Uniwersytet Warszawski

Wydział Matematyki, Informatyki i Mechaniki

Specyfikacja wskaźnikowych struktur danych

(Praca doktorska)

Marcin Kubica

Promotor: prof. dr hab. Jan Madey

16 stycznia 2000

Spis treści

Rozdział 1

Wstęp

Obecnie, gdy technika komputerowa ogarnia coraz większe sfery naszego życia, coraz większego znaczenia nabiera jakość wytwarzanego oprogramowania. Główną cechą decydującą o jakości oprogramowania jest jego niezawodność. Gorsza funkcjonalność oprogramowania może oznaczać większy nakład pracy użytkownika. Mniejszą efektywność można próbować kompensować większymi zasobami sprzętowymi lub dłuższym czasem pracy. Jednak błędy w oprogramowaniu potrafią przynieść trudne do przewidzenia straty i to w najmniej spodziewanym miejscu i chwili — nie tylko tam, gdzie następstwa takiego błędu mogą być groźne dla życia ludzkiego (ang. *safety critical applications*), ale i w zwykłym biurze. Dlatego też błędy w oprogramowaniu stawiają pod znakiem zapytania jego wartość.

Niezawodność oprogramowania ma również istotne znaczenie dla kosztów jego wytwarzania. W przypadku dużych systemów programistycznych koszty usuwania błędów przekraczają wielokrotnie koszty przygotowania jego pierwszej implementacji [Mye80]. W takiej sytuacji zwiększenie niezawodności oprogramowania może przynieść dużo większe oszczędności niż np. zwiększanie efektywności zespołu programistycznego.

Drogą do uzyskania niezawodnego oprogramowania jest zapewnienie jego poprawności na wszystkich etapach wytwarzania. W sytuacji, gdy stopień komplikacji dużych programów jest niemożliwy do ogarnięcia przez jednego człowieka, a zespoły programistyczne liczą wiele osób, nie ma miejsca na domysły co do poprawności poszczególnych zadań projektowych i programistycznych. Każda część programu powinna zostać zweryfikowana. Aby mówić o poprawności programu, musimy mieć specyfikację określającą wymagania, które ów program ma spełniać. Jeżeli zarówno specyfikacja jak i program są formalnie określone, to mamy możliwość formalnego zweryfikowania poprawności jednego względem drugiego. Stąd też wynika istotna rola formalnych specyfikacji w wytwórstwie niezawodnego oprogramowania.

Gdy porównamy stosowane techniki programistyczne oraz znane metody specyfikacji, możemy zauważyć pewną rozbieżność. Efektywne algorytmy [BDR96, BKR87, CLR97, Lip89, Wir99] bardzo często wykorzystują wskaźnikowe struktury danych. Z drugiej strony znane metody specyfikacji programów (np. Larch [GH93], Z [Spi92, Wor92], VDM [Jon84], RAISE [GHH$^+$92, GHH$^+$95], logika Hoare'a [Apt81, Hoa69, Dah92], transformatory predykatów Dijkstry [Dij85], czy logika algorytmiczna [MS87, MS92]) pomijają kwestię wskaźników lub traktują ją w sposób niewystarczający. Znane są próby opisywania wskaźników w ramach logiki Hoare'a (np. [CO81, JvEB77]), jednak pozwalają one jedynie na wyrażanie własności w logice pierwszego rzędu. Stąd też ich zastosowanie jest ograniczone, gdyż logika pierwszego rzędu nie pozwala na wyrażenie takich podstawowych własności jak spójność, istnienie ścieżki

o zadanych końcach, czy istnienie cyklu (np. [Cou90, Tho97]). Na uwagę zasługuje wariant logiki Hoare'a zaproponowany w pracy [BS95], gdzie język asercji jest wariantem logiki drugiego rzędu. Formalizm ten jest jednak również nastawiony na weryfikowanie własności wyrażonych w logice pierwszego rzędu. Zapisane w nim specyfikacje powszechnie stosowanych struktur danych są nieczytelne i nienaturalne.

Zastosowanie logiki algorytmicznej daje tu większe możliwości. Możemy ją wykorzystać dwojako. Po pierwsze, logiki algorytmicznej można użyć jako narzędzia do weryfikacji programów względem asercji nie zawierających konstrukcji algorytmicznych. Wówczas jednak, podobnie jak w przypadku logiki Hoare'a, nie jesteśmy w stanie wyrazić za pomocą asercji opartych o *FOL* podstawowych własności wskaźnikowych struktur danych. Po drugie, możemy opisywać wskaźnikowe struktury danych poprzez ich teorie algorytmiczne. Podejście takie jest przedmiotem innej rozprawy doktorskiej [Bał].

Można również znaleźć metody specyfikacji wskaźnikowych struktur danych oparte na logice monadycznej drugiego rzędu [KS93, KS94]. Są one bardzo użyteczne ze względu na rozstrzygalność wielu problemów związanych z weryfikacją specyfikacji, jednak rezultaty te są ograniczone do drzew i list.

W niniejszej pracy prześledzimy problem specyfikacji i weryfikacji wskaźnikowych struktur danych i programów na nich operujących, na przykładzie metody zwanej podejściem funkcyjnym, przedstawionym w [PM90, PM91, PM95] oraz w [EKM$^+$93, IKM$^+$95b, IMD97, IMPK93]. Jest ono też nazywane (np. w [BH97, HJL96]) *Parnas-Madey Four Variable Model*. W podejściu funkcyjnym w trakcie procesu wytwarzania programu powstaje szereg dokumentów opisujących ten program na różnych poziomach abstrakcji. Podejście to charakteryzuje się (w odróżnieniu np. od VDM czy RAISE) wyraźnym rozgraniczeniem poziomów abstrakcji, na których jest prezentowany program — dzięki temu nasze rozważania zyskują na przejrzystości.

1.1 Teza pracy

Celem niniejszej pracy jest wypełnienie opisanej luki między metodami formalnych specyfikacji, a dość powszechnie stosowanymi wskaźnikowymi strukturami danych. Chcemy zaproponować metodę specyfikacji wskaźnikowych struktur danych oraz metodę weryfikacji programów względem takich specyfikacji.

Zbadamy tutaj przydatność kilku formalizmów do specyfikowania zarówno wskaźnikowych struktur danych, jak i operacji modyfikujących te struktury. Przedstawimy też własną propozycję formalizmu służącego do takich celów. Dokonamy ich porównania na podstawie siły wyrazu oraz zwięzłości i czytelności specyfikacji przykładowych struktur danych.

Przedstawimy też wariant logiki Hoare'a, służącej do weryfikacji tzw. *while*-programów względem specyfikacji wyrażonych w wybranym przez nas formalizmie. Wykażemy, że przedstawiony system wnioskowania jest poprawny i zupełny w sensie Cook'a [Apt81, Coo78].

Tezą tej pracy jest więc stwierdzenie, że istnieje formalizm pozwalający na względnie zwięzłe specyfikowanie wskaźnikowych struktur danych i weryfikowanie programów względem specyfikacji zapisanych w tym formalizmie.

1.2 Struktura pracy

Struktura niniejszej pracy jest następująca. W rozdziale drugim znajduje się wprowadzenie do podejścia funkcyjnego, gdzie przedstawiamy również sposób modelowania stanów i modyfikacji wskaźnikowych struktur danych. Rozdział trzeci zawiera krótkie wprowadzenia do kilku formalizmów, z których korzystamy w dalszej części pracy. W rozdziale czwartym badamy przydatność różnych formalizmów do specyfikowania wskaźnikowych struktur danych, a także proponujemy własną logikę służącą do tego celu. W rozdziale piątym zajmujemy się problematyką weryfikacji programów operujących na wskaźnikowych strukturach danych, a prezentowane przez nas podejście jest ilustrowane za pomocą przykładu. Rozdział szósty zawiera podsumowanie uzyskanych wyników oraz uwagi końcowe.

1.3 Podstawowe pojęcia i stosowane notacje

Zakładamy, że Czytelnik jest zaznajomiony z logiką pierwszego rzędu (w skrócie *FOL*, ang. *first order logic*), logiką drugiego rzędu (w skrócie *SOL*, ang. *second order logic*) oraz algebrami wielorodzajowymi z funkcjami częściowymi. Dokładne omówienie tych tematów można znaleźć w licznych podręcznikach (np. [EM85, Ras84, ST, Tiu98, Wir90]). Zakładamy również, że Czytelnik zna podstawowe pojęcia z teorii grafów, algorytmiki oraz podstawowe wskaźnikowe struktury danych. Tematy te są omówione np. w [BDR96, CLR97, Lip89, RND85, Wil85, Wir99]. Dla ustalenia uwagi podajemy poniżej definicje niektórych pojęć oraz stosowane oznaczenia.

Bardzo często używamy pojęcia struktury danych. Przez *strukturę danych* rozumiemy zestaw zmiennych służących do przechowywania pewnych danych. Przykładowo, struktura danych przechowująca listę liczb całkowitych może mieć postać:

- zmiennej przechowującej długość listy oraz tablicy przechowującej elementy listy,
- rekordu złożonego z długość listy i tablicy zawierającej elementy listy,
- zestawu rekordów tworzących listę jednokierunkową.

Przez *wartość* struktury danych rozumiemy wartości zmiennych tworzących strukturę danych. Zauważmy, że w przypadku wskaźnikowych struktur danych pojęcie struktury danych i jej wartości są od siebie wzajemnie zależne. Przykładowo, mówiąc o liście jednokierunkowej jako o strukturze danych mamy na myśli rekordy tworzące tę listę. Wartość takiej struktury tworzą elementy listy przechowywane w rekordach oraz wartości wskaźników łączących rekordy w listę. Jednak do zapamiętania list o różnych liczbach elementów potrzebujemy list złożonych z różnej liczby rekordów. Tak więc, w przypadku wskaźnikowych struktur danych, mówiąc o *modyfikacji* lub *zmianie* struktury danych będziemy mieli na myśli nie tylko zmianę wartości struktury danych, ale i zmianę samego zestawu zmiennych tworzących daną strukturę danych. Przez specyfikację, definicję, opis itp. struktury danych rozumiemy warunki opisujące możliwe wartości struktury danych, a w przypadku wskaźnikowych struktur danych również postać samej struktury danych.

Ilekroć używamy równości, jest to tzw. silna równość, tzn. $t_1 = t_2$ wtw., gdy t_1 i t_2 są określone, a ich wartości są równe. Używamy też następujących skrótów: przez $t\downarrow$ rozumiemy $t = t$, oraz przez $t_1 \equiv t_2$ rozumiemy $t_1 = t_2 \vee (t_1 \neq t_1 \wedge t_2 \neq t_2)$. Inaczej mówiąc $t\downarrow$ oznacza, że

wartość t jest określona, a $t_1 \equiv t_2$ oznacza, że albo t_1 i t_2 są określone i ich wartości są równe, albo zarówno t_1 jak i t_2 są nieokreślone. Na oznaczenie definicji używamy symbolu $\hat{=}$.

Dla zwiększenia czytelności wszystkie pojawiające się w pracy definicje, przykłady, twierdzenia itp. zakończone są znakiem „□". Natomiast wszystkie dowody są zakończone znakiem „■".

W pracy tej wielokrotnie pojawiają się bardzo długie formuły. Często wiele wysiłku wymagało nadanie im takiej formy, aby mieściły się na szerokość strony. Dlatego też, nawiasy w formułach występują tylko tam, gdzie jest to konieczne. Tam, gdzie ich nie ma, przyjęto następujące zasady rozbioru formuł:

- wszystkie operatory jednoargumentowe (negacja, operatory modalne, kwantyfikatory i operatory punktu stałego) obejmują najmniejsze możliwe formuły znajdujące się po ich prawej stronie,
- koniunkcja wiąże silniej niż implikacja,
- implikacja wiąże silniej niż alternatywa,
- podział długiej formuły na kilka linii nie zmienia powyższych zasad.

W pracy tej często posługujemy się pojęciem zbioru wielorodzajowego. S-rodzajowym zbiorem X nazywamy rodzinę zbiorów indeksowaną elementami zbioru S, $X = \langle X_s \rangle_{s \in S}$. Dla ustalonego zbioru rodzajów S, na zbiorach wielorodzajowych możemy wykonywać operacje analogiczne do podstawowych operacji na zbiorach. Przykładowo:

$$\begin{gathered}\langle X_s \rangle_{s \in S} \cap \langle Y_s \rangle_{s \in S} = \langle X_s \cap Y_s \rangle_{s \in S}, \\ \langle X_s \rangle_{s \in S} \subseteq \langle Y_s \rangle_{s \in S} \Leftrightarrow \forall_{s \in S} X_s \subseteq Y_s, \\ \emptyset = \langle \emptyset \rangle_{s \in S}, \\ x \in X \Leftrightarrow x \in \bigcup_{s \in S} X_s.\end{gathered}$$

Jeżeli $X = \langle X_s \rangle_{s \in S}$ oraz $s_1, \ldots, s_n \in S$ są takie, że $\forall_{s \in S}(s \notin \{s_1, \ldots, s_n\} \Rightarrow X_s = \emptyset)$, to X zapisujemy również jako $\langle X_{s_1}, \ldots, X_{s_n} \rangle_S$.

Przez funkcję (funkcję częściową) f ze zbioru wielorodzajowego $X = \langle X_s \rangle_{s \in S}$ w $Y = \langle Y_s \rangle_{s \in S}$ rozumiemy rodzinę funkcji indeksowaną elementami zbioru S, $f = \langle f_s \rangle_{s \in S}$ taką, że f_s jest funkcją (funkcją częściową) $f_s : X_s \to Y_s$. Zamiast $f_s(x)$ piszemy $f(x : s)$, a o ile istnieje dokładnie jedno takie s, że $x \in X_s$, lub s wynika z kontekstu, to piszemy $f(x)$. W dalszej części pracy, o ile nie budzi to niejednoznaczności, zmiast „zbiory wielorodzajowe" i „funkcje wielorodzajowe" piszemy krótko „zbiory" i „funkcje".

Jeśli f jest funkcją (częściową), to przez $f[a \to b]$ oznaczamy taką funkcję (częściową), że $f[a \to b](a) = b$ oraz $\forall_{x \neq a} f[a \to b](x) \equiv f(x)$.

Niech p będzie dowolnym (niepustym) ciągiem (skończonym $p = \langle p_0, \ldots, p_n \rangle$, lub nieskończonym $p = \langle p_0, \ldots \rangle$). Przez $B(p)$ oznaczamy pierwszy element p, $B(p) = p_0$. Jeśli p zawiera więcej niż k elementów, to przez $p_{|k}$ oznaczamy ciąg powstały przez usunięcie z p k pierwszych elementów, czyli $p_{|k} = \langle p_k, \ldots, p_n \rangle$ gdy p jest skończone i $p_{|k} = \langle p_k, \ldots \rangle$, gdy p jest nieskończone.

Poniżej podajemy definicje sygnatury i algebry wielorodzajowej.

Def. 1 Sygnaturą algebry nazywamy parę $\Sigma = \langle S_\Sigma, F_\Sigma \rangle$, gdzie

- S_Σ jest zbiorem (nazw rodzai),

- $F_\Sigma = \langle F_{\Sigma,w} \rangle_{w \in S_\Sigma^+}$ jest S_Σ-rodzajowym zbiorem symboli funkcyjnych indeksowanym ich arnościami; $f \in F_{\Sigma,\langle s_1,\ldots,s_n,s\rangle}$ oznacza, że symbol funkcyjny f ma arność:

$$f : s_1, \ldots, s_n \to s \ .$$

□

O ile sygnatura jest znana z kontekstu, zamiast $f \in F_{\Sigma,\langle s_1,\ldots,s_n,s\rangle}$ piszemy:

$$f : s_1 \times \cdots \times s_n \to s \ .$$

Dopuszczamy przeładowanie nazw funkcji (tzn. F_Σ nie musi być rodziną zbiorów rozłącznych). W przypadku, gdy z kontekstu nie wynika arność symboli funkcyjnych, to symbole te indeksujemy ich arnościami, np. $f_{s_1 \times \cdots \times s_n \to s}$.

Def. 2 Niech $\Sigma = \langle S_\Sigma, F_\Sigma \rangle$ będzie sygnaturą. Algebra wielorodzajowa A o sygnaturze Σ z funkcjami częściowymi, lub krócej — częściowa Σ-algebra, składa się z:

- nośnika $|A|$, będącego S_Σ-rodzajowym zbiorem (elementów poszczególnych rodzai),
- dla każdego symbolu funkcyjnego $f : s_1 \times \cdots \times s_n \to s$, z jego interpretacji f^A będącej funkcją częściową $f^A : s_1 \times \cdots \times s_n \to s$.

Jeżeli interpretacje symboli funkcyjnych są funkcjami całkowitymi, to mówimy o algebrze całkowitej. □

Klasę wszystkich częściowych Σ-algebr oznaczamy przez $PAlg(\Sigma)$, a klasę wszystkich całkowitych Σ-agebr przez $Alg(\Sigma)$. Niech $\sigma : \Sigma_1 \to \Sigma_2$ będzie morfizmem sygnatur, A będzie Σ_2-algebrą częściową. Wówczas σ-redukt algebry A oznaczamy przez $A_{|\sigma}$. W dalszej części pracy posługujemy się algebrami częściowymi. Ilekroć więc mówimy po prostu o algebrze, oznacza to algebrę częściową.

Niech $\Sigma = \langle S_\Sigma, F_\Sigma \rangle$ bedzie ustaloną sygnaturą. Przez zbiór zmiennych rozumiemy S_Σ-rodzajowy zbiór (nazw zmiennych poszczególnych rodzai). Niech $X = \langle X_s \rangle_{s \in S_\Sigma}$ będzie zbiorem zmiennych, a A będzie Σ-algebrą. Wartościowaniem zmiennych z X w A nazywamy każdą (całkowitą) funkcję (wielorodzajową) $v : X \to |A|$. Przez $T_\Sigma(X) = \langle T_\Sigma(X)_s \rangle_{s \in S_\Sigma}$ oznaczamy S_Σ-rodzajowy zbiór termów (poszczególnych rodzai) nad zbiorem zmiennych X. Przez T_Σ oznaczamy $T_\Sigma(\emptyset)$. Każde wartościowanie $v : X \to |A|$ rozszerzamy w naturalny sposób na termy $v^A : T_\Sigma(X) \to |A|$.

Termami z dziurą $[] : s$ nad zbiorem zmiennych X ($[] \notin X$) nazywamy termy nad zbiorem zmiennych $X \cup \{[] : s\}$, gdzie „[]" (dziura) jest wyróżnionym symbolem zmiennej.

Zbiór formuł FOL nad sygnaturą Σ o zmiennych wolnych ze zbioru X oznaczamy przez $FOL_\Sigma(X)$. Jeśli $\varphi \in FOL_\Sigma(X)$, to przez $A \models_v \varphi$ oznaczamy fakt, że φ jest spełniona w A przy wartościowaniu v. Zbiór formuł SOL nad sygnaturą $\Sigma = \langle S_\Sigma, F_\Sigma \rangle$, o zmiennych wolnych pierwszego rzędu ze zbioru X i zmiennych wolnych drugiego rzędu ze zbioru $Y = \langle Y_w \rangle_{w \in S_\Sigma^+}$ (indeksowanego ich arnościami) oznaczamy przez $SOL_\Sigma(X, Y)$.

1.4 Podziękowania

Przede wszystkim chciałbym serdecznie podziękować mojemu promotorowi, profesorowi Janowi Madeyowi za nieustającą pomoc i opiekę jaką otacza mnie już od blisko czternastu lat, a także za bardzo cenne uwagi, bez których niniejsza praca straciłaby dużo na swej wartości.

Pragnę również gorąco podziękować wszystkim członkom zespołu kierowanego przez prof. Jana Madeya za wiele cennych rad i uwag, a w szczególności dr Janinie Mincer-Daszkiewicz, Marcinowi Engelowi, Krzysztofowi Stenclowi, Adamowi Bałabanowi, Arturowi Kretowi i Januszowi Jabłonowskiemu.

Szczególnie gorąco dziękuję Agnieszce Radomskiej za pomoc w zredagowaniu niniejszej rozprawy, a także za cierpliwość i wyrozumiałość, które okazywała mi w trakcie mojej pracy, oraz wsparcie w trudnych chwilach.

Niniejsza rozprawa została zrealizowana częściowo w ramach projektu badawczego KBN nr 8 T11C 015 15.

Rozdział 2

Podejście funkcyjne

W pracy tej przyjęliśmy, jako modelową metodykę tworzenia programów, metodykę zwaną podejściem funkcyjnym (ang. *functional approach*[1]) [PM90, PM91, PM95] oraz [EKM+93, IKM+95b, IMD97, IMPK93]. Podejście to jest również nazywane *Parnas-Madey Four Variable Model* [BH97, HJL96] — od wchodzącej w skład podejścia funkcyjnego tzw. *metody czterech zmiennych*, która służy do specyfikowania wymagań wobec wbudowanych systemów komputerowych czasu rzeczywistego. Ponieważ w pracy tej nie zajmujemy się tematyką związaną z systemami wbudowanymi, ani systemami czasu rzeczywistego, zainteresowanych Czytelników odsyłamy do [EKM+93, PM92, PM95, vSPM93].

Istotnym elementem podejścia funkcyjnego jest podział tworzonego programu na moduły. Zakładamy, że podział ten jest zgodny z zasadą ukrywania informacji (ang. *information hiding*) [Par72] — to co każdy moduł udostępnia na zewnątrz jest bezpośrednio związane z funkcjonalnością modułu, zaś wszystkie inne aspekty budowy modułu stanowią jego sekret. Zwykle takim sekretem modułu jest sposób implementacji pewnej struktury danych, a dla użytkowników modułu istotna jest funkcjonalność tej struktury danych. Przykładowo — moduł udostępnia strukturę słownikową wraz z podstawowymi operacjami na niej (funkcjonalność); struktura ta jest zaimplementowana jako drzewo AVL (sekret). Należy zaznaczyć, że może istnieć wiele implementacji struktury danych posiadających tę samą funkcjonalność. Taki podział na moduły nie tylko zwiększa czytelność i niezawodność programów, ale również, w odróżnieniu od innych metod podziału programów na moduły, zwiększa wzajemną niezależność i możliwość powtórnego wykorzystania (ang. *reusability*) modułów.

W podejściu funkcyjnym, każdy moduł udostępnia zmienne oraz operacje na nich. Struktura tych zmiennych oraz sposób implementacji operacji nie są znane na zewnątrz modułu. Zmienne charakteryzują się tym, że mają stany. Stany te mogą ulegać zmianie wyłącznie na skutek wykonania operacji udostępnianych przez moduł. Stany różnych zmiennych są niezależne, tzn. wykonanie operacji udostępnianej przez moduł może zmienić stany tylko tych zmiennych, które są argumentami operacji. Zmienne udostępniane przez ten sam moduł są homogeniczne, tzn. wynik operacji zależy jedynie od stanu jej argumentów, a nie od ich tożsamości[2].

[1] Choć podejście funkcyjne wywodzi swą nazwę od funkcji, to ang. *functional approach* nie bez powodu brzmi dwuznacznie i mogłoby się tłumaczyć jako podejście funkcjonalne. Należy zaznaczyć, że podejście funkcyjne nie ma nic wspólnego z programowaniem funkcyjnym.

[2] Przykładowo, jeżeli operacja ma dwa argumenty tego samego typu, to wynik operacji może zależeć od tego, czy przez argumenty zostaną przekazane dwie różne zmienne, czy też będzie to ta sama zmienna — nie może natomiast zależeć od tego, która to jest zmienna.

Ze względu na sposób w jaki moduły udostępniają zmienne, dzielimy je na scentralizowane i zdecentralizowane [IMD97]. Moduły scentralizowane posiadają w swej strukturze danych określoną pulę zmiennych, którymi zarządzają. W szczególnym przypadku może to być jedna zmienna. Moduły takie spotyka się np. w systemach operacyjnych. Moduły zdecentralizowane udostępniają na zewnątrz typ danych. Użytkownicy modułu kontrolują istnienie zmiennych tego typu. Moduł może wymagać jawnego inicjowania i niszczenia zmiennych za pomocą udostępnianych operacji. Ze względu na większą elastyczność, moduły zdecentralizowane stosuje się dużo częściej niż scentralizowane. Ponieważ rozróżnienie między modułami scentralizowanymi i zdecentralizowanymi jest niezależne od tematu pracy, bez zmniejszenia ogólności możemy skupić się na modułach zdecentralizowanych.

Przyjmujemy, że każdy moduł udostępnia dokładnie jeden typ danych (ukrywając strukturę tego typu — tzw. eksport nieprzezroczysty) oraz podstawowe operacje na obiektach tego typu. Jest to zgodne z zasadą ukrywania informacji. Nie jest to też tylko uproszczenie. Zwykle, gdy nie można opisać funkcjonalności modułu tak, aby udostępniał on tylko jeden typ danych, można go podzielić na dwa mniejsze moduły. Dla uproszczenia przyjmujemy, że nazwa modułu jest identyczna z nazwą typu udostępnianego przez moduł. Czasami używamy wymiennie pojęcia moduł i typ, dla oznaczenia modułu implementującego dany typ oraz dla typu udostępnianego przez dany moduł.

Sposób dekompozycji programu na moduły jest opisany w dokumencie nazywanym przewodnikiem po modułach (ang. *module guide*) [BP81, PCW85]. Dokładna postać tego dokumentu nie jest tutaj istotna. Na potrzeby tej pracy wystarczy powiedzieć, że przewodnik po modułach wymienia wszystkie moduły oraz określa, które moduły mogą korzystać z których tak, aby nie powstały cykliczne zależności między modułami. Co prawda języki programowania pozwalają na cykliczne zależności między częściami implementacyjnymi modułów, przyjmujemy jednak takie zależności za niepożądane — pozwala to na implementowanie różnych modułów w sposób całkowicie niezależny.

