych są one prawdziwe, gdy tylko zostaną osiągnięte. Sztuczka polega na

rozważeniu wszystkich możliwych „przeskoków” od punktu kontrolnego do punktu kontrolnego, które procesor może wykonać podczas wykonywania algorytmu. W tym przypadku, biorąc pod uwagę formę tego konkretnego schematu blokowego, możliwe są trzy przeskoki: punkt (1) do punktu (2), punkt (2) do punktu (3) i punkt (2) z powrotem do punktu (2).



Pierwszy z nich przechodzi dokładnie raz w każdym wykonaniu A; drugi jest pokonywany co najwyżej raz, ponieważ na jego końcu algorytm się kończy; trzeci można przemierzyć wiele razy (ile?). Zauważ, że segmenty algorytmiczne odpowiadające tym ścieżkom są wolne od pętli. Składają się z prostych sekwencji elementarnych instrukcji i testów i nie zawierają iteracji. Jak zobaczymy, oznacza to, że można sobie z nimi dość łatwo poradzić. Musimy teraz pokazać, że dla każdego z tych prostych segmentów, jeśli założymy, że twierdzenie dołączone do jego punktu początkowego jest prawdziwe i że odcinek jest faktycznie przebyty, to twierdzenie dołączone do jego punktu końcowego również będzie prawdziwe, gdy zostanie osiągnięte. Twierdzi się, że to wystarczy, aby ustalić częściową poprawność, ponieważ nastąpi niezmienność wszystkich trzech twierdzeń. Dlaczego? Cóż, powód jest zakorzeniony w fakcie, że każde legalne wykonanie A składa się z sekwencji segmentów oddzielonych punktami kontrolnymi



W ten sposób, jeśli prawdziwość początkowego stwierdzenia każdego z tych segmentów implikuje prawdziwość końcowego twierdzenia i jeśli pierwsze twierdzenie całej sekwencji jest tym, które odpowiada legalnemu wejściu, a zatem zakłada się, że jest prawdziwe po pierwsze, prawdziwość twierdzeń rozprzestrzenia się w całej sekwencji, sprawiając, że wszystkie twierdzenia są prawdziwe w miarę postępu egzekucji. W szczególności, jak wyjaśniono, ostateczne Twierdzenie 3 będzie prawdziwe po rozwiązaniu, co stanowi częściową poprawność A. Podsumowując, musimy teraz pokazać, że prawdziwość twierdzeń rzeczywiście rozchodzi się naprzód po prostych ścieżkach między punktami kontrolnymi. Szczegóły tych dowodów nie zostaną tutaj przedstawione, ale zachęcamy do ich uzupełnienia. Pomaga jednak uważne zanotowanie, w symbolach, a nie słowach, tego, co dokładnie

ma być udowodnione. Patrząc na rysunki 2 i 3 i przypominając sobie trzy możliwe „przeskoki” między punktami kontrolnymi, okazuje się, że istnieją trzy stwierdzenia do udowodnienia:

(1 → 2): dla dowolnego ciągu S, po wykonaniu dwóch instrukcji X ← S; Y ← Λ,

zachowana zostanie równość S = reverse(Y) · X .

(2 → 3): if S = reverse(Y) · X, and X = Λ, then Y = reverse(S).

(2→2): if S = reverse(Y) · X, and X ≠ Λ, to po wykonaniu instrukcji

Y ←head(X) · Y ; X ←tail(X), ta sama równość, czyli S = reverse(Y) · X,

będzie obowiązywać dla nowych wartości X i Y . Formalne ustalenie, że te trzy stwierdzenia są prawdziwe, kończy dowód, że algorytm jest częściowo poprawny. Musimy teraz pokazać, że algorytm kończy się dla dowolnego ciągu wejściowego S. W tym celu ponownie rozważmy punkt kontrolny (2). Jedynym sposobem, w jaki wykonanie algorytmu może się nie zakończyć, jest nieskończenie częste przechodzenie przez punkt (2). Wykazano, że jest to niemożliwe przez wykazanie zbieżności (tj. wielkości zależnej od bieżących wartości zmiennych), która z jednej strony zmniejsza się za każdym razem, gdy punkt kontrolny (2) jest ponownie odwiedzany, ale z drugiej strony nie może stać się coraz mniejszy. Zbieżność, która działa w naszym przypadku, to po prostu długość łańcucha X. Za każdym razem, gdy pętla jest wykonywana, X jest skracane dokładnie o jeden symbol, ponieważ staje się on ogonem swojej poprzedniej wartości. Jednak jego długość nie może być mniejsza niż 0, ponieważ gdy X ma długość 0 (czyli staje się pustym ciągiem), pętla nie jest dalej przemierzana i algorytm się kończy. Na tym kończy się dowód, że algorytm odwrotny jest całkowicie poprawny. Może przyszło ci do głowy, że ten dowód nie jest wart zachodu, ponieważ poprawność programu wydaje się wystarczająco oczywista, a dowód wydaje się być żmudnie techniczny. Jest w tym trochę prawdy, a przykład został wybrany, aby zilustrować samą technikę dowodową, a nie potrzebę jej w tym konkretnym przykładzie. Niemniej jednak nietrudno wyobrazić sobie wersję tego samego algorytmu z jakimś subtelnym błędem w jednym z ekstremalnych przypadków lub wersję bardziej skomplikowaną, w której, powiedzmy, pozycje parzyste mają być odwrócone, a nieparzyste. nie. W takich przypadkach weryfikacja programu przez samo patrzenie jest niebezpiecznie nieodpowiednia a formalne dowody są koniecznością. Na marginesie, podczas weryfikacji poprawności nie musimy pracować na schematach blokowych. Chociaż wizualna natura schematu blokowego może czasami być pomocna w wyborze punktów kontrolnych i wnioskowaniu o dynamice algorytmu, istnieją proste sposoby dołączania pośrednich asercji do punktów w standardowych formatach tekstowych algorytmu. Nawet w tym małym przykładzie widać problematyczną naturę programowania imperatywnego. Wszystkie twierdzenia, które miały zostać udowodnione, zostały sformułowane w terminach różnych czasów, z rozróżnieniem między „nowymi” i „pierwotnymi” wartościami zmiennych. Napisanie tego algorytmu w funkcjonalnym języku programowania jest proste, a stwierdzenia do udowodnienia byłyby prostsze. Na przykład ostatni staje się:

(2 → 2): if S = reverse(Y) · X, and X ≠ Λ, then

S = reverse(head(X) · Y) · tail(X).