Każdy z modułów jest opisywany na trzech poziomach szczegółowości. Są to: specyfikacja interfejsu, projekt implementacji oraz implementacja modułu. Specyfikacja interfejsu modułu opisuje typ udostępniany przez moduł jako abstrakcyjny typ danych [Gut77, GH93, Hoa72] — inaczej mówiąc, opisuje jedynie zewnętrznie obserwowalne cechy modułu, jego funkcjonalność, bez szczegółów implementacyjnych (traktując moduł jak „czarną skrzynkę"). Specyfikacja interfejsu jest jedynym dokumentem opisującym dany moduł, którego znajomość jest potrzebna do korzystania z tego modułu. Projekt implementacji i implementacja modułu odnoszą się do sekretu modułu. Projekt implementacji opisuje sposób implementacji struktury danych modułu w wybranym języku programowania. Nie określa jednak sposobu implementacji operacji udostępnianych przez moduł, a jedynie określa w jaki sposób operacje te modyfikują opisywaną strukturę danych. Należy zaznaczyć, że dla jednej specyfikacji interfejsu może istnieć wiele różnych projektów implementacji. Implementacja modułu jest zapisana całkowicie w języku programowania. Zaznaczmy również, że dla danego projektu implementacji może istnieć wiele różnych implementacji.

W ramach podejścia funkcyjnego jako metoda specyfikacji interfejsów modułów zwykle występuje *metoda tropów*, w skrócie TAM (ang. *trace assertion method*). Ze względu na dość skomplikowaną notację stosowaną w tej metodzie pominiemy jej opis, odsyłając zainteresowanych do [IKM+97, Ste99] oraz [BIMO94, BP78, CP93, IMS94, Jan95, Kub94, PW89, Wan94]. Do specyfikacji interfejsu może być również użyta inna metoda specyfikacji abstrakcyjnych typów danych. Wymagamy jedynie, aby pozwalała ona na modularyzację specyfikacji, opisywała zewnętrznie obserwowalne zachowanie modułu oraz nie narzucała sposobu implementa-

cji modułu. Wewnętrzna struktura udostępnianych typów danych oraz sposób implementacji operacji nie powinny być widoczne na zewnątrz modułu. Jako metodę specyfikacji interfejsu modułów można zastosować np. algebraiczne specyfikacje równościowe [EM85, ST, Wir90] lub język specyfikacji Larch, w skrócie LSL (ang. *Larch Shared Language*) [GH93]. W niniejszej pracy nie chcemy przesądzać o metodzie użytej do specyfikacji interfejsów modułów. Wybór konkretnej metody jest nieistotny dla pozostałych rozważań. W podrozdziale 2.3 podajemy jedynie ogólną charakterystykę specyfikacji interfejsu.

Wybór konkretnej metody specyfikacji może zależeć od tego, czy przyjmiemy, że operacje udostępniane przez moduły są deterministyczne, czy też mogą być niedeterministyczne. Kwestia ta nie jest bezpośrednio związana z problemem rozważanym w pracy, choć jej rozstrzygnięcie ma wpływ na postać specyfikacji interfejsów i projektów implementacji modułów. Przyjmujemy wariant ogólniejszy, w którym operacje udostępniane przez moduły mogą być niedeterministyczne. Należy zaznaczyć, że problem niedeterminizmu jest powiązany z wzajemną niezależnością zmiennych udostępnianych przez moduły oraz specyfikacją jedynie zewnętrznie obserwowalnych aspektów modułu [IKM95a].

Operacje udostępniane przez moduł mogą być operacjami częściowymi, tzn. ich wywoływanie może być dozwolone tylko w określonych sytuacjach. Specyfikacja interfejsu modułu określa kiedy jest dozwolone wywołanie danej operacji i tylko dla takich sytuacji określa skutki jej wywołania. Niedozwolone wywołanie operacji może mieć dowolne następstwa, łącznie z zapętleniem się programu lub wystąpieniem błędu.

Bardziej szczegółowy opis modułu, to projekt jego implementacji [IMD97]. Opisuje on sekret modułu, jakim jest sposób implementacji struktury danych modułu w wybranym języku programowania. Wartości struktury danych tworzącej daną zmienną udostępnianą przez moduł nazywamy jej konkretnymi wartościami — w odróżnieniu od wartości abstrakcyjnych, do których odnosi się specyfikacja interfejsu modułu. W odniesieniu do wartości typu udostępnianego przez moduł, w projekcie implementacji używa się wartości konkretnych, a w odniesieniu do pozostałych typów, wartości abstrakcyjnych. Projekt implementacji zawiera deklarację typu udostępnianego przez moduł oraz typów pomocniczych. Zawiera on też niezmiennik struktury danych — warunek ograniczający możliwe wartości (konkretne) typu udostępnianego przez moduł. Dla każdej operacji jest wyspecyfikowane, kiedy może ona być wywoływana oraz w jaki sposób zmienia ona swoje argumenty. Projekt implementacji modułu definiuje również funkcję abstrakcji [AL91, Hoa72] (ang. *abstraction function, refinement mapping*) przekształcającą wartości konkretne na ich abstrakcyjne odpowiedniki. Funkcja ta służy do weryfikowania poprawności projektu implementacji względem specyfikacji interfejsu. Dokładniejszy opis projektu implementacji, wraz z przykładami, znajduje się w podrozdziałach 2.4–2.8.

Najbardziej szczegółowy opis modułu stanowi jego implementacja w wybranym języku programowania. Na potrzeby tej pracy jako język programowania przyjmujemy język Modula 2 [Wir91]. Język ten pozwala na modularyzację programów, a także na nieprzezroczysty eksport typów danych. Pozwala to na udostępnianie na zewnątrz modułu typów implementowanych przez moduł, bez odsłaniania jego wewnętrznej struktury. Dzięki temu Modula 2 idealnie pasuje do podejścia funkcyjnego. Czytelnicy nie znający języka Modula 2, a zaznajomieni z językiem Pascal [IMM92, JW76] oraz z mechanizmami modularyzacji nie powinni mieć żadnych problemów ze zrozumieniem fragmentów odnoszących się do języka Modula 2. Zamiast Moduli, w podejściu funkcyjnym można zastosować inny imperatywny język programowania. Należy jednak pamiętać, że w zależności od języka programowania, podejście funkcyjne może narzucać programiście pewne ograniczenia.

2.1 Typy wbudowane

W każdym z języków programowania jest dostępny pewien zestaw typów wbudowanch. Typów tych oczywiście nie trzeba implementować. W podejściu funkcyjnym typy wbudowane mają jedynie specyfikacje interfejsów. Nie trzeba ich też jawnie wymieniać na liście modułów wykorzystywanych. Zakładamy, że wśród typów wbudowanych znajduje się typ `BOOLEAN` wraz ze standardową interpretacją. Oczywiście typ `BOOLEAN` można wyspecyfikować za pomocą prostej formuły *FOL*. Dla uproszczenia, czasami utożsamiamy termy t rodzaju `BOOLEAN` z formułami postaci $t =$ `TRUE`. Pozwala nam to traktować funkcje o wyniku rodzaju `BOOLEAN` jako predykaty. Podejście takie można spotkać np. w metodzie specyfikacji Larch [GH93].

Dla prostoty zakładamy, że jedynymi operacjami dostępnymi dla typów wbudowanych są funkcje (bez efektów ubocznych) porównania i przypisania. Zakładamy też, że dana jest ustalona sygnatura $\Sigma_B = \langle S_B, F_B \rangle$ (`BOOLEAN` $\in S_B$) oraz ustalona Σ_B-algebra częściowa B, takie, że rodzaje zawarte w B reprezentują typy wbudowane, a funkcje zawarte w B modelują podstawowe operacje na tych typach.

2.2 Typy udostępniane

Sposób implementacji typu udostępnianego przez moduł powinien być sekretem modułu. Nie zawsze jest to do końca możliwe. Zauważmy, że zależnie od sposobu implementacji struktury danych różne może być znaczenie instrukcji przypisania dla zmiennych danego typu. Jeżeli dany typ jest zdefiniowany bez użycia typów wskaźnikowych, to przypisanie powoduje skopiowanie wartości struktury danych. Jeżeli jednak dany typ jest typem wskaźnikowym, to przypisanie powoduje skopiowanie wskaźnika, ale nie struktury przez niego wskazywanej. Przykładowo, jeżeli przypiszemy wartość zmiennej wskazującej na korzeń drzewa innej zmiennej, to obydwie zmienne będą wskazywać na korzeń tego samego drzewa. Z punktu widzenia użytkownika w pierwszym przypadku następuje skopiowanie wartości zmiennych, a w drugim zmienne stają się swoimi aliasami (tj. reprezentują tę samą strukturę danych). Możliwy jest też przypadek pośredni. Jeżeli przypiszemy wartość rekordu będącego korzeniem drzewa na inną zmienną, to powstanie struktura danych nie będąca poprawnym drzewem. Kwestia ta nie dotyczy tylko przypisania, lecz również przekazywania argumentów i porównywania. Zauważmy, że w przypadku bezwskaźnikowych struktur danych porównanie określa, czy wartości struktur danych są *takie same*, natomiast porównanie wskaźników określa, czy reprezentowane przez nie struktury są *te same*.

Pewne rozwiązanie tego problemu zaproponowano w metodzie Larch. W specyfikacji interfejsu modułu określa się, czy dany typ jest typem „zmiennym" czy „niezmiennym" (ang. *mutable* i *immutable*). Wartości typów zmiennych zachowują się tak jak wskaźniki do większych struktur danych. W szczególności, jeśli dwie zmienne są równe, to modyfikacja struktury reprezentowanej przez jedną z nich powoduje taką samą zmianę struktury reprezentowanej przez drugą z nich. Zmienne typów zmiennych wymagają również jawnego inicjowania i niszczenia reprezentowanych przez nie struktur. Typy niezmienne odpowiadają bezwskaźnikowym strukturom danych. Przypisanie wartości zmiennej typu niezmiennego powoduje skopiowanie całej struktury danych reprezentowanej przez tę zmienną.

Innym rozwiązaniem jest możliwość przedefiniowania porównania oraz przypisania dla typu udostępnianego przez moduł. Można tak przedefiniować operację porównania (na zewnątrz modułu) wartości typu udostępnianego przez moduł, że będzie ona powodować porównanie

wartości całych struktur wskaźnikowych. Podobnie, można tak przedefiniować operację przypisania (na zewnątrz modułu) wartości zmiennym typu udostępnianego przez moduł, że będzie ona powodować skopiowanie całej struktury wskaźnikowej. Jest to możliwe np. w języku C++ [Str97].

Sposób rozwiązania tego problemu nie wpływa na prezentowane tutaj rezultaty. Dlatego też przyjmiemy konwencję, dotyczącą typów nie wbudowanych, która uprości stosowaną przez nas notację. Przyjmujemy, że typy (nie wbudowane) udostępniane przez moduły są typami wskaźnikowymi. Jest to zgodne z wymogiem obowiązującym w wielu kompilatorach Moduli, aby typy eksportowane w sposób nieprzezroczysty były typami wskaźnikowymi. Zakładamy, że język programowania inicjuje te zmienne wartością `NIL` oraz, że taka inicjacja zachowuje niezmiennik struktury danych. Dla uproszczenia przyjmujemy, że wszystkie argumenty operacji są przekazywane przez zmienną, a wszystkie operacje mają postać procedur. Zakładamy też, że na zewnątrz danego modułu można używać przypisania wobec zmiennych typu udostępnianego przez ten moduł oraz porównania wartości tego typu, jeżeli moduł udostępnia odpowiednie operacje implementujące przypisanie i porównanie (o nazwach `assign` i `compare`). Ograniczenia te nie dotyczą typów wbudowanych — operacje na typach wbudowanych są opisane w p. 2.1.

2.3 Specyfikacja interfejsu modułu

Specyfikacja interfejsu modułu zawiera następujące części:

1. nazwa typu udostępnianego przez moduł,
2. lista wykorzystywanych modułów,
3. deklaracje udostępnianych operacji,
4. specyfikacje udostępnianych operacji.

Specyfikacje interfejsu będziemy modelować za pomocą algebr całkowitych. Dla naszych rozważań nie jest istotna dokładna postać specyfikacji interfejsu. Istotne jest, że określa ona następujące elementy:

- nazwę modułu i typu udostępnianego przez moduł,
- listę wykorzystywanych modułów,
- nagłówki operacji udostępnianych przez moduł,
- sygnaturę zawierającą nazwy rodzai reprezentujących typy występujące w specfikacji, oraz symbole funkcyjne opisujące operacje udostępniane przez moduł,
- semantykę specyfikacji, czyli jakie algebry (nad sygnaturą określoną przez specyfikację) spełniają tę specyfikację.

Lista wykorzystywanych modułów musi oczywiście być zgodna z przewodnikiem po modułach. Listy te muszą również być domknięte przechodnio, tzn. jeżeli moduł B występuje na liście modułów wykorzystywanych w specyfikacji interfejsu modułu A i moduł C występuje na liście modułów wykorzystywanych w specyfikacji interfejsu modułu B, to moduł C występuje

na liście modułów wykorzystywanych w specyfikacji modułu A. Wynika to stąd, że operacje udostępniane przez moduł B mogą mieć argumenty typu C. Operacje te powinny być widoczne w specyfikacji modułu A. Stąd specyfikacja modułu A powinna wykorzystywać moduł C.

Nazwa typu udostępnianego przez moduł, lista wykorzystywanych modułów oraz deklaracje operacji udostępnianych przez moduł pozwalają zdefiniować sygnaturę specyfikacji interfejsu modułu. Sygnatura ta, dla każdego typu t występującego w specyfikacji, zawiera dwa rodzaje t i $\uparrow t$, reprezentujące, odpowiednio, wartości danego typu oraz adresy (tj. wskaźniki do) zmiennych tego typu. Każdej operacji odpowiadają dwie funkcje całkowite o wyniku rodzaju `BOOLEAN`. Pierwszą z nich oznaczamy przez `legal` $m.o$, a drugą przez $m.o$, gdzie m jest nazwą modułu, a o nazwą operacji. (O ile nie będzie to powodować niejednoznaczności, $m.o$ będziemy czasem skracać do o, a `legal` $m.o$ do `legal` o.) Funkcje te nazywamy, odpowiednio, warunkiem początkowym i końcowym operacji. Funkcja `legal` $m.o$ określa dla jakich argumentów dozwolone jest wywołanie operacji o. Jej argumentami są wartości wszystkich argumentów operacji oraz adresy (wskaźniki do) tych spośród argumentów operacji, które mogą być aliasami pozostałych argumentów tej operacji. Jeśli dla danych argumentów operacji funkcja `legal` $m.o$ jest równa *true*, to wywołanie operacji jest dozwolone i zakończy się poprawnie. Funkcja $m.o$ określa działanie operacji o dla takich argumentów, dla których funkcja `legal` $m.o$ jest równa *true*. Jej argumentami są wartości wszystkich argumentów operacji (w momencie jej wywołania), adresy (wskaźniki do) tych spośród argumentów, które mogą być aliasami pozostałych argumentów operacji oraz nowe wartości argumentów.

Od specyfikacji interfejsu wymagamy, aby dla tych wartości argumentów operacji, dla których `legal` $m.o$ jest równe *true*, zawsze istniały takie nowe wartości argumentów operacji, dla których $m.o$ jest równe *true*. Innymi słowy, jeżeli dla danych argumentów operacja kończy się pomyślnie, to w momencie zakończenia jej argumenty (których wartości mogą ulec zmianie) mają pewne wartości.

Adresy argumentów pozwalają stwierdzić, czy różne argumenty (tego samego typu) operacji są swoimi aliasami. W przypadku gdy adresy argumentów są równe, to argumenty są swoimi aliasami, gdy adresy są różne, to argumenty są od siebie niezależne. W specyfikacji interfejsu adresy argumentów możemy jedynie porównywać ze sobą. Mówimy, że argumenty funkcji `legal` $m.o$ i $m.o$ są spójne, gdy dla argumentów operacji, które są aliasami, argumenty funkcji opisujące wartość argumentu operacji w momencie wywołania mają tę samą wartość, a także argumenty opisujące wartość argumentu operacji po wykonaniu operacji mają tę samą wartość. Funkcje `legal` $m.o$ i $m.o$ rozpatrujemy wyłącznie dla spójnych wartości argumentów.

Def. 3 Niech $Spec_t$ będzie specyfikacją interfejsu modułu t wykorzystującą moduły (nie wbudowane) $t_1, \ldots, t_n$. Ponadto niech $Spec_{t_1}, \ldots, Spec_{t_n}$ będą specyfikacjami interfejsów modułów $t_1, \ldots, t_n$. Sygnaturę $Sig(Spec_t) = \langle S_t, F_t \rangle$ specyfikacji $Spec_t$ definiujemy następująco:

- $S_t = S_B \cup \{\uparrow x \mid x \in S_B\} \cup \{t, t_1, \ldots, t_n, \uparrow t, \uparrow t_1, \ldots, \uparrow t_n\}$,
- niech $o_1, \ldots, o_k$ będą operacjami udostępnianymi przez moduł t, gdzie o_i (dla $i = 1, \ldots, k$) ma argumenty typów $u^i_1, \ldots, u^i_{l_i}$; oznaczmy przez $D_{i,j}$ (dla $j = 1, \ldots, l_i$):

$$D_{i,j} = \begin{cases} \uparrow u^i_j \times u^i_j & \text{gdy dla pewnego } h \neq j, u^i_h = u^i_j \\ u^i_j & \text{gdy dla każdego } h \neq j, u^i_h \neq u^i_j \end{cases},$$

$D_i = D_{i,1} \times \cdots \times D_{i,l_i}$, $E_i = u^i_1 \times \cdots \times u^i_{l_i}$. Ponadto niech $Sig(Spec_{t_j}) = \langle S_{t_j}, F_{t_j} \rangle$ (dla

$j = 1, \ldots, n$). Wówczas

$$F_t = F_B \cup \bigcup_{j=1,\ldots,n} F_{t_j} \cup \bigcup_{i=1,\ldots,k} \{\texttt{legal}\ t.o_i : D_i \to \texttt{BOOLEAN}, t.o_i : D_i \times E_i \to \texttt{BOOLEAN}\}$$

O ile nie będzie to powodować niejednoznaczności, będziemy skracać nazwy funkcji, pomijając nazwę modułu t. □

Zdefiniowane powyżej symbole mają następujące intuicyjne znaczenia:

- S_t zawiera nazwę typu udostępnianego przez moduł t, nazwy typów wbudowanych i nazwy typów wykorzystywanych (w $Spec_t$), oraz nazwy typów wskaźnikowych do tych wszystkich typów,
- D_i stanowi wspólną część arności funkcji $t.o_i$ i `legal` $t.o_i$; reprezentuje ono wartości wszystkich argumentów operacji $t.o_i$ oraz adresy (wskaźniki do) tych spośród argumentów operacji, które mogą być aliasami pozostałych argumentów tej operacji,
- E_i reprezentuje nowe wartości argumentów operacji $t.o_i$,
- F_t zawiera symbole funkcji opisujących operacje wbudowane, symbole funkcji opisujących operacje udostępniane przez moduły wykorzystywane oraz symbole funkcji opisujących operacje udostępniane przez moduł t.

Przykład 1 Moduł `stack` definiujący stosy elementów typu `elem` może:

- wykorzystywać moduł `elem`,
- udostępniać operacje[3]:

```
Push(stack, elem)
Pop(stack, elem)
Empty(stack, BOOLEAN)
Copy(stack, stack)
```

Sygnatura specyfikacji interfejsu takiego modułu zawiera (m.in.) rodzaje: `stack`, ↑`stack`, `BOOLEAN`, ↑`BOOLEAN`, `elem` i ↑`elem` oraz symbole funkcyjne:

```
legal stack.Push : stack × elem → BOOLEAN
stack.Push : stack × elem × stack × elem → BOOLEAN
legal stack.Pop : stack × elem → BOOLEAN
stack.Pop : stack × elem × stack × elem → BOOLEAN
legal stack.Empty : stack × BOOLEAN → BOOLEAN
stack.Empty : stack × BOOLEAN × stack × BOOLEAN → BOOLEAN
legal stack.Copy : ↑stack × stack× ↑stack × stack → BOOLEAN
stack.Copy : ↑stack × stack× ↑stack × stack × stack × stack → BOOLEAN
```

Zwróćmy uwagę, że tylko w przypadku operacji `Copy` dwa argumenty mogą być swoimi aliasami i tylko w tym przypadku wśród argumentów funkcji opisujących operację występują adresy argumentów operacji. □

[3]W literaturze spotyka się również inny zestaw podstawowych operacji na stosie. Zawiera on dodatkową operację `Top` udostępniającą wartość znajdującą się na wierzchołku stosu, a operacja `Pop` nie udostępnia wartości zdejmowanej ze stosu.

2.4 Projekt implementacji

Projekt implementacji modułu składa się z następujących części:

1. nazwy modułu,
2. listy nazw wykorzystywanych modułów,
3. deklaracji udostępnianych operacji,
4. deklaracji typów tworzących strukturę danych modułu,
5. niezmiennika struktury danych modułu,
6. definicji funkcji abstrakcji,
7. specyfikacji udostępnianych operacji.

Przypomnijmy, że nazwy modułów są takie same jak nazwy udostępnianych przez nie typów danych.

Na liście modułów wykorzystywanych w projekcie implementacji muszą pojawić się wszystkie te moduły, które są wymienione na liście modułów wykorzystywanych w specyfikacji interfejsu. Projekt implementacji może jednak wykorzystywać więcej modułów niż specyfikacja interfejsu, oczywiście zgodnie z przewodnikiem po modułach.

Lista deklaracji operacji jest powtórzeniem listy deklaracji operacji ze specyfikacji interfejsu modułu. Deklaracje typów tworzących strukturę danych modułu są zapisane w języku programowania (w naszym przypadku w Moduli), w oparciu o typy udostępniane przez moduły wymienione na liście wykorzystywanych modułów i typy wbudowane. Deklaracje te muszą zawierać deklarację typu udostępnianego przez moduł.

Dokładną postać deklaracji typów tworzących strukturę danych modułu oraz sposób modelowania stanu lub zmiany struktury danych modułu omawiamy w p. 2.5. Postać i znaczenie niezmiennika struktury danych omawiamy w podrozdziale 2.6. W podrozdziale 2.7 omawiamy rolę funkcji abstrakcji. We wszystkich tych podrozdziałach nie prezentujemy formalizmu służącego do opisu wskaźnikowych struktur danych — rozważania na ten temat są przedmiotem rozdziału 4.

2.5 Modelowanie struktur danych

Deklaracje typów tworzących strukturę danych modułu są zapisane we fragmencie języka Modula 2. W deklaracjach tych mogą pojawiać się wszystkie typy wymienione na liście typów wykorzystywanych oraz typy wbudowane. Muszą one zawierać deklarację typu udostępnianego przez moduł. Ponieważ zajmujemy się wskaźnikowymi strukturami danych, do konstrukcji typów danych używamy tylko typów rekordowych i wskaźnikowych. Pozostałe konstrukcje typów, takie jak typy tablicowe, okrojone, wyliczeniowe czy pliki, dla uproszczenia rozważań pomijamy. Trzeba przyznać, że pomijając typy tablicowe tracimy możliwość specyfikowania niektórych wskaźnikowych struktur danych, takich jak np. tablice mieszające. Możemy jednak dzięki temu pominąć wiele technikalii związanych z typami okrojonymi (używanymi do indeksowania tablic). Jednocześnie prezentowane przez nas wyniki zostają zachowane, jeżeli język struktur danych wzbogacimy o tablice. (Czytelnikom zainteresowanym modelowaniem stanów struktur danych zawierających wskaźniki i tablice polecamy rozprawę doktorską [Bał].)

Wymagamy również, aby deklaracje typów nie były złożone (tzn. konstrukcje typów nie mogą być zagnieżdżone) — deklarację złożoną można zawsze rozbić na kilka prostszych deklaracji. Dzięki temu uproszczeniu każdy typ wprowadzany przez deklaracje ma jawnie nadaną nazwę. W naszych rozważaniach zakładamy, że deklaracje nie powodują przysłaniania nazw typów.

Stany zadeklarowanej struktury danych modelujemy za pomocą algebr częściowych. Dla każdego typu mamy rodzaj reprezentujący wartości tego typu. Adresy zmiennych (typu t) reprezentujemy za pomocą wartości odpowiednich typów wskaźnikowych (oznaczanych przez $\uparrow t$). Jeżeli deklaracje nie wprowadzają takich typów wskaźnikowych, wprowadzamy odpowiednie rodzaje o nazwach postaci $\uparrow t$.

Dopuszczamy sytuację, w której projekt implementacji wprowadza kilka typów wskaźnikowych do tego samego typu. Wówczas przyjmujemy, że wskaźniki różnych typów wskazują zawsze na różne zmienne. Jeżeli mamy typ t oraz (jeden lub więcej) typów wskaźnikowych do t, to przez $\uparrow t$ oznaczamy ten typ wskaźnikowy do t, którego deklaracja występuje jako pierwsza.

Chcemy, aby wskaźniki do wartości typu udostępnianego przez moduł reprezentowały struktury danych udostępniane przez moduł. Można jednak wyobrazić sobie sytuację, w której zmienne typu udostępnianego przez moduł mogą zarówno reprezentować całe struktury udostępniane przez moduł, jak i być ich składowymi. Przykładowo, typem udostępnianym przez moduł implementujący listy jednokierunkowe jest zwykle typ wskaźnikowy do rekordów tworzących listę. Wówczas każdy wskaźnik tego typu zadeklarowany (lub alokowany) na zewnątrz modułu jest wskaźnikiem do początku pewnej listy i (z punktu widzenia użytkownika modułu) reprezentuje całą listę. Jednak typ ten jest również typem pola każdego rekordu tworzącego listę, będącego dowiązaniem do następnego rekordu na liście. Tak więc wskaźnik do rekordu może reprezentować zarówno całą strukturę udostępnianą przez moduł, jak i adres rekordu wchodzącego w skład takiej struktury. Problem ten rozwiązujemy przez zadeklarowanie dwóch typów wskaźnikowych do typu udostępnianego przez moduł.

Jeżeli mamy zadeklarowanych kilka typów wskaźnikowych do typu t udostępnianego przez moduł, to przyjmujemy, że ten z nich, który jest zadeklarowany jako ostatni, reprezentuje adresy zmiennych (typu udostępnianego przez moduł) zadeklarowanych (lub alokowanych) na zewnątrz modułu. Oznaczamy go przez $\Uparrow t$. W przeciwnym przypadku, tzn. gdy mamy zadeklarowany co najwyżej jeden typ wskaźnikowy do typu t, $\Uparrow t$ traktujemy jako synonim $\uparrow t$.

Sygnaturę algebry używanej do modelowania stanu struktury danych definiujemy następująco.

Def. 4 Niech i będzie danym projektem implementacji modułu udostępniającego typ t, u_1, $\ldots$, u_k będą modułami (nie wbudowanymi) wykorzystywanymi w projekcie implementacji i, natomiast $Spec_{u_1}, \ldots, Spec_{u_k}$ będą specyfikacjami interfejsów modułów $u_1, \ldots, u_k$, oraz niech i zawiera deklaracje typów postaci:

$$\begin{array}{l}\texttt{TYPE}\\ \quad t_1 \texttt{ = } d_1\texttt{;}\\ \quad\quad\vdots\\ \quad t_n \texttt{ = } d_n\texttt{;}\end{array}$$

Sygnaturę $Sig(i) = \langle S_i, F_i \rangle$ projektu implementacji i definiujemy następująco:

- $S_i = S_B \cup \{\uparrow v \mid v \in S_B\} \cup \{u_1, \ldots, u_k, t_1, \ldots, t_n, \uparrow u_1, \ldots, \uparrow u_k, \uparrow t_1, \ldots, \uparrow t_n\}$, (zauważmy, że $t \in \{t_1, \ldots, t_n\}$),

- dla każdego $v \in S_i$ oznaczmy przez f_v następujący zbiór nazw symboli funkcyjnych indeksowany ich arnościami:
  - jeżeli v jest typem udostępnianym przez moduł wykorzystywany w i lub typem wbudowanym, to $f_v = \emptyset$,
  - jeżeli v jest typem wskaźnikowym do typu nierekordowego v', to
$$f_v = \{\_\uparrow\colon v \to v', \texttt{NIL} :\to v\},$$
  - jeżeli v jest typem wskaźnikowym do typu rekordowego v' o polach $p_1 : v_1, \ldots p_l : v_l$, to
$$f_v = \{\_\uparrow\colon v \to v', \texttt{NIL} :\to v, \_|p_1 : v \to v_1, \ldots, \_|p_l : v \to v_l\},$$
  - jeżeli v jest typem rekordowym o polach $p_1 : v_1, \ldots p_l : v_l$, to
$$f_v = \left\{ \begin{array}{c} \_.p_1 : v \to v_1, \ldots, \_.p_l : v \to v_l, \\ (p_1 = \_, \ldots, p_l = \_) : v_1 \times \cdots \times v_l \to v \end{array} \right\},$$

  dodatkowo niech $\mathit{Sig}(\mathit{Spec}_{u_i}) = \langle S_{u_i}, F_{u_i} \rangle$ (dla $i = 1, \ldots, k$), wówczas
$$F_t = F_B \cup \bigcup_{i=1,\ldots,k} F_{t_i} \cup \bigcup_{v \in S_t} f_v$$

□

Symbole funkcyjne należące do $\mathit{Sig}(i)$ mają następujące intuicyjne znaczenia:

- `NIL` jest stałą reprezentującą pusty wskaźnik,
- $\_\uparrow$ wyznacza wartość wskazywaną przez wskaźnik, w przypadku adresu zmiennej daje ona w wyniku wartość zmiennej, funkcja ta jest określona dla adresów alokowanych zmiennych,
- $\_.p$ wyznacza wartość pola p rekordu,
- $(p_1 = \_, \ldots, p_l = \_)$ konstruuje wartość typu rekordowego na podstawie wartości pól rekordu,
- $\_|p$ na podstawie adresu rekordu wyznacza adres odpowiedniego pola rekordu.

Od tego miejsca niech Int_t będzie ustalonym projektem implementacji modułu udostępniającego typ t. Przez *typ* bedziemy rozumieć rodzaj należący do zbioru $S_B \cup \{u_1, \ldots, u_k, t_1, \ldots, t_n\}$.

Przykład 2 Przypomnijmy sobie moduł udostępniający stosy (por. przykład 1). Stosy takie mogą być implementowane za pomocą list jednokierunkowych elementów typu `elem`:

```
TYPE
  stack = POINTER TO rec;
  rec = RECORD
    d: elem;
    nast: stack
  END;
  pprec = POINTER TO stack;
  pstack = POINTER TO stack;
```

Typem udostępnianym przez moduł jest `stack`. Zakładamy, że `elem` jest typem udostępnianym przez moduł wykorzystywany w $Int_{\texttt{stack}}$. Na $Sig(Int_{\texttt{stack}})$ składają się (m.in.):

- rodzaje: `BOOLEAN`, `elem`, `stack`, `rec`, `pprec`, `pstack`, $\uparrow$`BOOLEAN`, $\uparrow$`elem`, $\uparrow$`pstack`, $\uparrow$`pprec` ($\uparrow$`rec` jest synonimem `stack`, $\uparrow$`stack` jest synonimem `pprec` i $\Uparrow$`stack` jest synonimem `pstack`),
- symbole funkcji:

_ $\uparrow$: `stack` $\to$ `rec`	`NIL` :$\to$ `stack`	_.`d` : `rec` $\to$ `elem`
_ $\uparrow$: `pprec` $\to$ `stack`	`NIL` :$\to$ `pprec`	_|`d` : `stack` $\to\uparrow$`elem`
_ $\uparrow$: `pstack` $\to$ `stack`	`NIL` :$\to$ `pstack`	_.`nast` : `rec` $\to$ `stack`
_ $\uparrow$: $\uparrow$`elem` $\to$ `elem`	`NIL` :$\to\uparrow$`elem`	_|`nast` : `stack` $\to$ `pprec`
_ $\uparrow$: $\uparrow$`pstack` $\to$ `pstack`	`NIL` :$\to\uparrow$`pstack`	(`d` = _, `nast` = _) : `elem` $\times$ `stack` $\to$ `rec`
_ $\uparrow$: $\uparrow$`pprec` $\to$ `pprec`	`NIL` :$\to\uparrow$`pprec`	

□

Spośród algebr częściowych należących do $PAlg(Sig(Int_t))$ interesują nas tylko te, które spełniają pewne naturalne warunki dotyczące operacji na wskaźnikach i rekordach.

Def. 5 Niech i będzie projektem implementacji modułu. Przez $DStr(i)$ oznaczamy klasę tych $Sig(i)$-algebr częściowych A, które spełniają następujące warunki:

- dla każdego typu rekordowego u o polach $p_1 : v_1, \ldots, p_l : v_l$ mamy:
  - funkcje wyznaczające wartości pól rekordów są funkcjami całkowitymi, tzn.:

$$\bigwedge_{i=1,\ldots,l} \forall_{r \in |A|_u} (r.p_i^A)\downarrow \ ,$$

  - funkcja konstruująca wartości typu rekordowego jest całkowita, tzn.:

$$\forall_{x_1 \in |A|_{v_1}, \ldots, x_l \in |A|_{v_l}} (p_1 = x_1, \ldots, p_l = x_l)\downarrow \ ,$$

  - zbiór wartości typu rekordowego jest izomorficzny z produktem kartezjańskim zbiorów wartości typów pól rekordów, tzn.:

$$\forall_{r_1, r_2 \in |A|_u} (r_1.p_1^A = r_2.p_1^A \wedge \ldots \wedge r_1.p_l^A = r_2.p_l^A) \Rightarrow r_1 = r_2 \ ,$$

$$\forall_{x_1 \in |A|_{v_1}, \ldots, x_l \in |A|_{v_l}} \bigwedge_{j=1,\ldots,l} (p_1 = x_1, \ldots, p_l = x_l)^A.p_j^A = x_j \ ,$$

- dla każdego typu wskaźnikowego u do typu v funkcja $\uparrow^A_{u \to v}$ jest nieokreślona dla `NIL` : u, tzn.:

$$\neg(\texttt{NIL}_u^A \uparrow^A)\downarrow \ ,$$

- dla każdego typu wskaźnikowego u do typu rekordowego r o polach $p_1 : v_1, \ldots, p_l : v_l$ mamy:

– funkcja $\_|p_j^A : u \to\uparrow v_j$ (dla $j = 1, \ldots, l$) jest określona dla wskaźników różnych od $\texttt{NIL} : u$, tzn.:

$$\forall_{x \in |A|_u} x \neq \texttt{NIL}_u^A \Leftrightarrow (x|p_j^A)\downarrow ,$$

– jeśli $x \in |A|_u$ jest adresem rekordu, to funkcja $\uparrow^A_{\uparrow v_j \to v_j}$ (dla $j = 1, \ldots, l$) jest określona dla $x|p_j^A$ wtw., gdy x jest adresem alokowanego rekordu, tzn.:

$$\forall_{x \in |A|_u} (x\uparrow^A)\downarrow \Leftrightarrow (x|p_j^A\uparrow^A)\downarrow ,$$

– wartość pola rekordu wskazywanego przez wskaźnik jest taka sama jak wartość wskazywana przez wskaźnik do tego pola rekordu, tzn.:

$$\forall_{x \in |A|_u} (x\uparrow^A)\downarrow \Rightarrow x\uparrow^A.p_j^A = x|p_j^A\uparrow^A ,$$

– różne pola tego samego typu, tego samego rekordu mają różne adresy, tzn. dla $1 \leq j, k \leq l$, $j \neq k$, $v_j = v_k$ mamy:

$$\forall_{x \in |A|_u} x|p_j^A \neq x|p_k ,$$

– pola tego samego typu, różnych rekordów tego samego typu mają różne adresy, tzn. dla $1 \leq j, k \leq l$, $v_j = v_k$ mamy:

$$\forall_{x,y \in |A|_u} x \neq y \Rightarrow x|p_j^A \neq y|p_k^A ,$$

- pola (tych samych typów) rekordów, których adresy są różnych typów mają różne adresy, tzn. dla każdych dwóch różnych typów wskaźnikowych u i u' do typów rekordowych (odpowiednio) r o polach $p_1 : v_1, \ldots, p_l : v_l$, oraz s o polach $q_1 : w_1, \ldots, q_m : w_m$, dla $1 \leq j \leq l$, $1 \leq k \leq m$, $v_j = w_k$ mamy:

$$\forall_{x \in |A|_u, y \in |A|_{u'}} x|p_j^A \neq y|q_k^A .$$

□

Zauważmy, że dla danej sygnatury $Sig(i)$ wszystkie powyższe warunki można wyrazić w FOL.

Algebry częściowe należące do $DStr(Int_t)$ mogą być używane do modelowania stanów struktury danych modułu t. Należy jednak pamiętać, że jedna algebra reprezentuje pojedynczy stan struktury danych. Naszą intencją jest modelowanie za pomocą podklas $DStr(Int_t)$ takich własności, jak niezmienniki struktury danych czy warunki wstępne operacji. Zuważmy jednak, że struktury te są niewystarczające do modelowania zmian stanu struktury danych modułu, np. na skutek wywołania operacji.

Zmianę stanu struktury danych możemy opisać za pomocą zmiany funkcji $\_\uparrow$. Do tego celu rozszerzymy sygnaturę $Sig(Int_t)$ tak, aby zawierała zdublowany zestaw symboli funkcji $\_\uparrow$ — jeden odnoszący się do stanu struktury danych przed zmianą i jeden odnoszący się do stanu po zmianie. Zastosujemy tu konwencję notacyjną, podobną do stosowanych w metodzie specyfikacji Z czy Larch, polegającą na „dekorowaniu" symboli funkcji $\uparrow$. Niedekorowane funkcje $\uparrow$ odnoszą się do stanu struktury danych „po" zmianie. Funkcje $\overset{\backprime}{\uparrow}$ odnoszą się do stanu struktury danych „przed" zmianą. W dalszej części pracy używamy również innych, pomocniczych, dekoracji $*$, $\odot$, itp., zapisywanych jako $\uparrow^*$, $\uparrow^\odot$, itd. Wsteczny prim „$\backprime$" traktujemy

jako dekorację, choć tradycyjnie zapisujemy go po lewej stronie symbolu, którego dotyczy, gdyż jest on lustrzanym odbiciem „′". Symbolu „′" (prim) nie traktujemy jak dekoracji, lecz pozostajemy przy jego tradycyjnym znaczeniu. Na potrzeby tej pracy, niech $\mathcal{D}$ będzie ustalonym, nieskończonym, zbiorem wszystkich dekoracji ($\backslash \in \mathcal{D}$). Dla uproszczenia notacji, symbole nie dekorowane $\uparrow$ będziemy czasami traktować tak, jak dekorowane słowem pustym ε.

Def. 6 Niech Σ będzie sygnaturą algebry, $\delta \in \mathcal{D}$ będzie symbolem dekoracji. Przez Σ^δ oznaczamy sygnaturę powstałą z Σ przez zastąpienie każdego symbolu funkcyjnego $\uparrow$ przez $\uparrow^\delta$ (każdego symbolu funkcyjnego $\uparrow^\delta$ przez $\uparrow^{\delta\delta}$, $\uparrow^{\delta\delta}$ przez $\uparrow^{\delta\delta\delta}$ itd.).

Niech $\iota : \Sigma \to \Sigma^\delta$ będzie morfizmem sygnatur przyporządkowującym symbolom z Σ ich odpowiedniki z Σ^δ, t będzie termem, a φ formułą nad sygnaturą Σ (w dowolnej z omawianych logik), to przez t^δ i φ^δ oznaczamy, odpowiednio, $t^\delta = \iota(t)$, $\varphi^\delta = \iota(\varphi)$.

Niech Σ będzie sygnaturą nie zawierającą symboli dekorowanych, $\Sigma^\delta \subseteq \Sigma_1$, wszystkie symbole funkcyjne z dekoracjami δ należące do Σ_1 należą również do Σ^δ, A będzie częściową Σ_1-algebrą, ι jest określona j.w., ι_1 jest naturalnym włożeniem Σ^δ w Σ_1. Wówczas przez $A_{|\delta}$ oznaczamy $(A_{|\iota_1})_{|\iota}$. □

Intuicyjnie, $A_{|\delta}$ jest częściową Σ-algebrą zawierającą funkcje $\_\uparrow$ równe ich δ-odpowiednikom z A.

Def. 7 Niech Σ nie zawiera symboli z dekoracjami $\backslash$. Przez $\Delta\Sigma$ oznaczamy sygnaturę $\Sigma \cup {}^\backslash\Sigma$.

Niech i będzie projektem implementacji modułu oraz $\mathit{Sig}(i) = \langle S_i, F_i \rangle$. Przez $\mathit{Decor}(i)$ oznaczymy zbiór sygnatur:

$$\mathit{Decor}(i) = \{\, \langle S_i, F_Y \rangle \,|\, Y \subseteq \mathcal{D} \setminus \{\backslash\}\} \quad ,$$

gdzie

$$F_Y = F_i \cup \bigcup_{\delta \in Y} F_i^\delta$$

oraz

$$\mathit{Sig}(i)^\delta = \langle S_i, F_i^\delta \rangle \quad .$$

Przez $\mathit{Sig}(i)^{\mathcal{D}}$ oznaczymy największą sygnaturę ze zbioru $\mathit{Decor}(i)$. Przez $\Delta\mathit{Decor}(i)$ oznaczamy zbiór sygnatur:

$$\Delta\mathit{Decor}(i) = \{\Delta\Sigma \mid \Sigma \in \mathit{Decor}(i)\} \quad .$$

Definicję DStr rozszerzamy na sygnatury $\Sigma_1 \in \mathit{Decor}(i) \cup \Delta\mathit{Decor}(i)$. Przez $\mathit{DStr}(\Sigma_1)$ oznaczamy klasę wszystkich takich częściowych Σ_1-algebr A, że $A_{|\varepsilon} \in \mathit{DStr}(i)$, oraz dla każdej dekoracji δ występującej w Σ_1 mamy $A_{|\delta} \in \mathit{DStr}(i)$. □

Algebry częściowe należące do $\mathit{DStr}(\Delta\mathit{Sig}(\mathit{Int}_t))$ mogą być wykorzystywane do modelowania zmian stanów struktury danych modułu t. Należy jednak pamiętać, że jedna taka algebra modeluje jedną możliwą zmianę stanu struktury danych. Naszą intencją jest modelowanie zmian, jakie mogą powodować operacje udostępniane przez moduł, za pomocą podklas $\mathit{DStr}(\Delta\mathit{Sig}(\mathit{Int}_t))$.

Def. 8 Niech i będzie projektem implementacji modułu, $\Sigma \in Decor(i) \cup \Delta Decor(i)$, $A \in DStr(\Sigma)$. Przez $\overline{A}$ oznaczamy zbiór takich algebr częściowych $A_1 \in DStr(\Sigma)$, że A_1 różni sie od A wyłącznie interpretacją symboli funkcyjnych postaci $\uparrow^\delta$ (dla $\delta \in \mathcal{D} \cup \{\varepsilon\}$). Zauważmy, że dla dowolnego $A_1 \in \overline{A}$, $|A_1|$ jest takie samo. Możemy więc przez $|\overline{A}|$ oznaczyć $|A_1|$, gdzie $A_1 \in \overline{A}$. □

Intuicyjnie, $\overline{A}$ ustala interpretację rodzai oraz symboli funkcyjnych nie reprezentujących stanu struktury danych.

Niech $A \in DStr(\Delta Sig(Int_t))$. Zmiany stanu struktury danych modelujemy za pomocą podzbiorów $\overline{A}$, natomiast takie własności jak warunki wstępne operacji czy niezmienniki struktury danych modelujemy za pomocą podzbiorów $\overline{A_{|\setminus}}$.

2.6 Niezmiennik struktury danych

Zajmiemy się teraz znaczeniem niezmiennika struktury danych w prezentowanym podejściu. Niezmiennik struktury danych jest formułą, bez zmiennych wolnych, nad sygnaturą $Sig(Int_t)$. Nie precyzujemy tutaj logiki, w której jest wyrażony ten niezmiennik — jest to przedmiotem rozważań w rozdziale 4. Przyjmujemy, że w sposób niejawny (tj. bez konieczności zapisywania tego w projekcie implementacji) niezmiennik struktury danych obejmuje wszystkie warunki, jakie nałożyliśmy na algebry częściowe modelujące struktury danych w definicji 5. Inaczej mówiąc, niezmiennik struktury danych może być spełniony wyłącznie dla struktur należących do $DStr(Int_t)$. Tak więc formalizm, w którym zapisujemy niezmiennik struktury danych, powinien mieć siłę wyrazu wystarczającą do wyrażenia tych warunków, np. może on zawierać w sobie FOL.

Każda z operacji udostępnianych przez moduł zastaje strukturę danych spełniającą niezmiennik i po jej wykonaniu struktura danych powinna ponownie spełniać niezmiennik. Początkowy stan struktury danych reprezentują algebry częściowe należące do $DStr(Int_t)$, w których wszystkie funkcje $\_\uparrow$ są całkowicie nieokreślone. Wymagamy, aby dla takich struktur był spełniony niezmiennik struktury danych. Ponadto wymagamy, aby automatyczna inicjacja zmiennych typu udostępnianego przez moduł zachowywała niezmiennik. Inaczej mówiąc, jeżeli $A \in DStr(Int_t)$ jest algebrą spełniającą niezmiennik, p jest wartością rodzaju $\Uparrow t$, dla której $p\uparrow$ jest nieokreślone, A_1 jest strukturą powstałą z A przez takie rozszerzenie funkcji $\uparrow_{\Uparrow t \to t}$, że $\uparrow^{A_1}_{\Uparrow t \to t} = \uparrow^{A}_{\Uparrow t \to t} [p \to \mathtt{NIL}^A_t]$, to A_1 również spełnia niezmiennik danych.

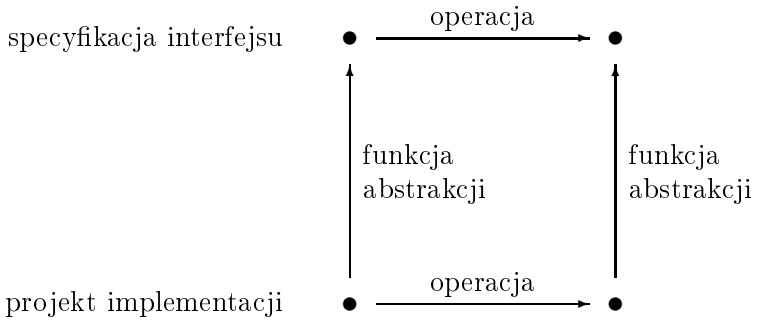
Def. 9 Niech i będzie projektem implementacji modułu. Przez $PDStr(i)$ oznaczamy podklasę $DStr(i)$ zawierającą wszystkie takie algebry częściowe, które spełniają niezmiennik struktury danych z i.

Niech $\Sigma_1 \in Decor(i) \cup \Delta Decor(i)$. Przez $PDStr(\Sigma_1)$ oznaczamy podklasę $DStr(\Sigma_1)$ zawierającą wszystkie takie algebry częściowe A, dla których $A_{|\varepsilon} \in PDStr(i)$ oraz dla każdej dekoracji δ pojawiającej się w Σ_1 mamy $A_{|\delta} \in PDStr(i)$. □

2.7 Funkcja abstrakcji

Funkcja abstrakcji przyporządkowuje wartościom konkretnym struktury danych modułu odpowiednie wartości abstrakcyjne zmiennych typu udostepnianego przez moduł. Funkcja ta pozwala powiązać projekt implementacji modułu ze specyfikacją jego interfejsu. Weryfikacja projektu implementacji względem specyfikacji interfejsu polega na wykazaniu, że operacje

udostępniane przez moduł zmieniają wartości konkretne zmiennych typu udostępnianego przez moduł w sposób zgodny ze specyfikacją interfejsu modułu. Można to zilustrować za pomocą następującego diagramu.



W przypadku wskaźnikowych struktur danych funkcja abstrakcji, dla danej algebry należącej do $PDStr(Int_t)$ oraz wskaźnika $p :\Uparrow t$, wyznacza abstrakcyjną wartość struktury danych reprezentowanej przez zmienną o adresie p. Problem sposobu definiowania takiej funkcji abstrakcji oraz szczegóły weryfikacji projektu implementacji względem specyfikacji interfejsu wykraczają poza ramy tej pracy. W przypadku wskaźnikowych struktur danych będą one przedmiotem przyszłych badań.

2.8 Operacje

Każda z operacji udostępnianych przez moduł jest opisana w projekcie implementacji przez dwie formuły — podobnie jak w specyfikacji interfejsu. Pierwsza z nich opisuje warunek wstępny wykonania operacji, a druga opisuje, w jaki sposób struktura danych może ulec zmianie na skutek wykonania operacji (warunek końcowy operacji). Formuły te, podobnie jak ich odpowiedniki ze specyfikacji interfejsu modułu, oznaczamy przez `legal` $m.o$ i $m.o$ (lub, jeśli nie budzi to niejednoznaczności, krócej przez `legal` o i o), gdzie m jest nazwą modułu, a o nazwą operacji. Formuła `legal` o jest nad sygnaturą $Sig(Int_t)$, a o nad sygnaturą $\Delta Sig(Int_t)$. Formuły te, dla każdego argumentu formalnego operacji mają po jednym argumencie (zmiennej wolnej) reprezentującym wskaźnik do argumentu aktualnego operacji. Argumentom formuł nadajemy nazwy postaci $\&x$, jednocześnie x traktujemy jako skrót od $\&x\uparrow$. W ten sposób możemy równocześnie modelować wartości zmiennych (nazwanych i dynamicznych) za pomocą funkcji $\_\uparrow$, oraz w intuicyjny sposób odwoływać się do wartości zmiennych nazwanych.

Dla każdego stanu struktury danych należącego do $PDStr(Int_t)$ oraz argumentów spełniających warunek wstępny `legal o`, operacja `o` musi się pomyślnie kończyć. Inaczej mówiąc, jeśli dla $A \in PDStr(Int_t)$ oraz dla adresów argumentów operacji $p_1, \ldots, p_k$ mamy $A \models$ `legal o`$(p_1, \ldots, p_k)$, to istnieje taka algebra częściowa $A_1 \in PDStr(\Delta Sig(Int_t))$, że $A_{1\,|\backslash} = A$ oraz $A_1 \models$ `o`$(p_1, \ldots, p_k)$.

Podobnie, jak w przypadku niezmiennika struktury danych, nie prezentujemy tu logiki, w której zapisujemy formuły specyfikujące operacje. Rozdział 4 jest poświęcony rozważaniom na ten temat.

Przykład 3 Przypomijmy sobie deklaracje operacji udostępnianych przez moduł implementujący stos (por. przykład 1) oraz implementację struktury danych tego modułu za pomocą

list jednokierunkowych (por. przykład 2). Formuły opisujące działanie tych operacji mogą mieć następujące argumenty:

`legal stack.Push`$(\&s\colon$ `pstack`$, \&e\colon\uparrow$ `elem`$)$	`stack.Push`$(\&s\colon$ `pstack`$, \&e\colon\uparrow$ `elem`$)$
`legal stack.Pop`$(\&s\colon$ `pstack`$, \&e\colon\uparrow$ `elem`$)$	`stack.Pop`$(\&s\colon$ `pstack`$, \&e\colon\uparrow$ `elem`$)$
`legal stack.Empty`$(\&s\colon$ `pstack`$, \&b\colon\uparrow$ `BOOLEAN`$)$	`stack.Empty`$(\&s\colon$ `pstack`$, \&b\colon\uparrow$ `BOOLEAN`$)$
`legal stack.Copy`$(\&s_1\colon$ `pstack`$, \&s_2\colon$ `pstack`$)$	`stack.Copy`$(\&s_1\colon$ `pstack`$, \&s_2\colon$ `pstack`$)$

□

2.9 Implementacja

Implementacja modułu jest zapisana w konkretnym języku programowania.