Nie rozwiązuje to zasadniczego problemu udowadniania poprawności, ale usuwa nieco kłopotliwy element jawnego zajmowania się zmianami wartości w czasie, co nie zawsze w naturalny sposób łączy się z naszymi matematycznymi oczekiwaniami dotyczącymi zmiennych.

Co zawiera dowód?

Dowód poprawności przedstawiony w ostatniej sekcji jest jednym z najprostszych w swoim rodzaju, nie będąc całkowicie trywialnym. Jest to co najmniej zniechęcające. Omówmy jego składniki. Podstawowym elementem zarówno częściowego dowodu poprawności, jak i zakończenia, jest wybór punktów kontrolnych w tekście algorytmu. Na ogół składają się one z punktów początkowych i końcowych oraz dostatecznie wielu lokalizacji pośrednich, aby każda pętla tekstu algorytmu zawierała przynajmniej jedną taką lokalizację. Po wybraniu punktów kontrolnych musimy dołączyć do nich asercje pośrednie, których niezmienność należy ustalić poprzez udowodnienie lokalnych warunków weryfikacji. Obejmują one tylko segmenty algorytmiczne wolne od pętli, ponieważ uważaliśmy, aby „otworzyć" wszystkie pętle z punktami kontrolnymi. Musimy również wykazać zbieżność i pokazać, że faktycznie jest zbieżna. Które z tych działań można zautomatyzować algorytmicznie? Innymi słowy, na co możemy oczekiwać znacznej pomocy od komputera? Znalezienie zestawu punktów kontrolnych do pokrycia każdej z pętli, nawet pewnego rodzaju zestawu minimalnego, może być w pełni zautomatyzowane. Co więcej, pod pewnymi warunkami technicznymi, wiele z lokalnych kontroli warunków weryfikacji, jak również lokalne zmniejszenie wartości zbieżności, może być również zautomatyzowane. Jednak sedno takich dowodów można znaleźć gdzie indziej. Polega na doborze odpowiednich niezmienników i zbieżności. Tutaj można wykazać, że nie istnieje żaden ogólny algorytm, który potrafiłby automatycznie znaleźć niezmienniki i zbieżności, które „działają", co oznacza, że spełniają lokalne warunki potrzebne do wytworzenia dowodu. Właściwy wybór niezmiennika jest sztuką delikatną i subtelną i może wymagać większej pomysłowości niż ta związana z zaprojektowaniem algorytmu. Rzeczywiście, projektant algorytmu może posiadać odpowiednią intuicję wymaganą do stworzenia dobrego algorytmu, ale może być zagubiony, gdy zostanie poproszony o precyzyjne sformułowanie „co się dzieje" w pewnym momencie. Paradoksalnie zawsze istnieją adekwatne niezmienniki i zbieżności, tak że, jak wspomniano wcześniej, poprawny algorytm można w zasadzie zawsze udowodnić. Wydaje się to przeczyć naszym twierdzeniom, że procesu weryfikacji nie można zautomatyzować, a projektant algorytmu może nie być w stanie udowodnić poprawności. To nie. A kluczem jest fraza „w zasadzie". Chociaż dowody istnieją, komputer nie zawsze może je znaleźć, a ludzie często też są zagubieni. W przypadku dużych i złożonych systemów oprogramowania weryfikacja po fakcie jest często niemożliwa, po prostu z powodu niewykonalnej wielkości i złożoności zadania. W przypadku przedstawienia dużego oprogramowania, które było nieustannie zmieniane, poprawiane, łatane i aktualizowane, dostarczanie formalnych i precyzyjnych twierdzeń, które całkowicie charakteryzują jego zachowanie w różnych punktach, jest zasadniczo wykluczone. W takich przypadkach należy zastosować alternatywną metodę, zwaną weryfikacją as-you-go, która zostanie omówiona później.

Wieże Hanoi: Przykład

Niezależnie od powyższej dyskusji, weryfikacja jest czasami nieco łatwiejsza niż oczekiwano. Mogłoby się wydawać, że subtelna natura rekurencji może utrudnić weryfikację podprogramów rekurencyjnych niż zwykłych algorytmów iteracyjnych. Nie zawsze tak jest. Przypomnij sobie problem Wież Hanoi i następujące rozwiązanie rekurencyjne, przedstawione w Części 2:

podprogram przesuń N z X na Y za pomocą Z:

(1) jeśli N wynosi 1, to wypisz „przenieś X do Y";

(2) w przeciwnym razie (to znaczy, jeśli N jest większe niż 1) wykonaj następujące czynności:

(2.1) wywołaj przeniesienie N − 1 z X do Z za pomocą Y ;

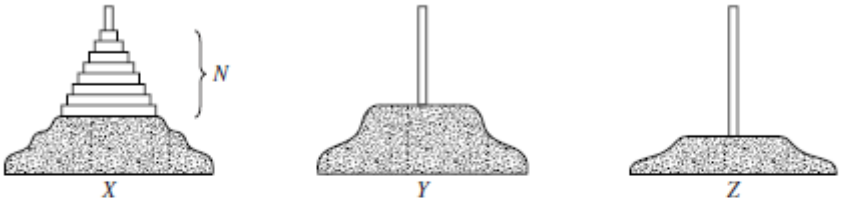
(2.2) wyjście „przesuń X do Y";

(2.3) wywołanie przesunięcia N − 1 z Z do Y przy użyciu X;

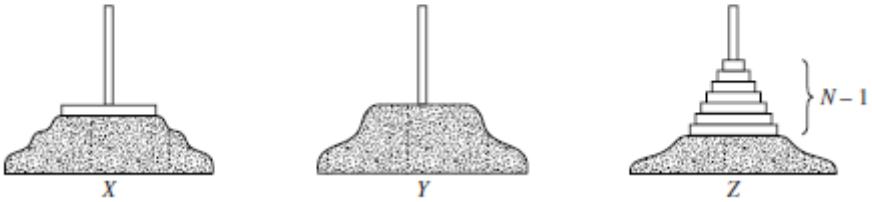
(3) powrót.