W rozdziale 5 będziemy zajmować się problemem weryfikacji implementacji względem projektu implementacji. Ograniczymy się jednak do weryfikacji *while*-programów. W podrozdziale tym przedstawiamy składnię oraz semantykę *while*-programów operujących na wskaźnikowych strukturach danych.

Na potrzeby tej pracy jako język implementacji przyjęliśmy język Modula 2. Dlatego też zaczerpniemy formę zapisu *while*-programów z tego języka. Jako że składnia Moduli jest opisana w [Wir91], podamy tu jedynie gramatykę wyznaczającą interesujący nas fragment języka. Oczywiście *while*-programy muszą spełniać wszelkie warunki kontekstowe narzucone przez język Modula 2.

Wobec lokalnych zmiennych programistycznych przyjmiemy podobną konwencję notacyjną jak w przypadku argumentów operacji (por. p. 2.8). Wartości zmiennych, zarówno nazwanych (deklarowanych) jak i nienazwanych (alokowanych dynamicznie), modelujemy za pomocą funkcji $\_\uparrow$. W formułach logicznych opisujących wykonanie programów każdy argument operacji oraz każdą lokalną zmienną programistyczną $x : t$ modelujemy za pomocą zmiennej $\&x : \uparrow t$ reprezentującej adres x, natomiast x traktujemy jako skrót od $\&x\uparrow$. W odróżnieniu od argumentów operacji, różne zmienne lokalne nie są nigdy swoimi aliasami, ani nie są aliasami innych zmiennych. Zauważmy, że sposób alokacji zmiennych lokalnych można opisać prostą formułą *FOL*.

Dla uproszczenia zakładamy, że zmienne lokalne nie przysłaniają argumentów operacji. Jeśli tak nie jest, to zawsze możemy je przemianować. Zakładamy też, że deklaracje zmiennych lokalnych nie wprowadzają implicite nowych typów, a jedynie odwołują się do typów zadeklarowanych w projekcie implementacji.

Def. 10 Niech A będzie ustalonym zestawem argumentów operacji postaci: $a_1 : t_1, \ldots, a_k : t_k$, V będzie zestawem deklaracji lokalnych zmiennych programistycznych postaci:

$$\begin{array}{l}\texttt{VAR}\\ \quad x_1\colon v_1\,;\\ \quad\quad\vdots\\ \quad x_n\colon v_n\,;\end{array}$$

Zbiorem zmiennych (logicznych) indukowanym przez A i V nazywamy taki zbiór zmiennych $X = \langle X_s \rangle_{s \in Sig(Int_t)}$, że:

$$X_s = \{\&a_i \mid \uparrow t_i = s\} \cup \{\&x_i \mid \uparrow v_i = s\} \ .$$

Jeśli zbiór zmiennych X jest znany z kontekstu, $\&x \in X_{\uparrow s}$, to x traktujemy jako skrót od $\&x\uparrow$. □

Na potrzeby tego podrozdziału, niech $\Sigma = \mathit{Sig}(\mathit{Int}_t)$, $t.o$ bedzie operacją udostępnianą przez moduł t, X będzie zbiorem zmiennych indukowanym przez argumenty $m.o$ oraz deklaracje zmiennych lokalnych implementacji $m.o$.

Def. 11 Przez $\mathit{Prog}_t(X)$ oznaczamy takie *while*-programy napisane w języku Modula 2, że w ich wyprowadzeniach (zgodnie z opisem zawartym w [Wir91]), wyprowadzenia symboli nieterminalnych: instrukcja, desygnator i deklaracja są zgodne z następującą gramatyką[4]:

while-program	→	ciąg_instrukcji | deklaracja `BEGIN` ciąg_instrukcji `END ;`
deklaracja	→	`VAR` deklaracje_zmiennych
deklaracje_zmiennych	→	deklaracja_zmiennych |
		deklaracja_zmiennych deklaracje_zmiennych
deklaracja_zmiennych	→	identyfikator `:` identyfikator `;`
ciąg_instrukcji	→	instrukcja | instrukcja `;` ciąg_instrukcji
instrukcja	→	ε | desygnator `:=` wyrażenie | `NEW(` desygnator `)` |
		`IF` wyrażenie `THEN` ciąg_instrukcji `ELSE` ciąg_instrukcji `END` |
		`WHILE` wyrażenie `DO` ciąg_instrukcji `END`
desygnator	→	identyfikator | desygnator `.` identyfikator | wyrażenie ↑

gdzie wyrażenia (gdy zamienić w nich nazwy zmiennych programistycznych x na $\&x\uparrow$) są termami odpowiednich rodzai, oraz dla każdej deklarowanej zmiennej lokalnej x mamy $\&x \in X$. □

Desygnatory reprezentują adresy zmiennych danego typu. Są one nazywane też L-wyrażeniami (ang. *L-values*). Definiujemy tłumaczenie desygnatorów na termy odpowiednich rodzai wskaźnikowych. W ten sposób określamy ich semantykę.

Def. 12 Niech o będzie desygnatorem zmiennej typu s. Przez $\&(o)$ oznaczamy term, rodzaju wskaźnikowego do rodzaju s, zdefiniowany następująco:

- jeśli o jest nazwą zmiennej programistycznej, to $\&(o) = \&o$,
- jesli o jest postaci $o_1.p$, to $\&(o) = \&(o_1)|p$,
- jeśli o jest postaci $o_1\uparrow$, to $\&(o) = o_1$.

□

Wyrażenia występujące w programach traktujemy jako termy, a wyrażenia logiczne jako formuły (por. p. 2.1). Należy jednak pamiętać, że jeśli wartość termu jest nieokreślona, to wyliczenie jego wartości powoduje przerwanie obliczeń. Przez $\mathit{OK}(\alpha)$ oznaczamy formułę określającą, kiedy wartość wyrażenia logicznego α jest określona.

Def. 13 Niech α będzie wyrażeniem logicznym mogącym pojawić się w programie. Przez $\mathit{OK}(\alpha)$ oznaczamy formułę zdefiniowaną indukcyjnie ze względu na strukturę α:

[4]Liczba mnoga użyta w symbolu nieterminalnym „deklaracje_zmiennych" może wydawać się myląca. Wynika ona z ograniczenia składni opisanej w [Wir91] przy jednoczesnym zachowaniu tych samych nazw wybranych symboli nieterminalnych.

- jeśli $\alpha = \texttt{TRUE}$ lub $\alpha = \texttt{FALSE}$, to $OK(\alpha) = true$,
- jeśli $\alpha = \alpha_1 \texttt{ AND } \alpha_2$, to $OK(\alpha) = OK(\alpha_1) \wedge (\neg\alpha_1 \vee OK(\alpha_2))$ — podobnie postępujemy dla pozostałych operatorów boolowskich,
- jeśli $\alpha = (t_1 = t_2)$, to $OK(\alpha) = t_1\downarrow \wedge t_2\downarrow$,
- w pozostałych przypadkach $OK(\alpha) = (\alpha)\downarrow$.

□

Do zdefiniowania semantyki *while*-programów jest pomocna funkcja wyznaczająca stan struktury danych powstałej poprzez zapamiętanie określonej wartości na określonej zmiennej.

Def. 14 Niech $A \in DStr(\Sigma)$, t_1 będzie rodzajem wskaźnikowym do t_2, $a \in |A|_{t_1}$, $b \in |A|_{t_2}$. Przez $Store(A, a, b)$ oznaczamy taką strukturę $A_1 \in DStr(Sig(Int_t))$, że A_1 powstaje z A poprzez zapamiętanie wartości b pod adresem a. □

Zauważmy, że $\_\uparrow^{A_1}_{t_1\to t_2} = \_\uparrow^{A}_{t_1\to t_2} [a \to b]$. Ponadto, jeżeli $t_1 = \uparrow t_2$, t jest typem rekordowym zawierającym pola typu t_2, u jest rodzajem wskaźnikowym do typu t, to funkcja $\_\uparrow_{u\to t}$ zmienia się również odpowiednio. Tak samo, jeśli t_2 jest typem rekordowym zawierającym pola typu v, to funkcja $\uparrow_{\uparrow v\to v}$ również zmienia się odpowiednio.

Def. 15 Niech t będzie typem. Powiemy, że $p \in |A|_{\uparrow t}$ jest alokacją, gdy dla każdego typu wskaźnikowego u do typu rekordowego zawierającego pole r typu t zachodzi $\forall_{x\in|A|_u} x|r \neq p$. Przez $loc(p)$ oznaczamy formułę *FOL* wyrażającą powyższy warunek.

Jeżeli $p \in |A|_{\uparrow t}$ jest alokacją, to przez $Dealloc(A, p)$ oznaczamy taką strukturę $A_1 \in DStr(Sig(Int_t))$, że A_1 powstaje z A poprzez usunięcie p z dziedziny funkcji $\uparrow_{\uparrow t\to t}$, oraz, jeżeli t jest typem rekordowym, przez usunięcie adresów wszystkich składowych zmiennej wskazywanej przez p z dziedzin odpowiednich funkcji $\uparrow$. □

Intuicyjnie alokacja jest adresem zmiennej, która może być alokowana i usuwana. Funkcja *Dealloc* reprezentuje zwolnienie pamięci zajmowanej przez zmienną. Warunek, aby p było alokacją wynika stąd, że nie można usunąć np. części rekordu.

Semantykę *while*-programów definiujemy w postaci funkcji, która dla danego programu, stanu struktury danych oraz wartościowania zmiennych wyznacza zbiór możliwych stanów struktury danych po wykonaniu danej operacji.

Def. 16 Niech $P \in Prog_t(X)$, $A \in DStr(\Sigma)$, $v : X \to |A|$ będzie wartościowaniem zmiennych. Semantykę *while*-programu P oznaczamy przez $[\![P]\!]_v(A)$ i definiujemy indukcyjnie ze względu na strukturę programu w następujący sposób:

- $[\![I; P_1]\!]_v(A) = \overrightarrow{[\![P_1]\!]}_v([\![I]\!]_v(A))$,
- $[\![\varepsilon]\!]_v(A) = \{A\}$,
- $[\![o \texttt{ := } e]\!]_v(A) = \begin{cases} \{Store(A, v^A(\&(o)), v^A(e))\} & \text{if } A \models_v o\downarrow \wedge e\downarrow \\ \emptyset & \text{wpp.} \end{cases}$,
- $[\![\texttt{NEW}(o)]\!]_v(A) = \left\{ A_1 \;\middle|\; \begin{array}{l} A \models_v o\downarrow \wedge \\ \exists_{x\in|A|_{t_2}, y\in|A|_{t_3}} \begin{pmatrix} \neg(x\uparrow^A)\downarrow \wedge loc(x) \wedge \\ A_1 = Store(Store(A, v^A(\&(o)), x), x, y) \end{pmatrix} \end{array} \right\}$

  gdzie t_1 jest rodzajem $\&(o)$, t_2 jest rodzajem o, t_2 jest rodzajem wskaźnikowym do t_3,

- $[\![\texttt{VAR } x_1 : u_1; \ldots; x_k : u_k; \texttt{ BEGIN } P_1 \texttt{ END};]\!]_v(A) =$

$$\bigcup_{\substack{p_1 \in |A|_{\uparrow u_1}, \ldots, p_k \in |A|_{\uparrow u_k} \\ y_1 \in |A|_{u_1}, \ldots, y_k \in |A|_{u_k}}} \left\{ A_2 \left| \begin{array}{l} \bigwedge_{i=1,\ldots,k} \left(\neg (p_i \uparrow^A) \downarrow \wedge \mathit{loc}^A(p_i)\right), \\ \bigwedge_{\substack{1 \le i < j \le k \\ u_i = u_j}} p_i \neq p_j, \\ v_1 = v[\&x_1 \to p_1] \ldots [\&x_k \to p_k], \\ A_1 \in [\![P_1]\!]_{v_1}(\mathit{Store}(\ldots \mathit{Store}(A, p_1, y_1) \ldots, p_k, y_k)), \\ A_2 = \mathit{Dealloc}(\ldots \mathit{Dealloc}(A_1, p_1) \ldots, p_k) \end{array} \right. \right\},$$

- $[\![\texttt{IF } e \texttt{ THEN } Q \texttt{ ELSE } R \texttt{ END}]\!]_v(A) = \begin{cases} [\![Q]\!]_v(A) & \text{if } A \models_v \mathit{OK}(e) \wedge e \\ [\![R]\!]_v(A) & \text{if } A \models_v \mathit{OK}(e) \wedge \neg e \\ \emptyset & \text{wpp.} \end{cases},$

- $[\![\texttt{WHILE } e \texttt{ DO } P \text{ END}]\!]_v(A) =$

$$\left\{ B \left| \exists_{\langle A_0, \ldots, A_n \rangle \in \overline{A}^+} \left(\begin{array}{l} A_0 = A \ \wedge \ A_n = B \wedge \\ \bigwedge_{i=0,\ldots,n-1} \left(\begin{array}{l} A_i \models_v (\mathit{OK}(e) \wedge e) \wedge \\ A_{i+1} \in [\![P]\!]_v(A_i) \end{array} \right) \wedge \\ A_n \models_v \mathit{OK}(e) \wedge \neg e \end{array} \right) \right. \right\}.$$

□

2.10 Podejście funkcyjne, a inne metody specyfikacji

Podejście funkcyjne jest z jednej strony podejściem całościowym, gdyż obejmuje prawie cały proces wytwarzania oprogramowania, z drugiej zaś strony podejściem ogólnym, gdyż na różnych poziomach opisu oprogramowania możemy stosować różne formalizmy. Dlatego też nie wszystkie metody formalnych specyfikacji stanowią alternatywę dla podejścia funkcyjnego — niektóre z nich mogą być użyte w ramach podejścia funkcyjnego.

Można pomyśleć o wykorzystaniu takich metod jak algebraiczne specyfikacje równościowe, [EM85, ST, Wir90], Larch [GH93], czy Z [Spi92, Wor92] do specyfikowania interfejsów modułów. Takie metody weryfikacji jak logika Hoare'a [Apt81, Hoa69, Dah92], transformatory predykatów Dijkstry [Dij85] czy logika algorytmiczna [MS87, MS92] mogą być zastosowane do weryfikacji poprawności implementacji względem projektów implementacji.

Porównując podejście funkcyjne z takimi metodami specyfikacji jak VDM [Jon84] czy RAISE [GHH+92, GHH+95] widać, że główna różnica polega na tym, że w metodach tych specyfikacje mogą być stopniowo uszczegóławiane, od specyfikacji mniej konstruktywnych do bardziej konstruktywnych. Natomiast w przypadku podejścia funkcyjnego mamy wyraźnie rozgraniczone trzy poziomy opisu modułów. Na potrzeby tej pracy jest to niewątpliwie zaleta podejścia funkcyjnego, gdyż pozwala nam ona na oddzielenie od siebie różnych aspektów specyfikacji wskaźnikowych struktur danych, przez co nasze rozważania zyskują na przejrzystości.

2.11 Realizacja celu pracy w ramach podejścia funkcyjnego

Tematem tej pracy jest specyfikacja wskaźnikowych struktur danych. W ramach podejścia funkcyjnego specyfikacje wskaźnikowych struktur danych występują w projektach implementacji. Problem specyfikacji wskaźnikowych struktur danych sprowadza się do opisywania warunków dotyczących stanu wskaźnikowych struktur danych (np. niezmiennika struktury

danych, warunków wstępnych wywołania operacji) oraz opisywania zmian stanu wskaźnikowych struktur danych (wyników działania operacji). W obydwu przypadkach opisujemy warunek charakteryzujący pewną klasę algebr częściowych — odpowiednio nad sygnaturą postaci $Sig(Int_t)$ lub $\Delta Sig(Int_t)$. Naszym celem jest znalezienie odpowiedniego formalizmu do opisu tych warunków. Formalizm taki powienien być wystarczająco silny, aby wyspecyfikować stosowane wskaźnikowe struktury danych, a jednocześnie specyfikacje te powinny być w miarę możliwości czytelne i zwięzłe. Kwestią tą zajmujemy się w rozdziale 4, gdzie porównujemy przydatność kilku wybranych formalizmów. Przedstawiamy tam też propozycję własnej logiki służącej do specyfikacji wskaźnikowych struktur danych. Przydatność rozważanych formalizmów porównujemy na podstawie możliwości wyspecyfikowania przykładowych, stosowanych wskaźnikowych struktur danych, oraz postaci ich specyfikacji.

Ze specyfikacją jest związana kwestia jej weryfikacji. W podejściu funkcyjnym występują dwa kroki weryfikacji: weryfikacji projektu implementacji względem specyfikacji interfejsu oraz weryfikacji implementacji względem projektu implementacji. Aby weryfikować projekty implementacji opisujące wskaźnikowe struktury danych należy znaleźć dogodną metodę definiowania funkcji abstrakcji dla wskaźnikowych struktur danych. Problem ten, jak to już było wspomniane, wykracza poza ramy tej pracy i będzie przedmiotem przyszłych badań. Zajmujemy się natomiast kwestią weryfikacji implementacji operującej na wskaźnikowych strukturach danych względem projektu implementacji. Problem ten sprowadza się do pokazania, że określone fragmenty programu (implementacje operacji) przy zadanym warunku wstępnym (warunek wstępny operacji, niezmiennik struktury danych) gwarantują spełnienie określonego warunku końcowego (warunek końcowy operacji, niezmiennik struktury danych). Znanych jest wiele podejść do tego problemu, z czego najpopularniejsze to wspomniane już wcześniej: logika Hoare'a, transformatory predykatów Dijkstry czy logika algorytmiczna. W pracy tej, jako metodę weryfikacji implementacji, przyjmujemy logikę Hoare'a. Należy jednak zaznaczyć, że wybieramy ją jako przykładową metodę, a prezentowane rezultaty równie dobrze można by przedstawić w ramach formalizmu Dijkstry czy logiki algorytmicznej.

Potencjalne zastosowanie logiki algorytmicznej nie jest ograniczone tylko do weryfikacji implementacji względem projektu implementacji. Możemy sobie wyobrazić zastosowanie logiki algorytmicznej do opisania niezmiennika struktury danych oraz operacji udostępnianych przez moduł. Podejście takie jest analizowane w innej rozprawie doktorskiej [Bał] i nie rozważamy go tutaj.

Naszym celem jest więc również zaproponowanie wariantu logiki Hoare'a, dostosowanego do weryfikacji programów operujących na wskaźnikowych strukturach danych, względem warunków wyrażonych w formalizmie wybranym do specyfikowania wskaźnikowych struktur danych. Problemem tym zajmujemy się w rozdziale 5.

Rozdział 3

Wprowadzenie do wybranych formalizmów

W rozdziale tym przedstawiono po krótce kilka formalizmów wykorzystywanych w dalszej części pracy.

3.1 Logika monadyczna drugiego rzędu

Logika monadyczna drugiego rzędu (w skrócie *MSOL*, ang. *monadic second order logic*) stanowi ograniczenie *SOL* — zmienne drugiego rzędu mogą reprezentować wyłącznie relacje jednoargumentowe. Inaczej mówiąc, zmienne pojawiające się pod kwantyfikatorami mogą reprezentować tylko wartości danego rodzaju lub zbiory wartości danego rodzaju. Tak więc każda formuła *MSOL* jest również formułą *SOL*. Semantyka formuł *MSOL* jest taka sama jak w przypadku *SOL*. Na *MSOL* można również spojrzeć jak na rozszerzenie *FOL* o możliwość wprowadzania jednoargumentowych zmiennych drugiego rzędu.

Dla lepszej czytelności zmienne pierwszego rzędu oznaczamy małymi literami, a zmienne drugiego rzędu wielkimi literami. Przykładowo, formuła $\forall_{x:t}\exists_{S:t}S(x)$ oznacza, że dla każdej wartości (rodzaju t) istnieje zbiór wartości (tego rodzaju) zawierający tę wartość.

Niech $\Sigma = \langle S_\Sigma, F_\Sigma\rangle$ będzie sygnaturą, X będzie zbiorem zmiennych pierwszego rzędu, a Y będzie zbiorem zmiennych drugiego rzędu. Przez $MSOL_\Sigma(X,Y)$ oznaczamy zbiór formuł *MSOL* nad sygnaturą Σ o zmiennych wolnych pierwszego i drugiego rzędu, odpowiednio, ze zbiorów X i Y.

Bardziej szczegółowe omówienie logiki monadycznej drugiego rzędu można znaleźć np. w [Bas96, Cou90, Tho97].

3.2 Logika stałopunktowa

Logika stałopunktowa, w skrócie *FPL* (ang. *fixed-point logic*), stanowi rozszerzenie *FOL*, choć można na nią też spojrzeć jak na ograniczenie *SOL*. Niech Σ będzie ustaloną sygnaturą, A będzie ustaloną algebrą (częściową), a $\varphi(x,X)$ będzie formułą zawierającą dwie zmienne wolne: $x : s$ pierwszego rzędu i $X : s$ drugiego rzędu. Możemy wówczas spojrzeć na formułę φ jak na funkcję $\varphi : \mathcal{P}(|A|_s) \to \mathcal{P}(|A|_s)$ — dla ustalonego zbioru $X^A \subseteq |A|_s$ formuła $\varphi(x)$ jest predykatem charakterystycznym pewnego podzbioru zbioru $|A|_s$. Dodatkowo, jeżeli zmienna X występuje w formule φ zawsze pod parzystą liczbą negacji, to formule φ odpowiada funkcja

monotoniczna (w sensie $\subseteq$). Wówczas możemy mówić o najmniejszym i największym punkcie stałym takiej funkcji. Powyższą koncepcję możemy rozszerzyć na produkty kartezjańskie nośników rodzai w A.

Logika stałopunktowa stanowi rozszerzenie logiki pierwszego rzędu o operatory najmniejszego i największego punktu stałego (w powyższym rozumieniu). Szersze omówienie logik stałopunktowych można znaleźć np. w [EF95].

Ujmując to bardziej formalnie, niech $\Sigma = \langle S_\Sigma, F_\Sigma \rangle$ będzie ustaloną sygnaturą, V będzie zbiorem zmiennych pierwszego rzędu, $W = \langle W_w \rangle_{w \in S_\Sigma^+}$ będzie zbiorem zmiennych drugiego rzędu indeksowanym ich arnościami. Zmienne pierwszego rzędu oznaczamy małymi literami x, y, ... , a zmienne drugiego rzędu oznaczamy wielkimi literami X, Y, ... Zbiór formuł logiki stałopunktowej definiujemy następująco.

Def. 17 Zbiór formuł stałopunktowych $FPL_\Sigma(V, W)$ jest to najmniejszy taki zbiór, że:

- jeśli $s_1 \in S_\Sigma$, $t_1, t_2 \in T_\Sigma(V)_{s_1}$, to $t_1 = t_2 \in FPL_\Sigma(V, W)$,
- jeśli $\varphi_1, \varphi_2 \in FPL_\Sigma(V, W)$, to: $\varphi_1 \Rightarrow \varphi_2, \neg\varphi_1 \in FPL_\Sigma(V, W)$,
- jeśli $s_1, \ldots, s_k \in S_\Sigma$, $X \in W_{\langle s_1, \ldots, s_k \rangle}$, $t_1 \in T_\Sigma(V)_{s_1}, \ldots t_k \in T_\Sigma(V)_{s_k}$, to
$$X(t_1, \ldots, t_k) \in FPL_\Sigma(V, W) \ ,$$
- jeśli $s_1 \in S_\Sigma$, x jest identyfikatorem, oraz $\varphi \in FPL_\Sigma(V \cup \{x : s_1\}, W)$, to
$$\forall_{x:s_1}(\varphi), \exists_{x:s_1}(\varphi) \in FPL_\Sigma(V, W) \ ,$$
- jeśli $s_1, \ldots, s_k \in S_\Sigma$, $x_1, \ldots, x_k$ i X są różnymi identyfikatorami, $\varphi \in FPL_\Sigma(V \cup \{x_1 : s_1, \ldots, x_k : s_k\}, W \cup \{X : s_1 \times \cdots \times s_k\})$, zmienna X występuje w φ pod parzystą liczbą negacji, oraz $t_1 \in T_\Sigma(V)_{s_1}, \ldots, t_k \in T_\Sigma(V)_{s_k}$, to
$$\mu_{x_1:s_1,\ldots,x_k:s_k,X}(\varphi)(t_1, \ldots, t_k),\ \nu_{x_1:s_1,\ldots,x_k:s_k,X}(\varphi)(t_1, \ldots, t_k) \in FPL_\Sigma(V, W) \ .$$

□

Niech A będzie częściową Σ-algebrą, $s \in S_\Sigma$, $v : V \to |A|$ będzie wartościowaniem zmiennych pierwszego rzędu, a $w : W \to \mathcal{P}(|A|^*)$ będzie wartościowaniem zmiennych drugiego rzędu, tzn. jeśli $X \in W_{s_1,\ldots,s_k}$, to $w(X) \subseteq |A|_{s_1} \times \cdots \times |A|_{s_k}$. Semantykę formuł stałopunktowych definiujemy następująco.