To, że ta procedura kończy się dla każdego N (gdzie wartości X, Y i Z to A, B i C w pewnej kolejności) wydaje się dość łatwe do zauważenia: jedyny możliwy rodzaj nieskończonego obliczenia jest uzyskiwany przez wykonanie nieskończonego głęboka kaskada wywołań rekurencyjnych. Jednak obserwując, że głębokość drzewa wywołań rekurencyjnych nie może być większa niż N, ponieważ jedyne, co dzieje się z N podczas wykonywania, to spadek o 1 za każdym razem, gdy wykonywane jest wywołanie rekurencyjne niższego poziomu, stwierdzamy, że N musi „ trafić" w pewnym momencie na wartość 1. Ale gdy N jest dokładnie 1, klauzula ucieczki rekurencji zostaje osiągnięta, powodując nie kolejne, głębsze wywołanie rekurencyjne, ale prostą instrukcję, po której następuje powrót, co pociąga za sobą wznoszenie drzewa. W konsekwencji drzewo wywołań rekurencyjnych jest skończone i wykonanie musi się zakończyć. Zauważ, że ten dowód zakończenia nie wymaga żadnego „zrozumienia" działania procedury; wykorzystaliśmy tylko powierzchowne obserwacje dotyczące zachowania N i obecności odpowiedniej klauzuli korekcyjnej. Jednak dowód nie jest do końca słuszny! Jeśli początkowa wartość N wynosi zero lub mniej, można ją zmniejszyć nieskończoną liczbę razy bez uderzania w 1, a procedura rzeczywiście nie zakończy się, jeśli otrzyma taką wartość dla N. To, co musimy tutaj zrobić, to dodać do ta procedura specyfikacja legalnych danych wejściowych, które wykluczają niedodatnie wartości N. (Co ciekawe, porównanie tego algorytmu z wersją podaną w PROLOGU w części 3 pokazuje, że wersja PROLOGa w rzeczywistości obsłuży przypadek N = 0 poprawnie.) Aby udowodnić częściową poprawność, używamy wariantu pośredniej metody asercji, który pasuje do nieiteracyjnego charakteru algorytmów rekurencyjnych. Zamiast próbować sformułować lokalną sytuację w danym punkcie, staramy się sformułować nasze oczekiwania dotyczące całej procedury rekurencyjnej tuż przed jej wejściem. Jest to następnie używane w cyklicznie wyglądającym, ale doskonale zdrowym stylu, aby się utrzymać! Oto możliwe sformułowanie rutyny ruchu. Dla dowolnego N prawdziwe jest następujące stwierdzenie, nazwij je (S):

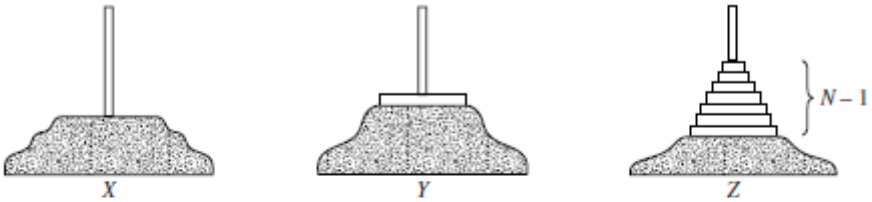
Załóżmy, że nazwy kołków A, B i C są skojarzone w pewnej kolejności ze zmiennymi X, Y i Z. Następnie kończące wykonanie wywołania przenosi N z X na Y za pomocą Z wymienia sekwencję pierścieni. instrukcje przemieszczania, które, jeśli zostaną uruchomione (i będą wiernie przestrzegane) w dowolnej legalnej konfiguracji pierścieni i kołków, w której co najmniej N najmniejszych pierścieni jest na kołku X, prawidłowo przenosi te N pierścieni z X na Y, prawdopodobnie używając Z jako tymczasowego przechowywania . Co więcej, sekwencja jest zgodna z zasadami problemu Wież Hanoi i pozostawia nietknięte wszystkie inne pierścienie. Jeśli teraz możemy pokazać, że (S) jest prawdziwe dla wszystkich N, ustalimy częściową poprawność naszego rozwiązania, ponieważ szczególnie interesuje nas wywołanie przeniesienia N z A do B za pomocą C, gdzie A zawiera N pierścieni i B i C są puste. W tym przypadku (S) jest tylko przeformułowaniem wymagań problemu, co możesz zweryfikować. Zdanie (S) można ustalić klasyczną metodą indukcji matematycznej. Oznacza to, że najpierw pokazujemy bezpośrednio, że zdanie jest prawdziwe, gdy N wynosi 1, a następnie pokazujemy, że przy założeniu, że jest prawdziwe dla pewnego danego N − 1 (gdzie N − 1 wynosi co najmniej 1, więc N wynosi co najmniej 2), musi to być również prawdziwe dla samego N. Wynika z tego, że (S) musi być prawdziwe dla wszystkich N, ponieważ oddzielnie udowodniona prawda dla przypadku N = 1 implikuje prawdę dla N = 2, co z kolei implikuje prawdę dla N = 3 itd., ad nieskończoność. Przyjrzyjmy się teraz uważnie szczegółom dowodu.