Def. 18 Niech $\varphi \in FPL_\Sigma(V, W)$. Przez $A \models_{v,w} \varphi$ oznaczamy fakt, iż φ jest prawdziwe w A dla wartościowań v i w. Prawdziwość formuł definiujemy następująco:

- $A \models_{v,w} t_1 = t_2$ wtw., gdy $v^A(t_1) = v^A(t_2)$,
- $A \models_{v,w} \varphi_1 \Rightarrow \varphi_2$ wtw., gdy jeżeli $A \models_{v,w} \varphi_1$, to $A \models_{v,w} \varphi_2$,
- $A \models_{v,w} \neg\varphi_1$ wtw., gdy $A \not\models_{v,w} \varphi_1$,
- $A \models_{v,w} X(t_1, \ldots, t_k)$ wtw., gdy $\langle v^A(t_1), \ldots, v^A(t_k) \rangle \in w(X)$,
- $A \models_{v,w} \forall_{x:s}(\varphi)$ ($A \models_{v,w} \exists_{x:s}(\varphi)$) wtw., gdy $A \models_{v[x \to a],w} \varphi$ dla każdego (dla pewnego) $a \in |A|_s$,

- $A \models_{v,w} \mu_{x_1:s_1,\ldots,x_k:s_k,X}(\varphi)(t_1,\ldots,t_k)$ ($A \models_{v,w} \nu_{x_1:s_1,\ldots,x_k:s_k,X}(\varphi)(t_1,\ldots,t_k)$) wtw., gdy dla funkcji $f : \mathcal{P}(|A|_{s_1} \times \cdots \times |A|_{s_k}) \to \mathcal{P}(|A|_{s_1} \times \cdots \times |A|_{s_k})$ takiej, że

$$f(R) = \left\{\langle a_1,\ldots,a_k\rangle | A \models_{v[x_1\to a_1]\ldots[x_k\to a_k],w[X\to R]} \varphi\right\} \quad ,$$

oraz zbioru

$$Q = \bigcap \{R \subseteq |A|_{s_1} \times \cdots \times |A|_{s_k} \mid f(R) \subseteq R\}$$

$$\left(Q = \bigcup \{R \subseteq |A|_{s_1} \times \cdots \times |A|_{s_k} | f(R) \supseteq R\}\right)$$

mamy $\langle v^A(t_1),\ldots,v^A(t_k)\rangle \in Q$.

□

Operatory boolowskie $\wedge$, $\vee$ i $\Leftrightarrow$ definiujemy w standardowy sposób za pomocą $\Rightarrow$ i $\neg$.

Zauważmy, że zbiór Q w powyższej definicji jest najmniejszym (największym) punktem stałym funkcji f. Zauważmy też, że zbiór ten można scharakteryzować za pomocą formuły *SOL*. Wynika stąd, że *FPL* ma co najwyżej taką siłę wyrazu jak *SOL*. W rzeczywistości jednak jej siła wyrazu jest mniejsza, gdyż np. za pomocą formuły *SOL* można wyrazić fakt, że nośnik (danego rodzaju) ma parzystą lub nieskończoną liczbę elementów, a nie da się tego faktu wyrazić w *FPL* [EF95].

Operatory stałopunktowe pozwalają np. w prosty sposób wyrazić grafowe własności struktur danych.

Przykład 4 Rozważmy listy jednokierunkowe (por. przykład 2). Niech r będzie pierwszym rekordem pewnej listy. Poniższa formuła wyraża, że na tej liście występuje rekord zawierający wartość e:

$$\exists_{z:\mathtt{rec}}\,(\mu_{x:\mathtt{rec},X}(x = r \vee X(x) \vee \exists_{y:\mathtt{rec}}(X(y) \wedge y.\mathtt{nast}\uparrow = x))(z) \wedge z.\mathtt{d} = e) \quad .$$

X reprezentuje tu zbiór wartości rekordów tworzących daną listę. □

3.3 Gramatyki podmiany hiperkrawędzi

Gramatyki podmiany hiperkrawędzi, w skrócie *HRG* (ang. *hyper-edge replacement grammar*), są ciekawą metodą definiowania zbiorów hipergrafów, będących uogólnieniem grafów. Szersze omówienie tej tematyki można znaleźć np. w [Cou90, Eng94].

Grafy skierowane składają się z wierzchołków oraz krawędzi będących uporządkowanymi parami wierzchołków. Hipergrafy różnią się tym od grafów skierowanych, że hiperkrawędź jest n-tką wierzchołków. Dodatkowo hiperkrawędzie są etykietowane, przy czym etykieta hiperkrawędzi określa jej liczbę wierzchołków.

Def. 19 Niech B będzie alfabetem ważonym, $rank_B : B \to \mathbb{N}$ będzie funkcją przyporządkowującą elementom B ich wagi. Hipergrafem H nad alfabetem B nazywamy czwórkę $H = \langle V_H, E_H, lab_H, vert_H\rangle$, gdzie V_H jest zbiorem wierzchołków, E_H jest zbiorem hiperkrawędzi, $lab_H : E_H \to B$ jest funkcją przyporządkowującą krawędziom ich etykiety, $vert_H : E_H \to V_H^*$ jest funkcją przyporządkowującą krawędziom ich wierzchołki, przy czym dla każdego $e \in E_H$

$vert_H(e)$ ma długość $rank_B(lab_H(e))$. Liczbę wierzchołków hiperkrawędzi nazywamy jej rzędem.

Hipergrafem nad alfabetem B z uchwytem rzędu n nazywamy parę $\langle H, h\rangle$, gdzie H jest hipergrafem (nad alfabetem B), a $h \in V_H^n$. Czasami hipergrafy bez uchwytu traktujemy tak, jakby miały uchwyt rzędu 0.

W dalszej części pracy izomorficzne hipergrafy utożsamiamy ze sobą. □

Grafy skierowane (z etykietowanymi krawędziami) możemy sobie przedstawić jako szczególny przypadek hipergrafów, gdy wszystkie krawędzie są rzędu 2. Podobnie, grafy nieskierowane (z etykietowanymi krawędziami) możemy sobie przedstawić jako szczególny przypadek hipergrafów, gdy wszystkie krawędzie są rzędu 2 i zaniedbamy kolejność wierzchołków tworzących krawędź (tzn. $v - w$ wtw., gdy $v \to w$ lub $w \to v$). Drzewami nieukorzenionymi będziemy nazywali spójne, acykliczne grafy nieskierowane.

Na hipergrafach z uchwytami określamy operacje pozwalające nam budować bardziej skomplikowane hipergrafy z prostszych.

Def. 20 Niech B będzie ustalonym alfabetem ważonym. Rozważmy następującą (nieskończoną) sygnaturę algebry wielosortowej $\mathbb{G}\Sigma = \langle \mathbb{N}, F_{\mathbb{G}\Sigma}\rangle$, gdzie $F_{\mathbb{G}\Sigma}$ zawiera następujące symbole:

- $\_ \oplus_{n,m} \_ : n \times m \to n + m$, dla $n, m \in \mathbb{N}$,
- $\theta_{\delta,n} : n \to n$, dla $n \in \mathbb{N}$ i relacji równoważności δ na zbiorze $\{1, \ldots, n\}$,
- $\sigma_{\alpha,p,n} : n \to p$, dla $p, n \in \mathbb{N}$ i funkcji $\alpha : \{1, \ldots, p\} \to \{1, \ldots, n\}$,
- $b : rank(b)$, dla $b \in B$,
- $\mathbf{0} : 0$ i $\mathbf{1} : 1$.

Oznaczmy przez $\mathbb{G}(B)$ następującą (całkowitą) $\mathbb{G}\Sigma$-algebrę:

- elementami $|\mathbb{G}(B)|_n$ są skończone hipergrafy (nad alfabetem B) z uchwytami rzędu n,
- $\langle H_1, \langle h_1, \ldots, h_n\rangle\rangle \oplus_{n,m}^{\mathbb{G}(B)} \langle H_2, \langle h_{n+1}, \ldots, h_{n+m}\rangle\rangle = \langle H, \langle h_1, \ldots, h_{n+m}\rangle\rangle$, gdzie H jest sumą (rozłączną) H_1 i H_2,
- $\theta_{\delta,n}^{\mathbb{G}(B)}(\langle H_1, \langle h_1, \ldots, h_n\rangle\rangle) = \langle H, \langle [h_1]_\delta, \ldots, [h_n]_\delta\rangle\rangle$, gdzie H powstaje z H_1 przez sklejenie wszystkich takich wierzchołków h_i i h_j, dla których $\langle i, j\rangle \in \delta$,
- $\sigma_{\alpha,p,n}^{\mathbb{G}(B)}(\langle H_1, \langle h_1, \ldots, h_n\rangle\rangle) = \langle H_1, \langle h_{\alpha(1)}, \ldots, h_{\alpha(p)}\rangle\rangle$,
- $b^{\mathbb{G}(B)}$ jest hipergrafem zawierającym $rank_B(b)$ wierzchołków i jedną krawędź e o etykiecie $lab_b(e) = b$ złożoną ze wszystkich wierzchołków, z uchwytem postaci $vert_b(e)$,
- $\mathbf{0}^{\mathbb{G}(B)}$ jest pustym hipergrafem, a $\mathbf{1}^{\mathbb{G}(B)}$ jest hipergrafem złożonym tylko z jednego wierzchołka będącego równocześnie jego uchwytem.

Termy bez zmiennych wolnych nad sygnaturą $\mathbb{G}\Sigma(B)$ nazywamy wyrażeniami grafowymi. □

Fakt 1 [Cou90] Dla każdego skończonego hipergrafu H (z uchwytem) istnieje takie wyrażenie grafowe $t \in T_{\mathbb{G}\Sigma(B)}(\emptyset)$, że $t^{\mathbb{G}(B)} = H$. □

Def. 21 Szerokością $wd(t)$ wyrażenia grafowego $t \in T_{\mathbb{G}\Sigma(B)}(\emptyset)$ nazywamy maksymalny rodzaj podtermów termu t. Przez $wd(H)$ oznaczamy szerokość hipergrafu (z uchwytem) równą

$$wd(H) = \left\{ \min wd(t) | t \in T_{\mathbb{G}\Sigma(B)}(\emptyset), t^{\mathbb{G}(B)} = H \right\} \ .$$

□

Fakt 2 [Cou90] Szerokość n-elementowego grafu pełnego K_n spełnia nierówność $wd(K_n) \geq n$. □

Mając daną hiperkrawędź rzędu n oraz dany hipergraf z uchwytem rzędu n możemy zastąpić tę krawędź danym hipergrafem. Operację taką nazywamy podmianą hiperkrawędzi.

Def. 22 Niech H_1 będzie hipergrafem (nad alfabetem B), e hiperkrawędzią rzędu n należącą do H_1, H_2 będzie hipergrafem z uchwytem h rzędu n. Przez $H_1[e \to H_2]$ oznaczamy hipergraf H_3 powstały w następujący sposób:

- niech H_4 będzie hipergrafem powstałym z H_1 przez usunięcie hiperkrawędzi e,
- H_3 powstaje z sumy rozłącznej H_4 i H_2 poprzez sklejenie wierzchołków tworzących e z odpowiadającymi im wierzchołkami uchwytu h.

Hipergraf $H_1[e \to H_2]$ nazywamy wynikiem podmiany hiperkrawędzi e przez hipergraf H_2.

Operację podmiany hiperkrawędzi rozszerzamy w naturalny sposób na hipergrafy z uchwytami:

$$\langle H_1, h \rangle [e \to H_2] = \langle H_1[e \to H_2], h \rangle \ .$$

□

Podobnie do gramatyk bezkontekstowych na słowach można zdefiniować gramatyki podmiany hiperkrawędzi — zastąpieniu symbolu nieterminalnego przez jego rozwinięcie odpowiada podmiana hiperkrawędzi.

Def. 23 Gramatyką podmiany hiperkrawędzi nazywamy czwórkę postaci $\Gamma = \langle B, U, P, Z \rangle$, gdzie B i U są rozłącznymi, skończonymi alfabetami ważonymi, elementy B nazywamy symbolami terminalnymi, a U nieterminalnymi, P jest skończonym zbiorem produkcji postaci $u \to D$, gdzie $u \in U$, a $D \in |\mathbb{G}(B \cup U)|_{rank_U(u)}$, oraz $Z \in U$ jest nazywane aksjomatem.

Niech $k \in \mathbb{N}$, $H_1, H_2 \in |\mathbb{G}(B \cup U)|_k$. Przez $H_1 \to_P H_2$ oznaczamy fakt, że w H_1 istnieje taka hiperkrawędź e, oraz w P mamy taką produkcję $u \to D$, że $lab(e) = u$ oraz $H_1[e \to D] = H_2$. Przez $L(\Gamma, H_1)$ oznaczamy zbiór hipergrafów, nad alfabetem B, wyprowadzalnych z H w gramatyce Γ:

$$L(\Gamma, H_1) = \{ H \in |\mathbb{G}(B)|_0 \ | H_1 \to_P^* H \} \ .$$

Zbiór hipergrafów nad alfabetem B generowany przez gramatykę Γ oznaczamy przez $L(\Gamma) = L(\Gamma, Z)$. Zbiór hipergrafów nad danym alfabetem B nazywamy bezkontekstowym, jeżeli jest on generowany przez pewną gramatykę *HRG*. □

Gramatyki *HRG* możemy też określić w sposób alternatywny za pomocą wyrażeń hipergrafowych. Zauważmy, że każdy hipergraf z uchwytem, występujący po prawej stronie produkcji możemy przedstawić jako wyrażenie hipergrafowe (nad alfabetem $B \cup U$). W ten sposób uzyskujemy gramatykę bezkontekstową (klasyczną) generującą pewien zbiór wyrażeń hipergrafowych (nad alfabetem B). Wartościami tych wyrażeń są hipergrafy generowane przez daną gramatykę podmiany hiperkrawędzi. Konsekwencją takiego spojrzenia na gramatyki *HRG* jest następujący fakt.

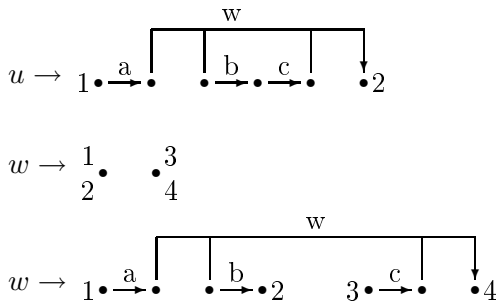
Fakt 3 [Cou90] Jeśli L jest bezkontekstowym zbiorem hipergrafów, to szerokość $wd(L) = \max\{wd(H)|H \in L\}$ jest skończona. □

Fakt ten dostarcza nam prostego wstępnego kryterium sprawdzającego, czy dany zbiór hipergrafów może być opisany za pomocą *HRG*.

Gramatyki podmiany hiperkrawędzi są również przydatne do opisywania zwykłych grafów — hiperkrawędzie rzędu innego niż 2 występują wówczas jedynie jako symbole nieterminalne. Ilustruje to następujący przykład.

Przykład 5 [Cou90] Słowo nad alfabetem A możemy sobie przedstawić jako graf będący ścieżką prostą — kolejne litery tworzące słowo są etykietami kolejnych krawędzi.

Okazuje się, że zbiór grafów reprezentujących słowa postaci $a^n b^n c^n$ (dla $n \geq 1$) jest generowany przez następującą gramatykę. Alfabet terminalny zawiera symbole a, b i c — wszystkie rzędu 2. Alfabet nieterminalny zawiera dwa symbole: w rzędu 4 i aksjomat u rzędu 2. Produkcje przedstawiono na poniższym rysunku. Wierzchołki tworzące uchwyty zaznaczono kolejnymi liczbami.



□

W przypadku grafów skierowanych, oprócz szerokości grafu, możemy również mówić o drzewiastej szerokości grafu.

Def. 24 Pokryciem drzewiastym grafu G o zbiorze wierzchołków V_G nazywamy parę $\langle T, f \rangle$, gdzie T jest nieskierowanym, nieukorzenionym drzewem o zbiorze wierzchołków V_T, a f jest taką funkcją $f : V_T \to \mathcal{P}(V_G)$, że:

- $V_G = \bigcup_{i \in V_T} f(i)$,
- każda krawędź G ma obydwa końce w pewnym zbiorze $f(i)$,
- jeżeli $v \in f(i) \cap f(j)$, to $v \in f(k)$ dla każdego k należącego do jedynej ścieżki prostej łączącej i i j w drzewie T.

Szerokością pokrycia drzewiastego nazywamy liczbę $\max_{i \in V_T} |f(i)| - 1$. Szerokością drzewiastą $twd(G)$ grafu G nazywamy minimalną szerokość drzewiastą jego pokrycia drzewiastego. □

Łatwo zauważyć, że drzewa mają szerokość drzewiastą 1 oraz, że $twd(G) \leq |V_G| - 1$. Dzięki następującemu faktowi można wykorzystywać szerokość drzewiastą jako wstępne kryterium na to, czy dany zbiór grafów można zdefiniować za pomocą *HRG*.

Fakt 4 [Cou90] Dla każdego skończonego grafu skierowanego G mamy:

$$twd(G) + 1 \leq wd(G) \leq 2twd(G) + 3 \ .$$

□

3.4 Modalne specyfikacje dynamicznych typów danych

Przedstawimy teraz pewną modyfikację formalizmu zaproponowanego w [CR97] do specyfikacji dynamicznych[1] typów danych. Formalizm ten oznaczamy przez *DDtL* (ang. *dynamic data-type logic*). Dynamiczne typy danych są modelowane za pomocą algebr dynamicznych, będących rozszerzeniem częściowych struktur relacyjnych[2]. Wersja formalizmu prezentowana w tym podrozdziale posłuży nam do opisu algebr częściowych, modelujących wskaźnikowe struktury danych.

Możemy spojrzeć na strukturę wskaźnikową jak na graf, którego wierzchołkami są rekordy, a krawędziami są wskaźniki łączące te rekordy. W prezentowanym podejściu (por. p. 2.5), wskaźniki/krawędzie modelujemy za pomocą jednoargumentowych funkcji częściowych. Argumentem takiej funkcji jest początek krawędzi, a wynikiem jej koniec. Zbiór krawędzi, które chcemy w danej chwili rozważać, możemy zadać za pomocą zbioru jednoargumentowych funkcji częściowych — lub bardziej ogólnie, za pomocą zbioru termów z dziurami.

Na potrzeby tego podrozdziału, niech $\Sigma = \langle S_\Sigma, F_\Sigma \rangle$ będzie sygnaturą, $X = \langle X_s \rangle_{s \in S_\Sigma}$ będzie zbiorem zmiennych (zakładamy, że $[] \notin X$).

Def. 25 Niech $s \in S_\Sigma$. Opisem krawędzi (między elementami rodzaju s, nad zbiorem zmiennych X) nazywamy dowolny skończony zbiór e termów (nad zbiorem zmiennych X) rodzaju s, z dziurą $[] : s$, $e \subseteq T_\Sigma(X \cup \{[] : s\})_s$. Zbiór wszystkich opisów krawędzi między elementami rodzaju s, nad zbiorem zmiennych X, oznaczamy przez $E_\Sigma(X, s)$. □

Def. 26 Niech A będzie częściową Σ-algebrą, $s \in S_\Sigma$, $v : X \to |A|$ będzie wartościowaniem, oraz $e \in E_\Sigma(X, s)$. Przez $Path(A, s, v, e)$ oznaczymy zbiór (skończonych i nieskończonych) ciągów $p = \langle p_0, p_1, \ldots \rangle \in |A|_s^+ \cup |A|_s^\omega$ (nazywanych ścieżkami) spełniających następujące własności:

- dla każdego $0 \leq k \leq length(p) - 2$ istnieje takie $t \in e$, że $(v[[] \to p_k])^A(t) = p_{k+1}$, oraz
- jeśli $p = \langle p_0, \ldots, p_k \rangle$ to dla każdego $t \in e$, $(v[[] \to p_k])^A(t)$ jest nieokreślone.

□

[1] Określenia „dynamiczne typy danych" używa się czasem w odniesieniu do wskaźnikowych struktur danych. Tutaj słowo „dynamicznych" pochodzi od zdolności do zmieniania się i nie ma nic wspólnego ze wskaźnikami.

[2] Struktury relacyjne są też nazywane algebrami z predykatami. Stąd też przyjął się termin „algebry dynamiczne", a nie „dynamiczne struktury relacyjne".

Rozróżniamy dwa rodzaje formuł: formuły dynamiczne oraz formuły ścieżkowe, przy czym formuły ścieżkowe są tylko składnikiem służącym do budowy formuł dynamicznych.

Def. 27 Niech $s \in S_\Sigma$. Zbiór formuł dynamicznych $D_\Sigma(X)$ oraz zbiór formuł ścieżkowych $P_\Sigma(X, s)$ to najmniejsze takie zbiory, że:

- jeśli $s_1 \in S_\Sigma$, $t_1, t_2 \in T_\Sigma(X)_{s_1}$, to $t_1 = t_2 \in D_\Sigma(X)$,
- jeśli $\varphi_1, \varphi_2 \in D_\Sigma(X)$, to: $\varphi_1 \Rightarrow \varphi_2, \neg\varphi_1 \in D_\Sigma(X)$,
- jeśli $s_1 \in S_\Sigma$, x jest identyfikatorem oraz $\varphi \in D_\Sigma(X \cup \{x : s_1\})$, to
$$\forall_{x:s_1}(\varphi), \exists_{x:s_1}(\varphi) \in D_\Sigma(X) \ ,$$
- jeśli $t \in T_\Sigma(X)_s$, $e = \{e_1, \ldots, e_k\} \in E_\Sigma(X, s)$ oraz $\pi \in P_\Sigma(X, s)$, to
$$\triangle_{t,e_1,\ldots,e_k}\pi, \triangledown_{t,e_1,\ldots,e_k}\pi \in D_\Sigma(X) \ ,$$
- jeśli $\pi_1, \pi_2 \in P_\Sigma(X, s)$, to: $\pi_1 \Rightarrow \pi_2, \neg\pi_1 \in P_\Sigma(X, s)$,
- jeśli $s_1 \in S_\Sigma$, x jest identyfikatorem, $\pi \in P_\Sigma(X \cup \{x : s_1\}, s)$, to
$$\forall_{x:s_1}(\pi), \exists_{x:s_1}(\pi) \in P_\Sigma(X, s) \ ,$$
- jeśli $\varphi \in D_\Sigma(X \cup \{x : s\})$, to $[\lambda x.\varphi] \in P_\Sigma(X, s)$,
- jeśli $\pi_1, \pi_2 \in P_\Sigma(X, s)$, to: $\pi_1 \mathcal{U} \pi_2, \in P_\Sigma(X, s)$.

□

Formuły dynamiczne są interpretowane przy zadanym wartościowaniu zmiennych, a formuły ścieżkowe przy zadanym wartościowaniu oraz ścieżce. Zarówno formuły dynamiczne jak i ścieżkowe można budować używając operatorów pierwszego rzędu. Formuła dynamiczna $\triangle_{t,e_1,\ldots,e_k}$ reprezentuje uniwersalną, a $\triangledown_{t,e_1,\ldots,e_k}$ reprezentuje egzystencjalną kwantyfikację ścieżek o początku w t i krawędziach opisanych przez $e_1, \ldots, e_k$. Odpowiadają one modalnościom „konieczne" i „możliwe". W formule ścieżkowej $[\lambda x.\varphi]$ zmienna x przyjmuje wartość pierwszego elementu na ścieżce, dla której jest interpretowana formuła. Operator temporalny $\mathcal{U}$ „*until*" jest interpretowany wzdłuż danej ścieżki.

Def. 28 Niech A będzie częściową Σ-algebrą, $v : X \to |A|$ będzie wartościowaniem, $s \in S_\Sigma$, $\{e_1, \ldots, e_k\} \in E_\Sigma(X, s)$, $p \in Path(A, s, v, \{e_1, \ldots, e_k\})$, $\varphi \in D_\Sigma(X)$, oraz $\pi \in P_\Sigma(X, s)$. Poniżej definiujemy, wzajemnie rekurencyjnie, kiedy formuła dynamiczna φ jest spełniona w A przy wartościowaniu v (co oznaczamy przez $A \models_v \varphi$), oraz kiedy formuła ścieżkowa π jest spełniona w A przy wartościowaniu v dla ścieżki p (co oznaczamy przez $A, p \models_v \pi$).

- $A \models_v t_1 = t_2$ wtw., gdy $v^A(t_1) = v^A(t_2)$,
- $A \models_v \varphi_1 \Rightarrow \varphi_2$ wtw., gdy jeżeli $A \models_v \varphi_1$, to $A \models_v \varphi_2$,
- $A \models_v \neg\varphi_1$ wtw., gdy $A \not\models_v \varphi_1$,
- $A \models_v \forall_{x:s_1}(\varphi)$ ($A \models_v \exists_{x:s_1}(\varphi)$) wtw., gdy $A \models_{v[x\to a]} \varphi$ dla każdego (dla pewnego) $a \in |A|_{s_1}$,

- $A \models_v \triangle_{t,e_1,\ldots,e_k}\pi$ ($A \models_v \triangledown_{t,e_1,\ldots,e_k}\pi$) wtw., gdy $v^A(t)$ jest określone oraz dla każdej (dla pewnej) ścieżki $p \in Path(A, s, v, \{e_1, \ldots, e_k\})$ (gdzie s jest rodzajem t) o początku $B(p) = v^A(t)$ mamy $A, p \models_v \pi$,
- $A, p \models_v \pi_1 \Rightarrow \pi_2$ wtw., gdy jeżeli $A, p \models_v \pi_1$, to $A, p \models_v \pi_2$,
- $A, p \models_v \neg\pi_1$ wtw., gdy $A, p \not\models_v \pi_1$,
- $A, p \models_v \forall_{x:s_1}(\pi)$ ($A, p \models_v \exists_{x:s_1}(\pi)$) wtw., gdy $A, p \models_{v[x\to a]} \pi$ dla każdego (dla pewnego) $a \in |A|_{s_1}$,
- $A, p \models_v [\lambda x.\varphi]$ wtw., gdy $A \models_{v[x\to B(p)]]} \varphi$,
- $A, p \models_v \pi_1 \mathcal{U} \pi_2$ wtw., gdy istnieje takie $j > 0$, że $p_{|j}$ jest określone, $A, p_{|j} \models_v \pi_2$, oraz dla każdego $0 < i < j$, $A, p_{|i} \models_v \pi_1$.