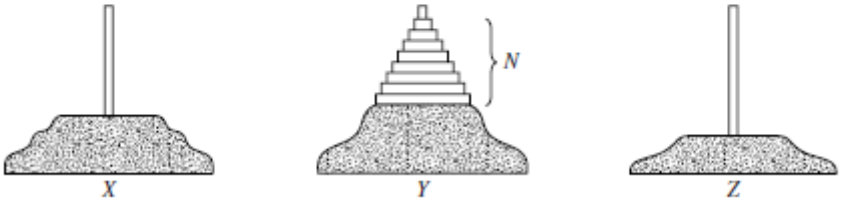
(a) **Initial configuration:** X contains (at least) N smallest rings; others are scattered.



(b) **Inductive hypothesis:** $N-1$ top rings on X are correctly moved to Z.



(c) **Direct move:** 1 ring on X is moved directly to Y.



(d) **Inductive hypothesis:** $N-1$ top rings on Z are correctly moved to Y.

To (S) obowiązuje, gdy N wynosi 1, jest trywialne: stwierdzenie (S) w tym przypadku jest po prostu stwierdzeniem, że kiedy podprogram zostanie wywołany, gdy N wynosi 1, tworzona jest sekwencja, która prawidłowo przenosi najwyższy pierścień z X na Y . Od razu widać, że zostało to osiągnięte przez pierwszą linię podprogramu. Załóżmy teraz, że zdanie (S) obowiązuje dla pewnego arbitralnego N − 1. Teraz musimy pokazać, że dotyczy ono również dla N. W związku z tym załóżmy, że wywołanie procedury ruchu zostało właśnie wykonane z liczbą N i pewnym powiązanie kołków A, B i C ze zmiennymi X, Y i Z. Trzy kołki zawierają pewien legalny układ pierścieni, przy czym kołek X zawiera co najmniej N najmniejszych pierścieni (patrz (a)). Teraz, ponieważ wartość N nie wynosi 1, pierwszą rzeczą, jaką robi podprogram, jest wywołanie siebie rekurencyjnie w linii (2.1) z parametrem N − 1. Zgodnie z hipotezą indukcyjną, to znaczy zakładając, że (S) obowiązuje dla wywołania do procedury z N − 1, to wywołanie, jeśli zostanie zakończone pomyślnie, poprawnie i legalnie przesunie najwyższe N − 1 dzwonków z X na Z, pozostawiając wszystko inne bez zmian (patrz (b)). Jednakże, ponieważ X miał gwarancję, że na początku będzie zawierało co najmniej N dzwonków, po zakończeniu połączenia on-line (2.1) na X nadal pozostaje co najmniej jeden dzwonek, a instrukcja online (2.2) przenosi ten dzwonek bezpośrednio do Y (patrz (c)). Ponieważ wszystkie pierścienie na Y przed tym ruchem (jeśli były) musiały być większe niż N-ty poruszany pierścień, ten ruch jest prawidłowy. To samo założenie dotyczące N − 1, ale teraz zastosowane do następnego wywołania, w linii (2.3) podprogramu, można

zobaczyć, aby uzupełnić obraz, przesuwając pierścienie N − 1 z Z do Y, tuż nad pojedynczym pierścień, który został przeniesiony oddzielnie (patrz (d)). Powinieneś dokładnie przeanalizować ten dowód, zauważając, dlaczego legalność wszystkich ruchów jest również gwarantowana przez cały proces, a nie tylko ostateczny wynik. Możliwe jest również wykazanie (używając tej samej hipotezy indukcyjnej dla dwóch wywołań), że żadne dzwonki inne niż górne N na kołku X nie są dotykane tą procedurą, tak że (S) zostało ustalone w całości dla tego wywołania z N, zakładając to dla wywołań z N − 1. Jak już wspomniano, ustala to, że zdanie (S) obowiązuje dla wszystkich N. Częściowa poprawność rekurencyjnego algorytmu Wież Hanoi jest w ten sposób udowodniona, a ponieważ już udowodniliśmy zakończenie, algorytm jest całkowicie poprawny.

Więcej o wieżach Hanoi: proste rozwiązanie iteracyjne

W Części 2 obiecaliśmy zaprezentować niezwykle prosty do wykonania algorytm iteracyjny dla problemu Wieże Hanoi. Powód omawiania tego tutaj, a nie w części 2, wynika z faktu, że nie jest do końca jasne, dlaczego to działa, a fakt, że to działa, wymaga dowodu. Rzeczywiście, nie będziemy tutaj przedstawiać dowodu jego poprawności, a zachęcamy do próby skonstruowania dowodu, pokazując, że algorytmy iteracyjne i rekurencyjne są naprawdę równoważne. Można to zrobić przez indukcję na N. Aby opisać algorytm, załóżmy, że trzy kołki są ułożone w okrąg i że wszystkie pierścienie N są ułożone na jednym z nich (nazwy kołków są nieistotne). Oto algorytm:

(1) wykonaj następujące czynności wielokrotnie, aż przed krokiem (1.2) wszystkie pierścienie zostaną prawidłowo ułożone na innym kołku:

(1.1) przesuń najmniejszy pierścień z obecnego kołka na następny kołek w kolejności zgodnej z ruchem wskazówek zegara;

(1.2) sprawiają, że jedyny możliwy ruch, który nie obejmuje najmniejszego pierścienia.

Powinno być jasne, że krok (1.2) jest dobrze zdefiniowany i jednoznaczny, ponieważ jeden z kołków musi mieć najmniejszy pierścień na górze, a z dwóch pozostałych kołków jeden ma mniejszy pierścień na górze niż drugi; stąd jedynym ruchem, który nie obejmuje najmniejszego pierścienia, jest przeniesienie tego mniejszego pierścienia na drugi kołek. Ten algorytm może być łatwo wykonany przez małe dziecko, nawet jeśli w grę wchodzi wiele dzwonków. Zauważ mimochodem, że jeśli N jest nieparzyste, pierścienie wylądują na następnym kołku w kolejności zgodnej z ruchem wskazówek zegara, a jeśli N jest parzyste, trafią na następny kołek w kolejności przeciwnej do ruchu wskazówek zegara.

Weryfikacja po fakcie a weryfikacja na bieżąco

Alternatywą dla udowodnienia poprawności już ukończonego algorytmu jest wspólne opracowanie algorytmu i dowodu. Chodzi o to, aby dopasować zdyscyplinowaną, stopniową, krok po kroku konstrukcję algorytmu lub systemu oprogramowania do stopniowej konstrukcji segmentów dowodu, które ostatecznie skumulują się, tworząc kompletny dowód całego systemu. Najwyraźniej łatwiej to powiedzieć niż zrobić. Jednak oczywistym mechanizmem, który zachęca do takiej praktyki, jest podprogram. Jak wyjaśniono w części 2, złożony program można starannie zbudować z wielu procedur, z których niektóre są zagnieżdżone w innych, co skutkuje dobrze ustrukturyzowanym i rozwarstwionym oprogramowaniem. Dobra praktyka projektowa nakazuje, aby każda procedura była dokładnie analizowana pod kątem jej ogólnego przeznaczenia, a następnie weryfikowana jako oddzielna całość. W zasadzie po wykonaniu tej czynności cały system również zostanie zweryfikowany. Powodem jest to, że to, co zostało udowodnione o danej procedurze, może być użyte do udowodnienia rzeczy o innych procedurach, które to nazywają. Jeśli, na przykład, zweryfikujemy, że procedura wyszukiwania

z Części 2 poprawnie odnajduje X w tekście wejściowym i wyprowadza licznik, jeśli dojdzie do końca tekstu, wówczas ten dowód może zostać użyty do zweryfikowania znajdowania „pieniędzy". algorytm. Tutaj również istnieje wiele możliwych pułapek. Nie jest łatwo poprawnie zidentyfikować interfejs procedury i jej zamierzone zachowanie w każdych okolicznościach, ale dobry projekt algorytmiczny zaleca, abyśmy nie tworzyli podprogramu, chyba że możemy to zrobić. Oczywiście procedura rekurencyjna wymaga cyklicznego procesu weryfikacji, który obejmuje założenie, że jest poprawny dla własnych wywołań, tak jak zrobiono to w przypadku Wież Hanoi. Oczywiście ten proces nie zawsze jest tak prosty, jak się wydaje. Krótko mówiąc, dobry krokowy projekt algorytmiczny, w połączeniu z informacyjną i precyzyjną dokumentacją decyzji projektowych, może stanowić podstawę konstrukcji zweryfikowanego oprogramowania, poprzez rozbicie ogólnego dowodu na części, które odzwierciedlają rozpad samego oprogramowania. Kiedy współbieżność i reaktywność są wprowadzane do algorytmów, jak to ma miejsce w częściach 10 i 14, sprawy stają się trudniejsze, a nawet mały i niewinnie wyglądający algorytm może powodować ogromne problemy, jeśli chodzi o weryfikację. Porozmawiamy o tym moście, ale nie do końca go przekroczymy, kiedy do niego dotrzemy.

Projekt według umowy

Paradygmat zorientowany obiektowo przenosi ideę bieżącej weryfikacji o krok naprzód, w formie metodologii zwanej projektowaniem na podstawie umowy. Jak wspomniano w części 3, każda klasa w programie zorientowanym obiektowo opisuje pewien abstrakcyjny zestaw obiektów z powiązanymi z nimi zachowaniami lub metodami. Taki opis może zawierać instrukcje, jak przeprowadzić każdą metodę, ale nie musi. W rzeczywistości często bardzo przydatne jest posiadanie całkowicie abstrakcyjnych klas, które określają tylko, co należy zrobić, ale nie jak. Wcielenie JAVA klasy Queue jest takim przykładem, w którym nie określa zachowania każdej metody, a jedynie sposób jej wywołania. Co więcej, zastępując słowo „Kolejka" słowem „Stos" otrzymalibyśmy całkowicie legalny opis stosów zamiast kolejek. Brakuje znaczenia każdej metody: jak wpływa ona na stan obiektu i co (jeśli w ogóle) zwraca. W podejściu projektowania po kontrakcie specyfikacja ta nazywana jest kontraktem i składa się z trzech rodzajów stwierdzeń: niezmienników klasy, warunków wstępnych metody i warunków końcowych metody. Niezmienniki klasy określają warunki, które muszą być spełnione dla każdego obiektu klasy przed i po wykonaniu każdej operacji. Warunki wstępne metody określają legalne dane wejściowe do każdej metody (lub innymi słowy, co musi być przechowywane przed wykonaniem metody), a warunki końcowe metody określają, co musi być przechowywane po wykonaniu metody. W kontekście poprzednich dyskusji w tej części, warunki wstępne metody i warunki końcowe są tak naprawdę tylko twierdzeniami na początku i na końcu procedur, a niezmienniki klasy są analogiczne do niezmienników pętli. Na przykład w klasie Queue warunkiem wstępnym dla metody front byłoby to, że kolejka nie jest pusta. Warunkiem końcowym add(X) jest to, że front zwróci X, jeśli kolejka była pusta przed wywołaniem add, w przeciwnym razie zwróci tę samą wartość, którą zwróciłby przed tą operacją. Natomiast warunek końcowy dla odpowiedniej metody w klasie Stack powiedziałby, że po wywołaniu metody add(X), front zwróci X, niezależnie od poprzedniego stanu obiektu. Pozwala to uchwycić istotną różnicę między stosami a kolejkami. Metodologia projektowania po kontrakcie wymaga sprecyzowania kontraktu przed napisaniem programu wdrożeniowego oraz starannego modyfikowania kontraktu w miarę zmian specyfikacji. Stąd już tylko krok do weryfikacji programu, a pojawiają się narzędzia, które mogą w tym pomóc. Projektowanie według umowy jest integralną cechą języka programowania EIFFEL, który obsługuje również niezmienniki pętli i konwergencję. Chociaż nie jest to jeszcze część oficjalnego języka JAVA, istnieje wiele narzędzi, które dodają tę możliwość również do JAVA. Pozwalają one kompilatorowi generować kod, który faktycznie sprawdza asercje podczas działania programu, ostrzegając programistę o napotkanych naruszeniach. Jednak chociaż takie narzędzia są przydatne do testowania programów, głównym wkładem metody projektowania według umowy jest zwracanie uwagi na poprawność programu podczas jego pisania i modyfikowania. W