Jeśli $A \models_v \varphi$ dla każdego wartościowania $v : X \to |A|$, to zapisujemy $A \models \varphi$. □

Operatory boolowskie $\wedge$, $\vee$ i $\Leftrightarrow$ definiujemy w standardowy sposób za pomocą $\Rightarrow$ i $\neg$. Podobnie, za pomocą $\mathcal{U}$ definiujemy operatory ścieżkowe $\Box$ (wszędzie na ścieżce), $\Diamond$ (gdzieś na ścieżce) oraz $\mathcal{O}$ (począwszy od następnego wierzchołka ścieżki).

Rozdział 4

Specyfikacja wskaźnikowych struktur danych

W rozdziale tym badamy przydatność kilku wybranych formalizmów do specyfikowania wskaźnikowych struktur danych. Wiele metod specyfikacji opiera się na logice pierwszego rzędu. Naturalne jest więc pytanie o jej przydatność do specyfikowania wskaźnikowych struktur danych. Niestety okazuje się, że *FOL* ma poważne ograniczenia. Nie można w niej wyrazić np. takiej własności jak istnienie ścieżki (złożonej ze wskaźników) łączącej dwa dane rekordy (por. [Cou90, Tho97]). Uniemożliwia to wyspecyfikowanie praktycznie wszystkich stosowanych wskaźnikowych struktur danych. Dlatego też rozważamy dalej formalizmy silniejsze niż (lub nieporównywalne z) *FOL*.

Z drugiej strony, wszystkie takie własności jak istnienie ścieżki łączącej dwa dane rekordy, acykliczność, skończoność struktury, porównanie długości ścieżek czy nawet izomorficzność dwóch struktur są wyrażalne w logice drugiego rzędu. Wydaje się więc, że w *SOL* można wyrazić wszystkie interesujące nas własności wskaźnikowych struktur danych. Naszym celem jest jednak wybór formalizmu, w którym specyfikacje mogą być zapisane w sposób względnie czytelny i zwięzły oraz zbadanie, czy do specyfikowania wskaźnikowych struktur danych nie wystarczyłby formalizm słabszy od *SOL*. Dlatego też głównymi kandydatami do naszych badań są różne formalizmy plasujące się pod względem siły wyrazu pomiędzy logiką pierwszego a drugiego rzędu.

Badane formalizmy porównujemy na podstawie zapisanych w nich specyfikacji przykładowych struktur danych, bądź braku możliwości wyrażenia takich specyfikacji. Oczywiście nie jesteśmy w stanie wyspecyfikować tu wszystkich stosowanych struktur wskaźnikowych. Ze względu na ograniczoną objętość niniejszej pracy nie jesteśmy nawet w stanie wyspecyfikować wszystkich struktur pojawiających się w trakcie typowego kursu nt. algorytmów i struktur danych. Ograniczamy się do kilku przykładowych, najpopularniejszych struktur wybranych tak, aby reprezentowały najistotniejsze aspekty specyfikacji wskaźnikowych struktur danych. Są to:

- listy jednokierunkowe,
- cykliczne listy dwukierunkowe,
- drzewa binarne,
- drzewa BST,

- drzewa AVL,
- kopce binarne,
- drzewa *find-union* oraz
- grafy skierowane.

Listy jednokierunkowe są najprostszą a zarazem powszechnie stosowaną wskaźnikową strukturą danych. Często też występują one jako składowe bardziej złożonych struktur. Cykliczne listy dwukierunkowe to przykład struktury listowej, w której różne wskaźniki są ze sobą powiązane — lista musi mieć kształt cyklu, a wskaźniki prowadzące w przeciwnych kierunkach muszą być ze sobą skorelowane. Drzewa binarne, BST, AVL i kopce binarne to reprezentanci bogatej klasy struktur drzewiastych. Specyfikacje drzew binarnych sprawdzają możliwość opisania kształtu drzewiastej struktury danych. Drzewa BST i kopce binarne są przykładem drzew z nałożonymi ograniczeniami na wartości przechowywane w ich węzłach. Drzewa AVL to przykład struktury drzewiastej, w której ograniczenia nałożone na możliwe kształty drzew zapewniają ich zrównoważenie. Drzewa *find-union* nie są typowymi drzewami. Jest to przykład struktury mającej kształt drzewa, w której jednak wskaźniki są zorientowane w kierunku od potomka do przodka. Grafy skierowane są przykładem struktury danych o stosunkowo największym bogactwie możliwych kształtów.

Jak widać z rozdziału 2, problem specyfikacji wskaźnikowych struktur danych sprowadza się do wyspecyfikowania, dla zadanego $\overline{A}$, zbioru zmiennych X i wartościowania $v : X \to |\overline{A}|$, pewnego podzbioru $\overline{A}$. (Przypomnijmy, że $\overline{A}$ jest zbiorem algebr częściowych różniących się od A wyłącznie interpretacją symboli funkcyjnych postaci $\uparrow$ i $\uparrow^{\delta}$, dla $\delta \in \mathcal{D}$.) W przypadku, gdy chcemy opisać niezmiennik struktury danych, to $\overline{A} \subseteq \mathit{DStr}(\mathit{Int}_t)$, a $X = \emptyset$. W przypadku, gdy chcemy opisać warunek wstępny operacji, to $\overline{A} \subseteq \mathit{DStr}(\mathit{Int}_t)$, X zawiera zmienne reprezentujące adresy argumentów operacji, a v określa, które zmienne są aktualnymi argumentami operacji. Natomiast w przypadku, gdy chcemy opisać warunek końcowy operacji, to $\overline{A} \subseteq \mathit{DStr}(\Delta\mathit{Sig}(\mathit{Int}_t))$, X zawiera zmienne reprezentujące adresy argumentów operacji, a v określa które zmienne są aktualnymi argumentami operacji.

W naszych rozważaniach ograniczamy się do najbardziej istotnych elementów specyfikacji struktur danych. Przeważnie koncentrujemy się na niezmiennikach struktur danych, jako że jest to centralny element specyfikacji. Warunki wstępne operacji są zwykle prostym wzmocnieniem niezmiennika. Warunek końcowy operacji jest bardziej skomplikowany, choć w swej naturze jest on podobny do niezmiennika. Warunek ten wiąże stany struktury danych „przed" i „po" wywołaniu operacji. Istotną jego częścią jest zwykle to, że operacja zachowuje niezmiennik struktury danych.

Mimo, że zawarta w tym rozdziale analiza jest, z natury rzeczy, ograniczona, to pozwala ona wyraźnie rozróżnić badane formalizmy pod kątem ich przydatności do specyfikacji wskaźnikowych struktur danych.

4.1 Specyfikacje wskaźnikowych struktur danych w *MSOL*

Logika monadyczna drugiego rzędu jest przykładem logiki wykorzystywanej w formalnych specyfikacjach, silniejszej od logiki pierwszego rzędu, a słabszej od logiki drugiego rzędu. Warto tu nadmienić o możliwości wykorzystania *MSOL* do specyfikowania układów elektronicznych [Bas96] oraz list i drzew [JJKS97, KS93, KS94]. Ważną cechą jest możliwość automatycznej

weryfikacji takich specyfikacji, wynikająca z rozstrzygalności spełnialności formuł *MSOL* dla list i drzew (zarówno skończonych jak i nieskończonych), por. np. [Tho90, Tho97].

Nadmieńmy, że jeżeli jakaś relacja binarna (np. połączenie rekordów za pomocą wskaźników) jest wyrażalne w *MSOL*, to jej domknięcie zwrotno-tranzytywne też jest wyrażalne w *MSOL* [Cou90]. Wynika stąd, że w *MSOL* można wyrazić takie własności, jak istnienie ścieżki (złożonej ze wskaźników) łączącej dwa rekordy struktury danych. Można też reprezentować, jako zbiory (adresów) rekordów, minimalne (w sensie zawierania się zbiorów) ścieżki łączące dwa dane rekordy. Trudność w wyrażeniu pojęcia ścieżki w ogólności wynika z konieczności opisania relacji następstwa wierzchołków tworzących ścieżkę oraz możliwości powtarzania się wierzchołków na ścieżce.

Przyjrzyjmy się kilku przykładowym specyfikacjom wskaźnikowych struktur danych. Zacznijmy od prostego przykładu list jednokierunkowych.

Przykład 6 Deklaracja struktury danych modułu implementującego listy jednokierunkowe może wyglądać następująco (por. przykład 2):

```
TYPE
  lista = POINTER TO elem;
  elem = RECORD
    d: dane;
    nast: lista
  END;
  ppelem = POINTER TO lista;
  plista = POINTER TO lista
```

Początki list można scharakteryzować następującą formułą:

$$head(p : \texttt{lista}) \mathrel{\hat{=}} (p\uparrow)\downarrow \wedge \forall_{q:\texttt{lista}}(q\uparrow.\texttt{nast} \neq p) \ .$$

Niezmiennik list jednokierunkowych może być opisany jako koniunkcja następujących własności:

- zewnętrzne wskaźniki wskazują wyłącznie na początki list

$$\forall_{p:\texttt{plista}}((p\uparrow)\downarrow \wedge p\uparrow \neq \texttt{NIL} \Rightarrow head(p\uparrow)) \ ,$$

- każda lista jest zakończona wskaźnikiem `NIL`

$$\forall_{p:\texttt{lista}} \begin{pmatrix} head(p) \Rightarrow \\ \forall_{S:\texttt{lista}} (S(p) \wedge \forall_{q:\texttt{lista}} (S(q) \wedge (q\uparrow)\downarrow \Rightarrow S(q\uparrow.\texttt{nast})) \Rightarrow S(\texttt{NIL})) \end{pmatrix} \ ,$$

- każdy alokowany rekord jest na pewnej liście

$$\forall_{p:\texttt{lista}} \left((p\uparrow)\downarrow \Rightarrow \exists_{q:\texttt{plista}} \left(\begin{pmatrix} head(q\uparrow)\wedge \\ \forall_{S:\texttt{lista}} \left(\begin{pmatrix} S(q\uparrow)\wedge \\ \forall_{r:\texttt{lista}} \begin{pmatrix} S(r) \wedge (r\uparrow)\downarrow \Rightarrow \\ S(r\uparrow.\texttt{nast}) \end{pmatrix} \Rightarrow \\ S(p) \end{pmatrix} \right) \end{pmatrix} \right) \right) \ ,$$

- do każdego węzła prowadzi co najwyżej jedna krawędź

$$\forall_{p,q,r:\texttt{lista}}\,(p\uparrow.\texttt{nast} = r \wedge q\uparrow.\texttt{nast} = r \wedge r \neq \texttt{NIL} \Rightarrow p = q) \quad .$$

□

Przykład 7 Moduł implementujący cykliczne listy dwukierunkowe może wykorzystywać następujące deklaracje typów:

```
TYPE
  lista = POINTER TO elem;
  elem = RECORD
    d: data;
    pop, nast: lista
  END;
ppelem = POINTER TO lista;
plista = POINTER TO lista;
```

Następujący niezmiennik struktury danych określa dopuszczalne kształty list oraz wyraża fakt, że każda lista jest wskazywana przez dokładnie jeden zewnętrzny wskaźnik i vice versa:

$$\forall_{p:\texttt{lista}}\left((p\uparrow)\downarrow \Rightarrow \left(\begin{array}{l} p\uparrow.\texttt{nast}\uparrow.\texttt{pop} = p \ \wedge p\uparrow.\texttt{pop}\uparrow.\texttt{nast} = p \wedge \\ \forall_{X:\texttt{lista}}\left(\left(\begin{array}{l} X(p\uparrow.\texttt{nast})\wedge \\ \forall_{q:\texttt{lista}}\,(X(q) \wedge (q\uparrow)\downarrow \Rightarrow X(q\uparrow.\texttt{nast}))\end{array}\right) \Rightarrow X(p)\right) \wedge \\ \forall_{X:\texttt{lista}}\left(\left(\begin{array}{l} X(p\uparrow.\texttt{pop})\wedge \\ \forall_{q:\texttt{lista}}\,(X(q) \wedge (q\uparrow)\downarrow \Rightarrow X(q\uparrow.\texttt{pop}))\end{array}\right) \Rightarrow X(p)\right) \wedge \\ \exists_{q:\texttt{plista}}\left(\begin{array}{l} \exists_{X:\texttt{lista}}\left(\begin{array}{l} X(q\uparrow)\wedge \\ \forall_{r:\texttt{lista}}\,(X(r) \Rightarrow X(r\uparrow.\texttt{nast}))\wedge \\ \forall_{r:\texttt{plista}}\,(r \neq q \Rightarrow \neg X(r\uparrow))\end{array}\right) \wedge \\ \forall_{X:\texttt{lista}}\left(\left(\begin{array}{l} X(q\uparrow)\wedge \\ \forall_{r:\texttt{lista}}\left(\begin{array}{l} X(r) \Rightarrow \\ X(r\uparrow.\texttt{nast})\end{array}\right)\end{array}\right) \Rightarrow X(p)\right)\end{array}\right)\end{array}\right)\right) \wedge$$
$$\forall_{p:\texttt{plista}}\,((p\uparrow)\downarrow \wedge p\uparrow \neq \texttt{NIL} \Rightarrow (p\uparrow\uparrow)\downarrow) \quad .$$

□

Przykład 8 Deklaracja drzew *find-union* (używanych do reprezentowania rozłącznych zbiorów elementów) może być bardzo prosta:

```
TYPE
  node = POINTER TO node;
```

Niezmiennik tej struktury danych wyraża fakt, że każda ścieżka, zaczynająca się w dowolnym elemencie struktury, prowadzi do elementu wskazującego na samego siebie:

$$\forall_{p,S:\texttt{node}}\,((S(p) \wedge \forall_{q:\texttt{node}}(S(q) \Rightarrow S(q\uparrow))) \Rightarrow \exists_{q:\texttt{node}}(S(q) \wedge q\uparrow = q)) \quad .$$

Poniżej opisujemy operację *find*, która dla zadanego wskaźnika p do elementu struktury danych zmienia jego wartość tak, aby wskazywał na reprezentanta zbioru, do którego należy dany element, a przy okazji dokonuje kompresji ścieżki prowadzącej do tego reprezentanta. W

warunku wstępnym tej operacji, oprócz niezmiennika danych, wymagamy, aby początkowa wartość p wskazywała na istniejący element: $(p\uparrow)\downarrow$. Warunek końcowy ma następującą postać:

$$p\text{`}\uparrow = p \wedge p\uparrow = p\wedge$$

$$\forall_{S:\texttt{node}} \left(\left(S(\text{`}p) \wedge \forall_{q:\texttt{node}} (S(q) \Rightarrow S(q\text{`}\uparrow)) \right) \Rightarrow S(p) \right) \wedge$$

$$\exists_{S:\texttt{node}} \left(S(\text{`}p) \wedge \forall_{q:\texttt{node}} (S(q) \Rightarrow S(q\text{`}\uparrow)) \wedge \forall_{q:\texttt{node}} (S(q) \Rightarrow q\uparrow = p) \right) \wedge$$

$$\forall_{q:\texttt{node}} \left(\exists_{S:\texttt{node}} \left(S(\text{`}p) \wedge \forall_{q:\texttt{node}} (S(q) \Rightarrow S(q\text{`}\uparrow)) \wedge \neg S(q) \right) \Rightarrow q\text{`}\uparrow = q\uparrow \right) \ .$$

□

Przykład 9 Korzystając z list jednokierunkowych możemy wyspecyfikować wskaźnikową implementację grafów skierowaych. Wierzchołki grafu przechowujemy na liście jednokierunkowej. Z każdym wierzchołkiem mamy związaną jego listę incydencji — jednokierunkową listę wskaźników do tych wierzchołków, do których z danego wierzchołka prowadzą krawędzie. Deklaracje typów dla takiej struktury danych mają następującą postać:

```
TYPE
  graf = POINTER TO wierzchołek;
  sąsiedzi = POINTER TO krawędź;
  wierzchołek = RECORD
    s: sąsiedzi;
    nast: graf
  END;
  krawędź = RECORD
    w: graf;
    nast: sąsiedzi
  END;
ppwierzchołek = POINTER TO graf;
pgraf = POINTER TO graf;
```

Niezmiennik takiej struktury danych powinien wyrażać następujące własności:

- wierzchołki każdego grafu tworzą poprawne listy jednokierunkowe,
- krawędzie każdej listy incydencji tworzą poprawne listy jednokierunkowe,
- każda krawędź prowadzi do wierzchołka z tego samego grafu.

Podobnie jak w przykładzie 6, oznaczmy przez $head_w$ i $head_k$ następujące formuły:

- $head_w(p : \texttt{graf}) \mathrel{\hat{=}} (p\uparrow)\downarrow \wedge \forall_{q:\texttt{graf}} (q\uparrow.\texttt{nast} \neq p)$,
- $head_k(p : \texttt{sąsiedzi}) \mathrel{\hat{=}} (p\uparrow)\downarrow \wedge \forall_{q:\texttt{sąsiedzi}} (q\uparrow.\texttt{nast} \neq p)$,

mówiące, że dany wskaźnik wskazuje na pierwszy element listy, odpowiednio, wierzchołków lub krawędzi. Niezmiennik struktury danych grafu możemy zapisać następująco:

$$\forall_{p:\texttt{pgraf}}\left((p\uparrow)\downarrow \wedge p\uparrow \neq \texttt{NIL} \Rightarrow head_w(p\uparrow)\right) \wedge$$

$$\forall_{p:\texttt{graf}}\left(head_w(p) \Rightarrow \forall_{X:\texttt{graf}}\left(X(p) \wedge \forall_{q:\texttt{graf}}\left(X(q) \wedge (q\uparrow)\downarrow \Rightarrow X(q\uparrow.\texttt{nast})\right) \Rightarrow X(\texttt{NIL})\right)\right) \wedge$$

$$\forall_{p:\texttt{graf}}\left((p\uparrow)\downarrow \Rightarrow \exists_{q:\texttt{pgraf}}\left(\begin{array}{l} head_w(q\uparrow) \wedge \\ \forall_{X:\texttt{graf}}\left(X(q\uparrow) \wedge \forall_{r:\texttt{graf}}\left(\begin{array}{l} X(r) \wedge (r\uparrow)\downarrow \Rightarrow \\ X(r\uparrow.\texttt{nast}) \end{array}\right) \Rightarrow X(p)\right) \end{array}\right)\right) \wedge$$

$$\forall_{p,q,r:\texttt{graf}}\left(p\uparrow.\texttt{nast} = r \wedge q\uparrow.\texttt{nast} = r \wedge r \neq \texttt{NIL} \Rightarrow p = q\right) \wedge$$

$$\forall_{p:\texttt{graf}}\left((p\uparrow)\downarrow \wedge p\uparrow.\texttt{s} \neq \texttt{NIL} \Rightarrow head_k(p\uparrow)\right) \wedge$$

$$\forall_{p:\texttt{sąsiedzi}}\left(head_k(p) \Rightarrow \forall_{X:\texttt{sąsiedzi}}\left(X(p) \wedge \forall_{q:\texttt{sąsiedzi}}\left(\begin{array}{l} X(q) \wedge (q\uparrow)\downarrow \Rightarrow \\ X(q\uparrow.\texttt{nast}) \end{array}\right) \Rightarrow X(\texttt{NIL})\right)\right) \wedge$$

$$\forall_{p:\texttt{sąsiedzi}}\left((p\uparrow)\downarrow \Rightarrow \exists_{q:\texttt{graf}}\left(\begin{array}{l} head_k(q\uparrow.\texttt{s}) \wedge \\ \forall_{X:\texttt{sąsiedzi}}\left(\begin{array}{l} X(q) \wedge \\ \forall_{r:\texttt{sąsiedzi}}\left(\begin{array}{l} X(r) \wedge (r\uparrow)\downarrow \Rightarrow \\ X(r\uparrow.\texttt{nast}) \end{array}\right) \Rightarrow X(p) \end{array}\right) \end{array}\right)\right) \wedge$$

$$\forall_{p,q,r:\texttt{sąsiedzi}}\left(p\uparrow.\texttt{nast} = r \wedge q\uparrow.\texttt{nast} = r \wedge r \neq \texttt{NIL} \Rightarrow p = q\right) \wedge$$

$$\forall_{p:\texttt{pgraf}}\left((p\uparrow)\downarrow \Rightarrow \exists_{X:\texttt{graf}}\left(X(p\uparrow) \wedge \forall_{q:\texttt{graf}}\left(\begin{array}{l} X(q) \wedge (q\uparrow)\downarrow \Rightarrow \\ X(r\uparrow.\texttt{nast}) \wedge \\ \exists_{Y:\texttt{sąsiedzi}}\left(\varphi_Y\right) \end{array}\right)\right)\right) \quad ,$$

gdzie

$$\varphi_Y \mathrel{\hat{=}} Y(q\uparrow.\texttt{s}) \wedge \forall_{r:\texttt{sąsiedzi}}\left(\begin{array}{l} Y(r) \wedge (r\uparrow)\downarrow \Rightarrow \\ Y(r\uparrow.\texttt{nast}) \wedge \\ \forall_{Z:\texttt{graf}}\left(\begin{array}{l} Z(p\uparrow) \wedge \\ \forall_{s:\texttt{graf}}\left(\begin{array}{l} Z(s) \wedge (s\uparrow)\downarrow \Rightarrow \\ Z(s\uparrow.\texttt{nast}) \end{array}\right) \Rightarrow \\ Z(r\uparrow.\texttt{w}) \end{array}\right) \end{array}\right) \quad .$$

□

Przykład 10 Deklaracje typów dla drzew binarnych mogą mieć następującą postać:

```
TYPE
  tree = POINTER TO node;
  node = RECORD
    d: data;
    l, r: tree
  END;
  ppnode = POINTER TO tree;
  ptree = POINTER TO tree;
```

gdzie pole `d` zawiera wartość przechowywaną w węźle drzewa, a pola `l` i `r` są dowiązaniami do lewego i prawego poddrzewa.

Niezmiennik drzew binarnych można opisać za pomocą koniunkcji następujących warunków:

- zewnętrzne wskaźniki wskazują wyłącznie na korzenie drzew

$$\forall_{p:\texttt{ptree}}\left((p\uparrow)\downarrow \wedge\, p\uparrow \neq \texttt{NIL} \Rightarrow (p\uparrow\uparrow)\downarrow \wedge \forall_{q:\texttt{tree}}\left(q\uparrow.\texttt{l} \neq p\uparrow \wedge\, q\uparrow.\texttt{p} \neq p\uparrow\right)\right) \ ,$$

- każdy węzeł jest osiągalny z pewnego węzła wskazywanego przez zewnętrzny wskaźnik

$$\forall_{p:\texttt{tree}}\left(\begin{array}{l}(p\uparrow)\downarrow \Rightarrow \\ \exists_{q:\texttt{ptree}}\left(\forall_{S:\texttt{tree}}\left(\begin{array}{l}S(q\uparrow) \wedge \\ \forall_{r:\texttt{lista}}\left(\begin{array}{l}(S(r) \wedge (r\uparrow.\texttt{l})\downarrow \Rightarrow S(q\uparrow.\texttt{l})) \ \wedge \\ (S(r) \wedge (r\uparrow.\texttt{p})\downarrow \Rightarrow S(q\uparrow.\texttt{p}))\end{array}\right) \Rightarrow S(p)\end{array}\right)\right)\end{array}\right) \ ,$$

- krawędzie wychodzące z jednego węzła rozchodzą się

$$\forall_{p:\texttt{tree}}((p\uparrow)\downarrow \wedge\, p\uparrow.\texttt{l} = p\uparrow.\texttt{p} \Rightarrow p\uparrow.\texttt{l} = \texttt{NIL}) \ ,$$

- do każdego węzła prowadzi co najwyżej jedna krawędź

$$\forall_{p,q,r:\texttt{tree}}\left((p\uparrow.\texttt{l} = r \vee p\uparrow.\texttt{p} = r) \wedge (q\uparrow.\texttt{l} = r \vee q\uparrow.\texttt{p} = r) \wedge r \neq \texttt{NIL} \Rightarrow p = q\right) \ ,$$

- każda ścieżka w drzewie jest zakończona wskaźnikiem `NIL`

$$\forall_{p:\texttt{tree}}\left(\begin{array}{l}(p\uparrow)\downarrow \Rightarrow \\ \forall_{S:\texttt{tree}}\left(S(p) \wedge \forall_{q:\texttt{tree}}\left(\begin{array}{l}S(q) \wedge (q\uparrow)\downarrow \Rightarrow \\ (S(q\uparrow.\texttt{l}) \vee S(q\uparrow.\texttt{p}))\end{array}\right) \Rightarrow S(\texttt{NIL})\right)\end{array}\right) \ .$$

Niezmiennik drzew BST można wyrazić wzmacniając niezmiennik drzew binarnych następującą formułą (zakładając, że porządek na wartościach typu `data` opisuje relacja $\leq$):

$$\forall_{p:\texttt{tree}}\left(\begin{array}{l}\left(\begin{array}{l}(p\uparrow.\texttt{l}\uparrow)\downarrow \Rightarrow \\ \exists_{S:\texttt{elem}}\left(\begin{array}{l}S(p\uparrow.\texttt{l}\uparrow) \wedge \\ \forall_{q:\texttt{elem}}\left(\begin{array}{l}(S(q) \wedge (q.\texttt{l}\uparrow)\downarrow \Rightarrow S(q.\texttt{l}\uparrow)) \wedge \\ (S(q) \wedge (q.\texttt{p}\uparrow)\downarrow \Rightarrow S(q.\texttt{p}\uparrow))\end{array}\right) \wedge \\ \forall_{q:\texttt{tree}}(S(q) \Rightarrow q.\texttt{x} \leq p\uparrow.\texttt{x})\end{array}\right)\end{array}\right) \wedge \\ \left(\begin{array}{l}(p\uparrow.\texttt{p}\uparrow)\downarrow \Rightarrow \\ \exists_{S:\texttt{elem}}\left(\begin{array}{l}S(p\uparrow.\texttt{p}\uparrow) \wedge \\ \forall_{q:\texttt{elem}}\left(\begin{array}{l}(S(q) \wedge (q.\texttt{l}\uparrow)\downarrow \Rightarrow S(q.\texttt{l}\uparrow)) \wedge \\ (S(q) \wedge (q.\texttt{p}\uparrow)\downarrow \Rightarrow S(q.\texttt{p}\uparrow))\end{array}\right) \wedge \\ \forall_{q:\texttt{tree}}(S(q) \Rightarrow p\uparrow.\texttt{x} \leq q.\texttt{x})\end{array}\right)\end{array}\right)\end{array}\right) \ .$$

□

W podobny sposób można wyspecyfikować kopce binarne.