związku z tym może być praktykowane nawet przez programistów używających języków obiektowych, które nie mają jeszcze obsługi narzędzi do sprawdzania asercji w czasie wykonywania. Określanie zachowania funkcji lub podprogramów za pomocą warunków wstępnych i warunków końcowych jest możliwe w każdym języku programowania. Zasadniczo istnieją dwa powody, dla których projektowanie na podstawie umowy jest szczególnie odpowiednie dla paradygmatu zorientowanego obiektowo. Po pierwsze, podział programu na klasy i metody oraz możliwość klas abstrakcyjnych dostarcza naturalnych punktów do umieszczania asercji. Po drugie, styl zorientowany obiektowo daje realną obietnicę możliwości ponownego wykorzystania kodu, co oznacza możliwość używania tej samej klasy – kodu implementującego i wszystkiego – w wielu różnych aplikacjach. Jednak, aby to zadziałało, niezbędne jest jasne określenie, co mają oznaczać klasy i metody. Bez tego programista próbujący ponownie wykorzystać czyjąś klasę może popełniać takie błędy, jak mylenie kolejki ze stosem. Spektakularnym tego przykładem jest utrata pierwszego startu Ariane 5 w 1996 roku. Awarię modułu nawigacyjnego przypisuje się praktyce ponownego wykorzystywania programów z Ariane 4 bez zrozumienia założeń, na których opierały się oryginalne programy; te założenia nie były już prawdziwe dla Ariane 5, ale nie zostały udokumentowane w kodzie.

Interaktywna weryfikacja i kontrola dowodowa

Pomimo tego, że generalnie automatyczna weryfikacja algorytmiczna nie wchodzi w rachubę, przy pomocy komputera można wiele zrobić. W tym miejscu przeanalizujemy tylko kilka kwestii, ponieważ większość z nich jest zbyt techniczna, aby można ją było tutaj szczegółowo omawiać. Komputer można zaprogramować do pomocy w weryfikacji po fakcie. Możliwy scenariusz to interaktywny program, który próbuje zweryfikować dany algorytm pod kątem formalnego opisu problemu algorytmicznego. Od czasu do czasu prosi użytkownika, aby dostarczył mu, powiedzmy, niezmiennik kandydata (przypomnij sobie, że jest to zadanie, którego na ogół nie może wykonać samodzielnie). Następnie kontynuuje i próbuje ustalić odpowiednie warunki weryfikacji, a następnie przechodzi do dodatkowych punktów w algorytmie, jeśli się powiedzie, lub cofa do poprzednich decyzji, jeśli się nie powiedzie. Ponieważ cała procedura jest interaktywna, użytkownik może ją zatrzymać, gdy wydaje się, że idzie w złym kierunku, i spróbować skierować ją z powrotem na właściwe tory. Komputer jest tutaj używany jako szybki i skrupulatny praktykant, sprawdzający szczegóły lokalnych warunków logicznych, śledzący przeszłe niezmienniki, twierdzenia i komentarze, i nigdy niestrudzony w wypróbowywaniu innej możliwości. Taki proces może również prowadzić do odkrycia ukrytych błędów lub brakujących założeń. Na przykład skomputeryzowany weryfikator nie zaakceptuje podanego wcześniej błędnego dowodu zakończenia programu Wieże Hanoi. Zbadanie przyczyny odrzucenia dowodu doprowadziłoby programistę do uświadomienia sobie, że brakowało odpowiedniego warunku wstępnego. Ten pomysł może być również wykorzystany do pomocy w działaniach weryfikacyjnych na bieżąco. Tutaj, wcześniej ustalone dowody części algorytmu (powiedzmy, podprogramów) mogą być śledzone przez automatycznego ucznia i użyte w algorytmicznych próbach weryfikacji większych porcji (powiedzmy, wywoływanie podprogramów). Taki interaktywny system może być częścią całego środowiska programistycznego, które może obejmować również narzędzia do edycji, debugowania i testowania. Użytkownik może być zainteresowany weryfikacją niektórych mniejszych i lepiej zidentyfikowanych algorytmów w opracowywanym systemie oprogramowania i pozostawić mniej łatwe w zarządzaniu części do konwencjonalnego debugowania i testowania. Inną możliwością weryfikacji wspomaganej komputerowo jest sprawdzanie dowodu. Tutaj człowiek wytwarza coś, co wydaje się dowodem – niezmienniki, zbieżności i wszystko – a komputer jest zaprogramowany do generowania i weryfikacji długich sekwencji logicznych i symbolicznych manipulacji, które składają się na pełnoprawny dowód. Temu tematowi poświęcono i nadal poświęcono wiele pracy, istnieje kilka systemów, które mogą pomóc w projektowaniu i weryfikacji nietrywialnych programów. Jak omawiamy w częściach 10 i 14, potężne metody (wiele z nich opiera się na technice zwanej sprawdzaniem modelu) zostały

opracowane do weryfikacji nawet złożonych systemów oprogramowania i sprzętu. Przy pewnych rozsądnych założeniach, takich jak skończona liczba stanów zachowania systemu i proste, dobrze zdefiniowane właściwości do weryfikacji, działają one całkiem dobrze w praktyce. Są więc zachęcające znaki. Wciąż jednak pozostaje wiele problemów, a stosowanie sprawnych i szeroko stosowanych pomocy weryfikacyjnych jeszcze trochę potrwa, zanim stanie się powszechną praktyką.