Choć logika monadyczna drugiego rzędu jest wystarczająco silna, aby opisać większość stosowanych wskaźnikowych struktur danych, to okazuje się, że nie wszystkie takie struktury można za jej pomocą wyspecyfikować. Przykładem wskaźnikowych struktur danych, których nie można wyspecyfikować, są różnego rodzaju drzewa zrównoważone, np. drzewa AVL. Wynika to stąd, że nie jesteśmy w stanie porównywać długości (rozłącznych) ścieżek. W dalszej części pracy sprawdzamy możliwość wyspecyfikowania takich struktur danych na przykładzie drzew AVL.

Fakt 5 Rozważmy następujące deklaracje typów:

```
TYPE
  AVL = POINTER TO node;
  node = RECORD
    d: data;
    lw, pw: BOOLEAN;
    l, r: AVL
  END;
  ppnode = POINTER TO AVL;
  pAVL = POINTER TO AVL;
```

gdzie pole `d` reprezentuje wartość przechowywaną w danym węźle, pola `lw` i `pw` są równe *true* odpowiednio, gdy prawe/lewe poddrzewo ma większą wysokość, a pola `l` i `r` są dowiązaniami do lewego i prawego poddrzewa. Dla powyższych deklaracji nie istnieje formuła *MSOL* będąca niezmiennikiem drzew AVL z powtórzeniami. □

Należy zaznaczyć, że gdyby w każdym węźle zamiast znaczników mówiących, które poddrzewo ma większą wysokość, pamiętać liczbę naturalną będącą wysokością poddrzewa o korzeniu w danym węźle, to można by wyspecyfikować niezmiennik drzew AVL. Odbiega to jednak od opisów drzew AVL spotykanych w literaturze [BDR96, BKR87, Wir99] i wymaga wprowadzenia do języka specyfikacji arytmetyki liczb naturalnych.

W dowodzie tego faktu pomocne będą nam następujące pojęcia i lemat.

Def. 29 Sygnaturą etykietowanego drzewa binarnego nazywamy każdą sygnaturę postaci:

$$\Sigma_T = \langle \{v, \texttt{BOOLEAN}\}, \{k :\rightarrow v, l, p : v \rightarrow v, \mathit{true}, \mathit{false} :\rightarrow \texttt{BOOLEAN}, e_1, \ldots, e_n : v \rightarrow \texttt{BOOLEAN}\} \rangle \ .$$

Skończonym etykietowanym Σ_T-drzewem binarnym nazwiemy dowolną taką Σ_T-algebrę częściową A, że

- $|A|_{\texttt{BOOLEAN}} = \{\mathit{true}, \mathit{false}\}$, $\mathit{true}^A = \mathit{true}$, $\mathit{false}^A = \mathit{false}$,
- funkcje $e_1^A, \ldots, e_n^A$ są wszędzie określone,
- k^A jest określone,
- $\forall_{x \in |A|_v} l^A(x) \neq p^A(x)$,
- graf skierowany $\langle V, E \rangle$, o zbiorze wierzchołków $V = |A|_v$, oraz takim zbiorze krawędzi, że $\langle v, w \rangle \in E \Leftrightarrow (l^A(v) = w \vee p^A(v) = w)$, jest skończonym drzewem (o korzeniu k^A).

Dla każdego $x \in |A|_v$, wysokością x, $h(x)$ nazwiemy długość najdłuższej ścieżki wychodzącej z x w grafie skierowanym $\langle V, E\rangle$ opisanym powyżej. A nazwiemy kształtem drzewa AVL wtw., gdy dla każdego $x \in |A|_v$ zachodzi:

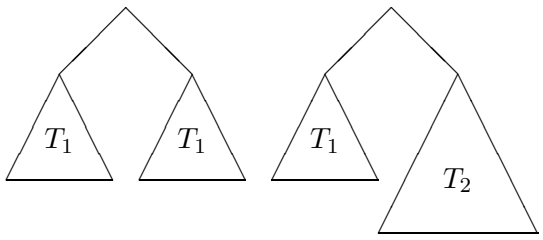
- jeśli $l^A(x)\downarrow$ i $p^A(x)\downarrow$, to $|h(l^A(x)) - h(p^A(x))| \leq 1$,
- jeśli $l^A(x)\downarrow$ i $\neg p^A(x)\downarrow$, to $h(l^A(x)) = 1$,
- jeśli $\neg l^A(x)\downarrow$ i $p^A(x)\downarrow$, to $h(p^A(x)) = 1$.

□

Intuicyjnie, k^A jest korzeniem drzewa, l^A i p^A reprezentują krawędzie drzewa, a $e_1, \ldots, e_n$ reprezentują etykiety węzłów.

Lemat 1 Niech Σ_T będzie ustaloną sygnaturą etykietowanych drzew binarnych. Nie istnieje taka formuła *MSOL* φ (bez zmiennych wolnych), że dla każdego skończonego etykietowanego Σ_T-drzewa binarnego A, $A \models \varphi$ wtw., gdy A jest kształtem drzewa AVL. □

Dowód Załóżmy przeciwnie. Wówczas (por. [Tho97]) istnieje deterministyczny automat skończeniestanowy A wspinający się po skończonych etykietowanych Σ_T-drzewach binarnych, akceptujący tylko drzewa będące kształtami drzew AVL. Niech Q oznacza zbiór stanów tego automatu oraz niech X_q oznacza zbiór wysokości drzew będących kształtami drzew AVL, w których korzeniach A przyjmuje stan q. Ponieważ drzew AVL o danej wysokości jest skończenie wiele, a wszystkich drzew AVL jest nieskończenie wiele, więc dla pewnego $q \in Q$, X_q jest nieskończone. Istnieją więc dwa drzewa AVL: T_1 i T_2, o wysokościach różniących się przynajmniej o 2, takie, że w ich korzeniach A przyjmuje stan q. Wówczas automat A tak samo zaakceptuje lub nie, drzewo powstałe z połączenia dwóch drzew T_1, jak i drzewo powstałe z połączenia drzewa T_1 i T_2 (por. poniższy rysunek).



Jednakże pierwsze z nich jest kształtem drzewa AVL, a drugie nie. Otrzymujemy stąd sprzeczność. ■

Dowód faktu 5 Dowód przebiega nie wprost. Załóżmy przeciwnie, że istnieje formuła *MSOL* będąca niezmiennikiem struktury danych dla drzew AVL.

Wybór drzew AVL z powtórzeniami pozwala nam ominąć przypadek, gdy zbiór wartości typu `data` jest skończony, a przez to i zbiór drzew AVL bez powtórzeń jest skończony. Skoncentrujmy się na samych kształtach drzew AVL pomijając wartości pól `d`. Zauważmy, że jeżeli w dowolnym drzewie AVL zmienimy wartości pól `d` we wszystkich węzłach na tę samą, ustaloną wartość, to otrzymamy poprawne drzewo AVL (z samymi powtórzeniami). Zauważmy

ponadto, że pola `lg` i `pg` są typu `BOOLEAN`, a więc mogą przyjmować skończoną liczbę wartości. Niech Σ_T będzie sygnaturą etykietowanych drzew binarnych zawierającą dwie funkcje etykietujące węzły: $\mathtt{lg}, \mathtt{pg} : v \to \mathtt{BOOLEAN}$. Skoro istnieje formuła *MSOL* będąca niezmiennikiem drzew AVL, to istnieje również formuła *MSOL*, nad sygnaturą Σ_T, bez zmiennych wolnych, charakteryzująca skończone etykietowane Σ_T-drzewa binarne, będące kształtami drzew AVL. Z lematu 1 otrzymujemy sprzeczność. ■

Analizowane w tym podrozdziale przykłady specyfikacji mogą się wydawać niepotrzebnie rozbudowane, a przez to mniej czytelne. Wynika to stąd, że zawierają one wielokrotnie powtarzane definicje takich własności grafowych jak spójność, osiągalność czy skończoność ścieżek. Wydaje się więc, że wprowadzenie do języka specyfikacji takich pojęć jak ścieżka czy osiągalność pozwoliłoby wydatnie skrócić specyfikacje wskaźnikowych struktur danych. Z drugiej strony, fakt 5 wskazuje na ograniczoną stosowalność *MSOL* do specyfikowania wskaźnikowych struktur danych.

4.2 Specyfikacje wskaźnikowych struktur danych w *SOL*

Logika drugiego rzędu zawiera w sobie logikę monadyczną drugiego rzędu. Dlatego też nie przytaczamy tu przykładów specyfikacji struktur danych, które można wyspecyfikować w *MSOL*. Przykłady zawarte w p. 4.1 ilustrują również, w jaki sposób można te struktury danych wyspecyfikować w *SOL*.

Logika drugiego rzędu, jako silniejsza od *MSOL*, pozwala na wyspecyfikowanie drzew AVL. Kluczowa jest tu możliwość porównania wysokości poddrzew. Możemy tego dokonać posługując się relacją binarną zawierającą pary wierzchołków leżących na tej samej głębokości w danym poddrzewie. Ilustruje to następujący przykład.

Przykład 11 Deklaracje typów drzew AVL mogą mieć postać jak w fakcie 5:

```
TYPE
  AVL = POINTER TO node;
  node = RECORD
    d: data;
    lw, pw: BOOLEAN;
    l, r: AVL
  END;
  ppnode = POINTER TO AVL;
  pAVL = POINTER TO AVL;
```

Niezmiennik drzew BST może mieć postać jak w przykładzie 10. Niezmiennik drzew AVL można uzyskać wzmacniając niezmiennik drzew BST następującą formułą:

$$\forall_{p:\mathtt{tree}} \left((p\uparrow)\downarrow \Rightarrow \left(\begin{array}{l} (p\uparrow.\mathtt{l} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p} = \mathtt{NIL}) \vee \\ (p\uparrow.\mathtt{l} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p}\uparrow.\mathtt{l} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p}\uparrow.\mathtt{p} = \mathtt{NIL}) \vee \\ (p\uparrow.\mathtt{l}\uparrow.\mathtt{l} = \mathtt{NIL}) \wedge p\uparrow.\mathtt{l}\uparrow.\mathtt{p} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p} = \mathtt{NIL}) \vee \\ \forall_{\substack{R:\mathtt{tree}\times\mathtt{tree} \\ L,P:\mathtt{tree}}} \left(\begin{pmatrix} \varphi_{poziomy} \wedge \\ \varphi_{poddrzewo}(L, p\uparrow.\mathtt{l}) \wedge \\ \varphi_{poddrzewo}(P, p\uparrow.\mathtt{p}) \end{pmatrix} \Rightarrow \begin{pmatrix} \varphi_{głębokość}(L,P) \wedge \\ \varphi_{głębokość}(P,L) \end{pmatrix} \right) \end{array} \right) \right),$$

gdzie:

$$\varphi_{poziomy} \mathrel{\hat{=}}$$
$$R(p,p) \wedge$$
$$\forall_{x,y:\texttt{tree}} \left(\begin{pmatrix} R(x,y) \wedge \\ (x\uparrow)\downarrow \wedge \\ (y\uparrow)\downarrow \end{pmatrix} \Rightarrow \begin{pmatrix} (x\uparrow.\texttt{l} \neq \texttt{NIL} \wedge y\uparrow.\texttt{l} \neq \texttt{NIL} \Rightarrow R(x\uparrow.\texttt{l}, y\uparrow.\texttt{l})) \wedge \\ (x\uparrow.\texttt{l} \neq \texttt{NIL} \wedge y\uparrow.\texttt{p} \neq \texttt{NIL} \Rightarrow R(x\uparrow.\texttt{l}, y\uparrow.\texttt{p})) \wedge \\ (x\uparrow.\texttt{p} \neq \texttt{NIL} \wedge y\uparrow.\texttt{l} \neq \texttt{NIL} \Rightarrow R(x\uparrow.\texttt{p}, y\uparrow.\texttt{l})) \wedge \\ (x\uparrow.\texttt{p} \neq \texttt{NIL} \wedge y\uparrow.\texttt{p} \neq \texttt{NIL} \Rightarrow R(x\uparrow.\texttt{p}, y\uparrow.\texttt{p})) \end{pmatrix} \right) ,$$

$$\varphi_{poddrzewo}(A,x) \mathrel{\hat{=}}$$
$$A(x) \wedge \forall_{y:\texttt{tree}} \left(\begin{pmatrix} A(y) \wedge \\ (y\uparrow)\downarrow \end{pmatrix} \Rightarrow \begin{pmatrix} (y\uparrow.\texttt{l} \neq \texttt{NIL} \Rightarrow A(y\uparrow.\texttt{l})) \wedge \\ (y\uparrow.\texttt{p} \neq \texttt{NIL} \Rightarrow A(y\uparrow.\texttt{p})) \end{pmatrix} \right) \wedge$$
$$\forall_{B:\texttt{tree}} \left(\begin{matrix} B(x) \wedge \forall_{y:\texttt{tree}} \left(\begin{pmatrix} B(y) \wedge \\ (y\uparrow)\downarrow \end{pmatrix} \Rightarrow \begin{pmatrix} (y\uparrow.\texttt{l} \neq \texttt{NIL} \Rightarrow B(y\uparrow.\texttt{l})) \wedge \\ (y\uparrow.\texttt{p} \neq \texttt{NIL} \Rightarrow B(y\uparrow.\texttt{p})) \end{pmatrix} \right) \Rightarrow \\ \forall_{y:\texttt{tree}} \left(A(y) \Rightarrow B(y) \right) \end{matrix} \right) ,$$

$$\varphi_{głębokość}(A,B) \mathrel{\hat{=}}$$
$$\forall_{a:\texttt{tree}} \left(A(a) \wedge (a\uparrow)\downarrow \wedge (a\uparrow.\texttt{l} \neq \texttt{NIL} \vee a\uparrow.\texttt{p} \neq \texttt{NIL}) \Rightarrow \exists_{b:\texttt{tree}} \left(B(b) \wedge R(a,b) \right) \right) .$$

Powyższe formuły pomocnicze mają następujące intuicyjne znaczenie:

- $\varphi_{poziomy}$ mówi, że relacja R zawiera wszystkie takie pary $\langle x, y \rangle$, że x i y są wskaźnikami do węzłów w poddrzewie o korzeniu p oraz x i y leżą na tej samej wysokości,
- $\varphi_{poddrzewo}$ mówi, że A zawiera dokładnie wskaźniki do węzłów tworzących poddrzewo o korzeniu x,
- $\varphi_{głębokość}$ mówi, że poddrzewo A nie jest wyższe od B o więcej niż 1.

Kolejne alternatywy w niezmienniku odpowiadają następującym przypadkom:

- p jest liściem,
- p ma tylko prawego syna, który jest liściem,
- p ma tylko lewego syna, który jest liściem,
- p ma lewego i prawego syna, przy czym wysokość prawego poddrzewa nie jest większa od wysokości lewego poddrzewa o więcej niż 1, a wysokość lewego poddrzewa nie jest większa od wysokości prawego poddrzewa o więcej niż 1.

Przypomnijmy, że powyższe formuły nie stanowią całego niezmiennika struktury danych drzew AVL, a jedynie wzmocnienie niezmiennika drzew BST do niezmiennika drzew AVL. □

Powyższy przykład wyraźnie pokazuje, że choć *SOL* ma wystarczającą siłę wyrazu, aby opisać interesujące nas własności struktur danych, to zapisane w niej specyfikacje mogą być bardzo rozbudowane, a przez to mało czytelne. Dużą część powyższej specyfikacji zajmuje

wyrażenie takich własności, jak przynależność wierzchołka do poddrzewa czy porównanie odległości dwóch wierzchołków od zadanego wierzchołka. Przy tym opis tych własności ma charakter „indukcyjny". Wydaje się więc, że wprowadzenie do języka specyfikacji takich pojęć jak osiągalność, ścieżka i zestawienie ze sobą kolejnych elementów kilku ścieżek, lub też wprowadzenie operatora najmniejszego punktu stałego mogą uprościć specyfikacje wskaźnikowych struktur danych.

4.3 Specyfikacje wskaźnikowych struktur danych w *FPL*

Na logikę stałopunktową możemy spojrzeć jak na ograniczenie logiki drugiego rzędu — zmienne drugiego rzędu mogą być wprowadzane nie przez kwantyfikatory, ale przez operatory najmniejszego i największego punktu stałego. Prześledźmy więc, co się zmieni w specyfikacjach przykładowych struktur danych, gdy zapiszemy je w logice stałopunktowej.

Przykład 12 Deklaracja struktury danych modułu implementującego listy jednokierunkowe może wyglądać następująco (por. przykłady 2 i 6):

```
TYPE
  lista = POINTER TO elem;
  elem = RECORD
    d: data;
    nast: lista
  END;
  ppelem = POINTER TO lista;
  plista = POINTER TO lista;
```

Początki list można scharakteryzować następującą formułą:

$$head(p : \texttt{lista}) \mathrel{\hat{=}} (p\uparrow)\downarrow \wedge \forall_{q:\texttt{lista}}(q\uparrow.\texttt{nast} \neq p) \ .$$

Niezmiennik list jednokierunkowych może być opisany za pomocą następującej formuły:

$$\forall_{p:\texttt{plista}}\left((p\uparrow)\downarrow \wedge\, p\uparrow \neq \texttt{NIL} \Rightarrow head(p\uparrow)\right) \wedge$$

$$\forall_{p:\texttt{lista}}\left(head(p) \Rightarrow \mu_{q:\texttt{lista},S}\left(q = p \vee S(q) \vee \exists_{r:\texttt{lista}}\left(S(r) \wedge r\uparrow.\texttt{nast} = q\right)\right)(\texttt{NIL})\right) \wedge$$

$$\forall_{p:\texttt{lista}}\left((p\uparrow)\downarrow \Rightarrow \exists_{q:\texttt{plista}}\left(head(q\uparrow) \wedge \mu_{r:\texttt{lista},S}\left(\begin{array}{l} r = q\uparrow \vee S(r) \vee \\ \exists_{t:\texttt{lista}}\left(S(t) \wedge t\uparrow.\texttt{nast} = r\right)\end{array}\right)(p)\right)\right) \wedge$$

$$\forall_{p,q,r:\texttt{lista}}\left(p\uparrow.\texttt{nast} = r \wedge q\uparrow.\texttt{nast} = r \wedge r \neq \texttt{NIL} \Rightarrow p = q\right)$$

□

Podobnie do list jednokierunkowych można wyspecyfikować grafy skierowane — por. przykład 9.

Przykład 13 Moduł implementujący cykliczne listy dwukierunkowe może wykorzystywać następujące deklaracje typów (por. przykład 7):

```
TYPE
  lista = POINTER TO elem;
  elem = RECORD
    d: dane;
    pop, nast: lista
  END;
ppelem = POINTER TO lista;
plista = POINTER TO lista;
```

Następujący niezmiennik struktury danych określa dopuszczalne kształty list oraz wyraża fakt, że każda lista jest wskazywana przez dokładnie jeden zewnętrzny wskaźnik i vice versa:

$$\forall_{p:\texttt{lista}} \left((p\uparrow)\downarrow \Rightarrow \begin{pmatrix} p\uparrow.\texttt{nast}\uparrow.\texttt{pop} = p \ \wedge p\uparrow.\texttt{pop}\uparrow.\texttt{nast} = p \wedge \\ \mu_{q:\texttt{lista},X} \begin{pmatrix} q = p\uparrow.\texttt{nast} \vee X(q) \vee \\ \exists_{r:\texttt{lista}} (X(r) \wedge q = r\uparrow.\texttt{nast}) \end{pmatrix} (p) \wedge \\ \mu_{q:\texttt{lista},X} \begin{pmatrix} q = p\uparrow.\texttt{pop} \vee X(q) \vee \\ \exists_{r:\texttt{lista}} (X(r) \wedge q = r\uparrow.\texttt{pop}) \end{pmatrix} (p) \wedge \\ \varphi \end{pmatrix} \right) \wedge ,$$
$$\forall_{p:\texttt{plista}} ((p\uparrow)\downarrow \wedge p\uparrow \neq \texttt{NIL} \Rightarrow (p\uparrow\uparrow)\downarrow)$$

gdzie

$$\varphi \mathrel{\hat{=}} \exists_{q:\texttt{plista}} \left(\begin{pmatrix} (q\uparrow)\downarrow \ \wedge \\ \forall_{r:\texttt{plista}} \left(\begin{pmatrix} (r\uparrow)\downarrow \Rightarrow \\ \begin{pmatrix} r = q \Leftrightarrow \\ \mu_{s:\texttt{lista},X} \begin{pmatrix} s = q\uparrow \vee X(s) \vee \\ \exists_{t:\texttt{lista}} \begin{pmatrix} X(t) \wedge \\ s = t\uparrow.\texttt{nast} \end{pmatrix} \end{pmatrix} (r\uparrow) \end{pmatrix} \right) \end{pmatrix} \right) .$$

□

Przykład 14 Moduł implementujący drzewa *find-union* (por. przykład 8), używane do reprezentowania rozłącznych zbiorów elementów, może mieć bardzo proste deklaracje typów:

```
TYPE
  node = POINTER TO node;
```

Niezmiennik tej struktury danych wyraża fakt, że każda ścieżka zaczynająca się w dowolnym elemencie struktury prowadzi do elementu wskazującego na samego siebie:

$$\forall_{p:\texttt{node}} (\exists_{q:\texttt{node}} (\mu_{r:\texttt{node},X} (r = p \vee X(r) \vee \exists_{s:\texttt{node}} (X(s) \wedge r = s\uparrow)) (q) \wedge q\uparrow = q)) \ .$$

Jedną z operacji, jakie możemy wykonywać na drzewach *find-union*, jest operacja *find*, która dla zadanego wskaźnika p do elementu struktury danych zmienia jego wartość tak, aby wskazywał na reprezentanta zbioru, do którego należy dany element, a przy okazji dokonuje kompresji ścieżki prowadzącej do tego reprezentanta. W warunku wstępnym tej operacji, oprócz niezmiennika danych, wymagamy, aby początkowa wartość p wskazywała na istniejący element: $(p\uparrow)\downarrow$. Warunek końcowy ma następującą postać:

$$p`\uparrow = p \wedge p\uparrow = p \wedge$$

$$\mu_{q:\mathtt{node},X}\left(q = \backprime p \vee X(q) \vee \exists_{r:\mathtt{node}}\left(X(r) \wedge q = r\backprime\uparrow\right)\right)(p) \wedge$$

$$\forall_{q:\mathtt{node}}\left(\mu_{r:\mathtt{node},X}\left(r = \backprime p \vee X(r) \vee \exists_{s:\mathtt{node}}\left(X(r) \wedge r = s\backprime\uparrow\right)\right)(q) \Rightarrow q\uparrow = p\right) \wedge$$

$$\forall_{q:\mathtt{node}}\left(\neg\mu_{r:\mathtt{node},X}\left(r = \backprime p \vee X(r) \vee \exists_{s:\mathtt{node}}\left(X(r) \wedge r = s\backprime\uparrow\right)\right)(q) \Rightarrow q\backprime\uparrow = q\uparrow\right) \quad .$$

□

Przykład 15 Deklaracje typów dla drzew binarnych mogą mieć następującą postać (por. przykład 10):

```
TYPE
  tree = POINTER TO node;
  node = RECORD
    d: data;
    l, r: tree
  END;
  ppnode = POINTER TO tree;
  ptree = POINTER TO tree;
```

gdzie pole `d` zawiera wartość przechowywaną w węźle drzewa, a pola `l` i `r` są dowiązaniami do lewego i prawego poddrzewa.