Badania poprawności algorytmicznej

Oprócz tematów omówionych w poprzednim częściach, badacze są zainteresowani opracowaniem nowych i użytecznych metod weryfikacji, a metody niezmienne i zbieżne są tylko przykładami najbardziej podstawowych z nich. Różne konstrukcje językowe uruchamiają różne metody, co stanie się widoczne po wprowadzeniu współbieżności i probabilizmu w Częściach 10 i 11. Jedną z kwestii, której w ogóle nie zajęliśmy się, jest kwestia języków specyfikacji, czasami zwane językami asercyjnymi. Jak formalnie określamy problem algorytmiczny? Jak zapisać twierdzenia pośrednie, aby były jednoznaczne i poddawały się manipulacji algorytmicznej? Alternatywy różnią się nie tylko pragmatyzmem, ale także podstawową matematyką. W przypadku niektórych języków asercji ustalenie lokalnych warunków weryfikacji segmentów pozbawionych pętli między punktami kontrolnymi jest algorytmicznie wykonalne, podczas gdy w przypadku innych można wykazać, że problem jest tak trudny, jak sam globalny problem weryfikacyjny, a zatem nie można go rozwiązać algorytmicznie. W obu przypadkach ludzie są zainteresowani rozwojem technik dowodzenia twierdzeń, które stanowią podstawę algorytmów sprawdzania warunków. Wiele tematów badawczych dotyczy zarówno weryfikacji i debugowania, jak i wydajnej i pouczającej kompilacji. Dobrym przykładem jest analiza przepływu danych, w ramach której opracowywane są metody symbolicznego śledzenia przepływu danych w algorytmie. Oznacza to, że możliwe zmiany, jakie zmienna może przejść podczas wykonywania, są analizowane bez faktycznego uruchamiania algorytmu. Taka analiza może ujawnić potencjalne błędy, takie jak indeksy wykraczające poza granice tablic, możliwe wartości zerowe dla dzielników w operacjach arytmetycznych i tak dalej. Badaczy interesują także bardziej złożone właściwości algorytmów. Czasami może być ważne, aby wiedzieć, czy dwa algorytmy są równoważne. Jednym z powodów może być dostępność prostego, ale nieefektywnego algorytmu oraz kandydata projektanta na bardziej wydajny, ale bardziej skomplikowany algorytm. Pragniemy tutaj udowodnić, że oba algorytmy zakończą się na tej samej klasie danych wejściowych i dadzą identyczne wyniki. Jednym z podejść do badania tak bogatszych klas własności algorytmicznych jest formułowanie logiki programów lub logik algorytmicznych, które są analogiczne do klasycznych systemów logiki matematycznej, ale które umożliwiają wnioskowanie o algorytmach i ich skutkach. Podczas gdy systemy klasyczne są czasami nazywane statycznymi, logiki algorytmiczne mają charakter dynamiczny; prawdziwość stwierdzeń zawartych w tych formalizmach zależy nie tylko od obecnego stanu świata (jak w stwierdzeniu „pada deszcz”), ale od relacji między stanem obecnym a innymi możliwymi. Jako przykład rozważ podstawową konstrukcję takiej dynamicznej logiki:

after (A, F)

co oznacza, że po wykonaniu algorytmu A, twierdzenie F z konieczności będzie prawdziwe. Twierdzenie F może stwierdzać, że lista L jest posortowana, w którym to przypadku instrukcja może być użyta do sformalizowania częściowej poprawności procedury sortującej. Siła takiego konstruktu tkwi jednak w możliwości wielokrotnego użycia go w wypowiedzi i połączenia z innymi obiektami logicznymi. Rozważmy następujące, gdzie „→” oznacza „implikacje”:

jeśli F1 → after (A, F2) i F2 → after (B, F3)

wtedy F1 → after (A; B, F3)

Stwierdzenie to stwierdza, że jeśli (1) zawsze, gdy F1 jest prawdziwe, F2 jest prawdziwe po wykonaniu A, oraz jeśli także (2) zawsze, gdy F2 jest prawdziwe, F3 jest prawdziwe po wykonaniu B, to możemy wywnioskować (3) zawsze, gdy F1 jest prawdziwe, F3 jest prawdziwe po wykonaniu A; B (czyli A, po którym następuje bezpośrednio B). To stwierdzenie może wydawać się oczywiste, ale jest dość subtelne. W rzeczywistości stanowi matematyczne uzasadnienie wstawiania twierdzeń pośrednich do algorytmu. F2 można traktować jako twierdzenie pośrednie dołączone do punktu oddzielającego A od B, a stwierdzenie to mówi, że ustalenie warunków lokalnych na A i B oddzielnie sprowadza się do ustalenia bardziej globalnego warunku na A; B. W takiej logice można sformułować (i udowodnić) bardziej skomplikowane stwierdzenia dotyczące równoważności i zakończenia programów oraz innych interesujących właściwości. Logika programów pomaga zatem oprzeć teorię weryfikacji algorytmicznej na solidnych podstawach matematycznych. Ponadto pozwalają nam badać pytania dotyczące możliwości zautomatyzowania metod sprawdzania terminacji, poprawności, równoważności i tym podobnych. Jedna z najskuteczniejszych metod weryfikacji, zwana modelem sprawdzania może być użyta do sprawdzenia programu pod kątem formuły logicznej, ale tutaj również istnieją nieodłączne ograniczenia takiego wysiłku, które zostaną omówione w Częściach 7 i 8. Ponadto Część 10 omawia weryfikację systemów współbieżnych, a Część 12 omawia ogólne dodawanie interakcji do dowodów matematycznych, a także dowody, które są zarówno interaktywne, jak i probabilistycznie sprawdzalne. Części 13 i 14, które omawiają duże i złożone systemy, również dotykają niektórych kwestii poprawności. Inny obiecujący kierunek badań dotyczy automatycznej lub półautomatycznej syntezy algorytmicznej. W tym przypadku przedmiotem zainteresowania jest synteza działającego algorytmu lub programu na podstawie specyfikacji problemu algorytmicznego. Podobnie jak w przypadku weryfikacji, ogólny problem syntezy jest algorytmicznie nierozwiązywalny, ale części procesu mogą być wspomagane przez komputer. Jednym z problemów związanych zarówno z syntezą, jak i dowodami równoważności jest kwestia transformacji programu, w której części algorytmu lub programu są przekształcane w sposób zapewniający zachowanie równoważności. Na przykład możemy być zainteresowani takimi przekształceniami, aby algorytm był zgodny z zasadami jakiegoś języka programowania (na przykład zastąpienie rekurencji przez iterację, gdy język nie pozwala na rekurencję) lub w celach wydajnościowych, jak pokazano w Części 6. Badania w zakresie poprawności i logiki algorytmów ładnie komponują się z badaniami nad semantyką języków programowania. Nie da się udowodnić czegokolwiek na temat programu bez rygorystycznego i jednoznacznego znaczenia dla tego programu. Im bardziej złożone są używane przez nas języki, tym trudniej jest zapewnić im semantykę i tym trudniej wymyślić metody dowodowe i zbadać je za pomocą logik algorytmicznych. Jest to kolejny powód, dla którego języki funkcjonalne są bardziej podatne na dowody poprawności niż języki imperatywne.

Twierdzenie o czterech kolorach

Wydaje się właściwe zamknięcie tej części opowieścią o pewnym filozoficznym znaczeniu. Problem czterokolorowy został sformułowany w 1852 roku i przez około 120 lat uważany był za jeden z najciekawszych otwartych problemów w całej matematyce. Obejmuje mapy, takie, jakie można znaleźć w atlasie: diagramy składające się z podziałów skończonej części płaszczyzny przez zamknięte regiony oznaczające kraje. Załóżmy, że chcemy pokolorować taką mapę, przypisując kolor każdemu krajowi, ale w taki sposób, aby żadne dwa kraje, które dzielą część granicy, nie były pokolorowane tym samym kolorem. Ile kolorów jest potrzebnych do pokolorowania dowolnej mapy? Na pierwszy rzut oka wydaje się, że możemy konstruować coraz bardziej skomplikowane mapy, wymagające coraz większej liczby kolorów. Kiedy jednak pobawimy się kilkoma przykładami, okazuje się, że zawsze wystarczą cztery kolory. Problem czterech kolorów pyta, czy to zawsze prawda. Z jednej strony nikt nie był w stanie udowodnić, że cztery kolory były wystarczające do pokolorowania dowolnej mapy, z drugiej strony nikt nie był w stanie wykazać mapy wymagającej pięciu. Przez lata nad tym problemem pracowało wiele

osób, a w dziedzinie matematyki zwanej topologią pojawiło się wiele głębokich i pięknych wyników. Opublikowano kilka „dowodów", że cztery kolory wystarczą, ale później okazało się, że zawierają one subtelne błędy. W 1976 roku problem został ostatecznie rozwiązany przez dwóch matematyków. Udowodnili to, co jest obecnie znane jako twierdzenie o czterech kolorach, które twierdzi, że cztery kolory rzeczywiście wystarczą. Co to ma wspólnego z nami? Cóż, dowód z 1976 roku został osiągnięty przy pomocy komputera. Dowód można uznać za składający się z grubsza z dwóch części. W pierwszym, dwaj badacze wykorzystali kilka wcześniej ustalonych wyników, z pewnym dodatkowym rozumowaniem matematycznym, aby udowodnić, że ogólny problem można sprowadzić do wykazania, że skończoną liczbę szczególnych przypadków można pokolorować czterema kolorami. Napisano wówczas programy komputerowe, które miały skrupulatnie generować wszystkie takie przypadki (okazało się, że jest ich około 1700) i przejrzeć je wszystkie w celu znalezienia czterokolorowości. Twierdzenie zostało ustalone, gdy programy się zakończyły, odpowiadając pozytywnie na pytanie: wszystkie 1700 przypadków okazało się czterokolorowych. Czy możemy umieścić tradycyjne Q.E.D. (oznaczające quod erat demonstrandum, swobodnie tłumaczone jako „to, co miało być udowodnione") na końcu dowodu? Problem w tym, że nikt nigdy nie zweryfikował programów użytych w dowodzie. Możliwe, że algorytmy skonstruowane w celu przeprowadzenia subtelnego generowania przypadków były wadliwe. Mało tego, nikt jeszcze nigdy nie zweryfikował poprawności kompilatora użytego do tłumaczenia programów na kod wykonywalny maszynowo. Nikt nie zweryfikował żadnego z innych odpowiednich programów systemowych, które mogłyby wpłynąć na prawidłowe działanie programów, a z tego powodu (choć nie jest to nasze zmartwienie) nikt nie sprawdził, czy sprzęt działa tak, jak powinien. W rzeczywistości społeczność matematyczna zaakceptowała twierdzenie. Być może ma to związek z faktem, że od 1976 r. pojawiło się wiele dodatkowych dowodów, z których wszystkie wykorzystują komputer, ale niektóre z nich muszą sprawdzać tylko mniejszy i łatwiejszy w zarządzaniu zestaw przypadków oraz używać krótszych i jaśniejszych programów. Filozoficzne pozostaje jednak pytanie: czy absolutna prawda matematyczna, w przeciwieństwie do praktycznych aplikacji komputerowych, będzie mogła zależeć od nieco wątpliwej wydajności niezweryfikowanego oprogramowania?