Niezmiennik drzew binarnych można opisać za pomocą następującej koniunkcji:

$$\forall_{p:\mathtt{ptree}}\left((p\uparrow)\downarrow \wedge p \neq \mathtt{NIL} \Rightarrow (p\uparrow\uparrow)\downarrow \ \wedge \forall_{q:\mathtt{tree}}\left(q\uparrow.\mathtt{l} \neq p\uparrow \wedge q\uparrow.\mathtt{p} \neq p\uparrow\right)\right) \wedge$$

$$\forall_{p:\mathtt{tree}}\left((p\uparrow)\downarrow \Rightarrow \exists_{q:\mathtt{ptree}}\left(\mu_{r:\mathtt{tree},X}\begin{pmatrix} r = q\uparrow \vee X(r) \vee \\ \exists_{s:\mathtt{lista}}\left(X(s) \wedge (r = s\uparrow.\mathtt{l} \vee r = s\uparrow.\mathtt{p})\right)\end{pmatrix}(p)\right)\right) \wedge$$

$$\forall_{p:\mathtt{tree}}\left((p\uparrow)\downarrow \wedge p\uparrow.\mathtt{l} = p\uparrow.\mathtt{p} \Rightarrow p\uparrow.\mathtt{l} = \mathtt{NIL}\right) \wedge$$

$$\forall_{p,q,r:\mathtt{tree}}\left((p\uparrow.\mathtt{l} = r \vee p\uparrow.\mathtt{p} = r) \wedge (q\uparrow.\mathtt{l} = r \vee q\uparrow.\mathtt{p} = r) \wedge r \neq \mathtt{NIL} \Rightarrow p = q\right) \wedge$$

$$\forall_{p:\mathtt{tree}}\left((p\uparrow)\downarrow \Rightarrow \mu_{q:\mathtt{tree},X}\left(q = \mathtt{NIL} \vee X(q) \vee \left(X(q\uparrow.\mathtt{l}) \wedge X(q\uparrow.\mathtt{p})\right)\right)(p)\right) \quad .$$

Niezmiennik drzew BST można wyrazić wzmacniając niezmiennik drzew binarnych następującą formułą (zakładamy, że porządek na wartościach typu `data` jest dany relacją $\leq$):

$$\forall_{p:\mathtt{tree}}\left(\begin{pmatrix}(p\uparrow.\mathtt{l}\uparrow)\downarrow \Rightarrow \\ \forall_{q:\mathtt{tree}}\left(\mu_{r:\mathtt{tree},X}\begin{pmatrix} r = p\uparrow.\mathtt{l} \vee X(r) \vee \\ \exists_{s:\mathtt{tree}}\begin{pmatrix}(X(s) \wedge r = s\uparrow.\mathtt{l}) \ \vee \\ (X(s) \wedge r = s\uparrow.\mathtt{p})\end{pmatrix}\end{pmatrix}(q) \wedge \right. \\ \left. q \neq \mathtt{NIL} \Rightarrow q\uparrow.\mathtt{x} \leq p\uparrow.\mathtt{x}\right)\end{pmatrix} \wedge \right. \\ \left.\begin{pmatrix}(p\uparrow.\mathtt{r}\uparrow)\downarrow \Rightarrow \\ \forall_{q:\mathtt{tree}}\left(\mu_{r:\mathtt{tree},X}\begin{pmatrix} r = p\uparrow.\mathtt{r} \vee X(r) \vee \\ \exists_{s:\mathtt{tree}}\begin{pmatrix}(X(s) \wedge r = s\uparrow.\mathtt{l}) \ \vee \\ (X(s) \wedge r = s\uparrow.\mathtt{p})\end{pmatrix}\end{pmatrix}(q) \wedge \right. \\ \left. q \neq \mathtt{NIL} \Rightarrow p\uparrow.\mathtt{x} \leq q\uparrow.\mathtt{x}\right)\end{pmatrix}\right) \quad .$$

Podobnie do drzew BST można wyspecyfikować kopce binarne.

Niezmiennik drzew AVL można uzyskać wzmacniając niezmiennik drzew BST następującą formułą:

$$\forall_{p:\mathtt{tree}} \left((p\uparrow)\downarrow \Rightarrow \left(\begin{array}{l} (p\uparrow.\mathtt{l} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p} = \mathtt{NIL}) \vee \\ (p\uparrow.\mathtt{l} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p}\uparrow.\mathtt{l} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p}\uparrow.\mathtt{p} = \mathtt{NIL}) \vee \\ (p\uparrow.\mathtt{l}\uparrow.\mathtt{l} = \mathtt{NIL}) \wedge p\uparrow.\mathtt{l}\uparrow.\mathtt{p} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p} = \mathtt{NIL}) \vee \\ \forall_{q:\mathtt{tree}} \left(\begin{array}{l} \left(\begin{array}{l} (q\uparrow)\downarrow \wedge \varphi_{subtree}(p\uparrow.\mathtt{l}, q) \wedge \\ \left(\begin{array}{l} q\uparrow.\mathtt{l} \neq \mathtt{NIL} \vee \\ q\uparrow.\mathtt{p} \neq \mathtt{NIL} \end{array} \right) \Rightarrow \exists_{r:\mathtt{tree}} \left(\begin{array}{l} \varphi_{subtree}(p\uparrow.\mathtt{p}, r) \wedge \\ \varphi_{depth}(p, q, r) \end{array} \right) \end{array} \right) \wedge \\ \left(\begin{array}{l} (q\uparrow)\downarrow \wedge \varphi_{subtree}(p\uparrow.\mathtt{p}, q) \wedge \\ \left(\begin{array}{l} q\uparrow.\mathtt{l} \neq \mathtt{NIL} \vee \\ q\uparrow.\mathtt{p} \neq \mathtt{NIL} \end{array} \right) \Rightarrow \exists_{r:\mathtt{tree}} \left(\begin{array}{l} \varphi_{subtree}(p\uparrow.\mathtt{l}, r) \wedge \\ \varphi_{depth}(p, q, r) \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) \right) \right) ,$$

gdzie:

$$\varphi_{subtree}(a, b) \mathrel{\hat{\equiv}} \mu_{c:\mathtt{tree},X} \left(c = a \vee X(c) \vee \exists_{d:\mathtt{tree}} \left(X(d) \wedge (d\uparrow.\mathtt{l} = c \vee d\uparrow.\mathtt{p} = c) \right) \right)(b)$$

wyraża, że b należy do poddrzewa o korzeniu a, oraz

$$\varphi_{depth}(a, b, c) \mathrel{\hat{\equiv}} \mu_{x,y:\mathtt{tree},R} \left(\begin{array}{l} (x = a \wedge y = a) \vee R(x, y) \vee \\ \exists_{d,e:\mathtt{tree}} \left(\begin{array}{l} R(d, e) \wedge (d\uparrow)\downarrow \wedge (e\uparrow)\downarrow \wedge \\ (d\uparrow.\mathtt{l} = x \vee d\uparrow.\mathtt{p} = x) \wedge \\ (e\uparrow.\mathtt{l} = y \vee e\uparrow.\mathtt{p} = y) \end{array} \right) \end{array} \right)(b, c)$$

wyraża, że b i c leżą na równej głębokości względem ich wspólnego przodka a.

Kolejne alternatywy w niezmienniku odpowiadają następującym przypadkom:

- p jest liściem,
- p ma tylko prawego syna, który jest liściem,
- p ma tylko lewego syna, który jest liściem,
- p ma lewego i prawego syna, przy czym wysokość prawego poddrzewa nie jest większa od wysokości lewego poddrzewa o więcej niż 1, a wysokość lewego poddrzewa nie jest większa od wysokości prawego poddrzewa o więcej niż 1.

□

Jak pokazują powyższe przykłady, logika stałopunktowa ma wystarczającą siłę wyrazu potrzebną do specyfikowania wybranych przez nas wskaźnikowych struktur danych. Jednocześnie, użycie operatorów punktu stałego pozwala na uproszczenie niektórych, przeważnie tych bardziej skomplikowanych, konstrukcji. Niestety, prostsze konstrukcje zwykle nie ulegają uproszczeniu lub nawet nieznacznie się wydłużają. Łatwo zauważyć, że zwykle konstrukcje stałopunktowe są wariacjami nt. osiągalności w grafie. Dlatego też wprowadzenie do języka specyfikacji pojęcia osiągalności lub pojęcia ścieżki może je istotnie uprościć.

4.4 Hipergrafowe specyfikacje wskaźnikowych struktur danych

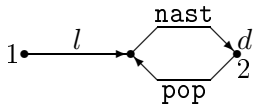
Wspomnieliśmy, że na wskaźnikowe struktury danych można spojrzeć jak na grafy — traktując rekordy jako wierzchołki, a łączące je wskaźniki jako krawędzie. Jednym z formalizmów stosowanych do opisu (hiper-) grafów są gramatyki podmiany hiperkrawędzi [Cou90, Eng94]. Formalizm ten jest o tyle ciekawy, że dostarcza mechanizmu do definiowanaia funkcji określonych na bezkontekstowych zbiorach grafów [Eng94]. Każdy taki zbiór jest generowany przez pewną gramatykę *HRG*. Zgodnie z produkcjami tej gramatyki możemy rekurencyjnie definiować funkcje określone na interesujących nas grafach. Mechanizm ten można by wykorzystać do definiowania funkcji abstrakcji. Powstaje więc pytanie, czy zbiory grafów będących kształtami wskaźnikowych struktur danych są bezkontekstowe. Poniższe przykłady pokazują, że w przypadku niektórych struktur danych odpowiedź na to pytanie jest twierdząca.

Przykład 16 Niech D będzie skończonym zbiorem. Listy cykliczne (por. przykłady 7 i 13) o elementach ze zbioru D możemy reprezentować za pomocą hipergrafów zawierających następujące etykiety hiperkrawędzi:

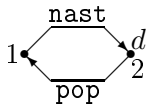
- dwa symbole rzędu 2: `nast` i `pop`,
- dla każdego $d \in D$, symbol d rzędu 1.

Gramatyka generująca wszystkie poprawnie zbudowane listy cykliczne elementów ze zbioru D zawiera dodatkowo dwa symbole nieterminalne: l rzędu 2 i aksjomat z rzędu 0. Gramatyka ta zawiera następujące produkcje (cyfry na rysunkach oznaczają kolejne wierzchołki uchwytów):

$z \rightarrow$ 

$l \rightarrow$ 

dla $d \in D$,

$l \rightarrow$ 

dla $d \in D$,

Podobnie możemy opisać listy jednokierunkowe czy dwukierunkowe. □

Wiadomo [Cou90], że zbiór wszystkich grafów skończonych nie jest bezkontekstowy. Fakt ten przenosi się na wskaźnikowe reprezentacje grafów. Pokażemy go dla reprezentacji grafów skierowanych, w której z każdym wierzchołkiem jest związana jednokierunkowa lista incydencji, a wszystkie wierzchołki tworzące graf są zebrane na liście jednokierunkowej (por. przykład 9).

```
TYPE
  graf = POINTER TO wierzchołek;
  incydencje = POINTER TO krawędź;
  wierzchołek = RECORD
    nast: graf;
    krawędzie: incydencje
  END;
  krawędź = RECORD
    nast: incydencje;
    koniec: wierzchołek
  END;
```

Mówiąc o kształcie takiej struktury danych myślimy o grafie, którego wierzchołkami są adresy alokowanych rekordów typu `wierzchołek` i `krawędź`, a krawędziami są łączące je wskaźniki (różne od `NIL`).

Tw. 1 Zbiór grafów będących kształtami (skończonych) grafów skierowanych nie jest bezkontekstowy. □

Lemat 2 Jeżeli $G_1 = \langle V_1, E_1\rangle$, $G_2 = \langle V_2, E_2\rangle$ dwa skończone grafy nieskierowane, $h : V_1 \to V_2$ epi oraz takie, że:

- h jest epi na zbiorze krawedzi, czyli $\forall_{v_1-v_2\in E_2}\exists_{w_1-w_2\in E_1}\,(h(w_1) = v_1 \wedge h(w_2) = v_2)$,
- przeciwobraz każdego wierzchołka z V_2 względem h tworzy w G_1 spójny podgraf,

to $twd(G_1) \geq twd(G_2)$. □

Dowód Wystarczy pokazać, że h zachowuje pokrycia drzewiaste grafów, gdyż h nie może zwiększać szerokości pokrycia. Niech $\langle T, f\rangle$ będzie pokryciem drzewiastym G_1, $T = \langle V_T, E_T\rangle$, $f : V_T \to \mathcal{P}(V_1)$ oraz niech $g : V_T \to \mathcal{P}(V_2)$, $g = \overrightarrow{h} \circ f$. Pokażemy, że $\langle T, g\rangle$ jest pokryciem drzewiastym G_2.

- Zauważmy, że h jest epi, a więc $\bigcup_{t\in V_T} g(t) = V_2$.
- Dla każdej krawędzi $e \in E_2$ mamy pewne dwa wierzchołki $v_1, v_2 \in V_1$ takie, że $v_1{-}v_2 \in E_1$ oraz $e = h(v_1){-}h(v_2)$. Tak więc dla pewnego $t \in V_T$ mamy $v_1, v_2 \in f(t)$. Stąd $h(v_1), h(v_2) \in g(x)$.
- Niech $v \in V_2$, $t_1, t_2 \in V_T$ takie, że $v \in g(t_1)$ oraz $v \in g(t_2)$. Musimy pokazać, że $v \in g(t)$ dla wszystkich t leżących na jedynej ścieżce prostej łączącej t_1 i t_2 w T. Wystarczy pokazać to dla dowolnej ścieżki łączącej t_1 z t_2.

  Niech $v_1, \ldots, v_n \in V_1$ będzie takim ciągiem wierzchołków, że $h(v_1) = \cdots = h(v_n) = v$, $v_1 \in f(t_1)$, $v_n \in f(t_2)$ oraz $v_1{-}\cdots{-}v_n$ jest ścieżką w grafie G_1. Ciąg taki zawsze istnieje ponieważ $h^{-1}(v)$ tworzy spójny podgraf G_1. Zauważmy, że dla każdej krawędzi $v_i - v_{i+1}$ (dla $1 \leq i \leq n-1$) istnieje $u_i \in V_T$ takie, że $v_i, v_{i+1} \in f(u_i)$. Tak więc v_1 należy do $f(t)$ dla wszystkich $t \in V_T$ ze ścieżki prostej łączącej t_1 z u_1. Podobnie v_i (dla $2 \leq i \leq n-1$) należy do $f(t)$ dla wszystkich $t \in V_T$ ze ścieżki prostej łączącej u_{i-1} z u_i, a także v_n należy do $f(t)$ dla wszystkich $t \in V_T$ ze ścieżki prostej łączącej u_n z t_2. Stąd istnieje w T ścieżka łącząca t_1 z t_2, prowadząca przez $u_1, \ldots, u_n$, taka, że dla każdego t z tej ścieżki $v \in g(t)$.

Zatem $\langle T, g\rangle$ jest faktycznie pokryciem drzewiastym. ■

Dowód twierdzenia 1 Bez straty ogólności możemy traktować kształt struktury wskaźnikowej jako graf nieskierowany. Rozpatrzmy kształt struktury S_n reprezentującej n-elementowy graf pełny K_n. Niech $h : S_n \to K_n$ będzie przekształceniem sklejającym każdy wierzchołek rodzaju $\uparrow$`wierzchołek` ze wszystkimi wierzchołkami rodzaju $\uparrow$`krawędź` tworzącymi jego listę incydencji. Zauważmy, że przekształcenie to spełnia wymagania lematu 2. Stąd $twd(S_n) \geq twd(K_n)$. Z faktu 2 oraz 4 mamy, że $twd(K_n) \geq \frac{wd(K_n)-3}{2} \geq \frac{n-3}{2}$, oraz $wd(S_n) \geq twd(S_n)+1$, czyli $wd(S_n) \geq \frac{n-1}{2}$. Ponieważ każdy bezkontekstowy zbiór grafów ma ograniczoną szerokość, więc zbiór wszystkich kształtów struktur reprezentujących grafy nie jest bezkontekstowy. ■

Z powyższego dowodu widać, że zastosowanie *HRG* jako języka specyfikacji wskaźnikowych struktur danych jest w zasadzie ograniczone do list i struktur drzewiastych. Poniższy przykład pokazuje jak za pomocą tego formalizmu można opisać drzewa BST.

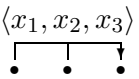
Przykład 17 Załóżmy, że wartości przechowywane w węzłach drzew pochodzą ze skończonego zbioru D, z określoną na nim relacją liniowego porządku $\leq$. Drzewo BST (por. przykłady 10 i 15) możemy sobie przedstawić jako hipergraf zawierający dwa rodzaje hiperkrawędzi: krawędzie rzędu 2, etykietowane symbolami `l` i `p`, reprezentujące krawędzie drzewa oraz krawędzie rzędu 1, etykietowane elementami zbioru D, reprezentujące wartości przechowywane w węzłach. Gramatyka *HRG* opisująca drzewa BST (z powtórzeniami) może mieć następującą postać. Niech B będzie alfabetem ważonym symboli terminalnych zawierającym:

- symbole rzędu 1 postaci d, dla $d \in D$,
- dwa symbole rzędu 2 postaci `l` i `p`.

Niech U będzie alfabetem ważonym symboli nieterminalnych zawierającym:

- aksjomat z rzędu 0,
- symbole rzędu 3 postaci $\langle x_1, x_2, x_3 \rangle$, dla $x_1, x_2, x_3 \in X$, $x_1 \leq x_2 \leq x_3$.

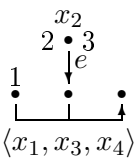
Symbol $\langle x_1, x_2, x_3 \rangle$ reprezentuje poddrzewo z uchwytem zawierającym skrajnie lewy element, korzeń i skrajnie prawy element. Produkcje gramatyki przedstawione są poniżej. Liczby na rysunkach reprezentują kolejne wierzchołki uchwytów.

$z \to$ 

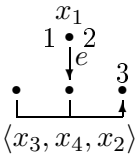
dla $x_1, x_2, x_3 \in D, x_1 \leq x_2 \leq x_3$,

$\langle x, x, x \rangle \to$ 

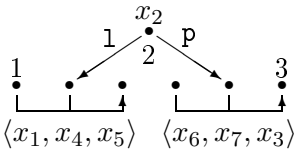
dla $x \in D$,

$\langle x_1, x_2, x_2 \rangle \to$ 

dla $x_1, x_2, x_3, x_4 \in D$,
$x_1 \leq x_3 \leq x_4 \leq x_2, e \in \{\texttt{l}, \texttt{p}\}$,

$\langle x_1, x_1, x_2 \rangle \rightarrow$ 

dla $x_1, x_2, x_3, x_4 \in D$,
$x_1 \leq x_3 \leq x_4 \leq x_2, e \in \{\mathtt{l}, \mathtt{p}\}$,

$\langle x_1, x_2, x_3 \rangle \rightarrow$ 

dla $x_1, \ldots, x_7 \in D$,
$x_1 \leq x_4 \leq x_5 \leq x_2 \leq x_6 \leq x_7 \leq x_3$

□

Jeżeli w powyższym opisie gramatyki zrezygnujemy z nierówności, jakie mają spełniać etykiety ze zbioru D, to otrzymamy gramatykę generującą wszystkie drzewa binarne o etykietach ze zbioru D. Podobnie jak drzewa BST możemy też opisać kopce binarne czy drzewa *find-union* — różnica polega tylko na skierowaniu krawędzi terminalnych i ew. relacji między wartościami przechowywanymi w węzłach.

Okazuje się, że zbiór drzew AVL, przechowujących w węzłach wartości ze skończonego zbioru, nie jest bezkontekstowy.

Tw. 2 Niech D będzie skończonym zbiorem z określonym porządkiem liniowym $\leq$ na elementach. Niech B będzie alfabetem ważonym symboli terminalnych zawierającym elementy zbioru B jako symbole rzędu 1 (reprezentujące wartości przechowywane w węzłach), oraz symbole `l` i `p` rzędu 2 (reprezentujące krawędzie drzewa). Zbiór hipergrafów nad alfabetem B będących drzewami AVL (z powtórzeniami) nie jest bezkontekstowy. □

Dowód Dowód przebiega nie wprost. Załóżmy przeciwnie. Niech $\Gamma = \langle B, U, P, Z \rangle$ będzie gramatyką *HRG* generującą zbiór drzew AVL.

Dla uproszczenia załóżmy, że dla każdej etykiety nieterminalnej hiperkrawędzi $h \in U$, z h można wyprowadzić pewien graf terminalny, oraz z aksjomatu Z można wyprowadzić hipergraf zawierający tylko jedną krawędź nieterminalną o etykiecie h. Zauważmy, że usunięcie krawędzi nie spełniających powyższych warunków nie zmienia zbioru grafów terminalnych wyprowadzalnych w Γ.

Mówiąc o spojności, ścieżkach i stopniach wierzchołków będziemy rozpatrywać wyłącznie krawędzie o etykietach `l` i `p`. Dla uproszczenia krawędzie te traktujemy jak nieskierowane.

W dalszej części dowodu twierdzenia korzystamy z następującego lematu.

Lemat 3 Dla dowolnego $l \in \mathbb{N}$, istnieje taka etykieta hiperkrawędzi $h \in U$, że z h można wyprowadzić graf G_h (z $rank(h)$ uchwytami) zawierający tylko jedną hiperkrawędź nieterminalną, o etykiecie h, oraz zawierający co najmniej l wierzchołków. □

Dowód lematu Niech $U_1 \subseteq U$ będzie zbiorem tych etykiet nieterminalnych hiperkrawędzi, z których można wyprowadzić grafy terminalne o dowolnie dużej liczbie wierzchołków. Oczywiście $U_1 \neq \emptyset$, gdyż istnieją drzewa AVL o dowolnie dużej liczbie wierzchołków. Skonstruujemy nieskończony ciąg hipergrafów $(H_i)_{i=0,1,\ldots}$ zawierających tylko po jednej hiperkrawędzi nieterminalnej i to o etykiecie ze zbioru U_1. $H_0 = u$ dla dowolnego, ustalonego $u \in U_1$.

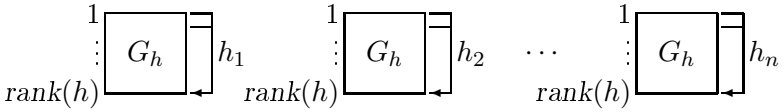
H_{i+1} tworzymy na podstawie H_i. Niech u oznacza jedyną nieterminalną hiperkrawędź w H_i. Rozważmy hipergrafy, które można wyprowadzić z u, zawierające tylko jedną nieterminalną hiperkrawędź i to o etykiecie należącej do U_1. Oczywiście takie hipergrafy istnieją, gdyż z u można wyprowadzić dowolnie duży graf. Zauważmy również, że wśród tych grafów istnieją takie, które zawierają co najmniej l wierzchołków. H_{i+1} otrzymujemy z H_i zastępując u dowolnie wybranym takim hipergrafem.

Etykiety jedynych hiperkrawędzi występujących w H_i muszą się powtarzać. Niech $h \in U_1, i, j \in \mathbb{N}, i < j$ takie, że h jest etykietą jedynych hiperkrawędzi H_i i H_j. Wówczas z konstrukcji ciągu $(H_i)_{i=0,1,\ldots}$ wynika, że z h można wyprowadzić hipergraf G_h zawierający co najmniej l wierzchołków oraz tylko jedną nieterminalną hiperkrawędź o etykiecie h.

■

Powróćmy do dowodu twierdzenia 2. Niech h i G_h będą jak w lemacie 3 dla $l = 4rank(\Gamma)+1$. Oznaczmy przez $(G_h^i)_{i=1,\ldots}$ następujący ciąg hipergrafów: $G_h^1 = G_h$, $G_h^{i+1} = G_h^i[h_i \to G_h]$, gdzie h_i jest jedyną hiperkrawędzią w G_h^i. Niech G' będzie dowolnym hipergrafem wyprowadzalnym z Z, zawierającym tylko jedną hiperkrawędź h' o etykiecie h, oraz niech G'' będzie dowolnym grafem terminalnym wyprowadzalnym z h. Zauważmy, że dla każdego $n \geq 0$, G_h^n może składać się z co najwyżej $2rank(h)$ spójnych składowych, gdyż $G'[h' \to G_h^n][h_i \to G'']$ jest grafem terminalnym, a więc drzewem AVL. Zauważmy ponadto, że każda krawędź o etykiecie `l` lub `p` należąca do G_h należy również do $G'[h' \to G_h][h_i \to G'']$, oraz, że w drzewie każdy wierzchołek ma stopień co najwyżej 3. Stąd G_h zawiera co najmniej jedną krawędź o etykiecie `l` lub `p`, której końce nie są ani uchwytami, ani nie należą do h_i. Tak więc G_h^n zawiera, co do rzędu, $\Theta(n)$ wierchołków i krawędzi o etykiecie `l` lub `p`.

Pokażemy, że G_h^n zawiera ścieżkę prostą długości rzędu $\Theta(n)$. Dla uproszczenia dalszych rozważań pisząc h_i mamy na myśli wierzchołki G_h^n, które w i-tym kroku konstrukcji tworzyły hiperkrawędź h_i. Na G_h^n możemy spojrzeć jak na sklejenie n kopii G_h.



Zauważmy, że w G_h^n każda krawędź o etykiecie `l` lub `p` pochodzi z jednej z kopii G_h tworzących G_h^n. Z wierzchołkami już tak nie jest, gdyż podstawianie hipergrafu z uchwytem za hiperkrawędź wiąże się ze sklejaniem wierzchołków — wierzchołek może należeć do kilku kolejnych kopii G_h.

Niech $R \subseteq \{1, \ldots rank(h)\}^2$ będzie relacją binarną określoną następująco: $\langle x, y\rangle \in R$ wtw., gdy w G_h wierzchołek będący x-tym uchwytem jest jednocześnie y-tym wierzchołkiem jedynej hiperkrawędzi o etykiecie `l` lub `p`. Zauważmy, że dla pewnych stałych $l, m \geq 1$ mamy $R^l = R^{l+m} = R^{l+2m} = \ldots$. Jeżeli pewien wierzchołek występuje zarówno w i-tej jak i w j-tej kopii G_h (dla $i < j-1$), to musiał on należeć do h_i oraz h_{j-1}. Mamy wówczas takie x i y, że występował on w h_i na x-tym, a w h_{j-1} na y-tym miejscu, oraz $\langle x, y\rangle \in R^{j-i-1}$. Jeżeli $j - i > l + m$, to dany wierzchołek występuje we wszystkich kopiach G_h począwszy od i-tej. Podobnie występuje on również we wszystkich kopiach poprzedzających j-tą. Tak więc, jeżeli dany wierzchołek występuje w i-tej i j-tej kopii G_h tworzącej G_h^n oraz $j - i - 1 \geq l + m$, to występuje on we wszystkich kopiach G_h tworzących G_h^n.

Rozważmy dwie krawędzie o etykietach `l` i/lub `p`, należące do G_h^n i mające wspólny wierzchołek. Niech krawędzie te pochodzą z i-tej i j-tej kopii G_h tworzących G_h^n. Wówczas ich wspólny wierzchołek należy zarówno do i-tej jak i j-tej kopii G_h. Jeżeli $j - i > l + m$, to dla

$n \geq j+2m$ mielibyśmy wierzchołek stopnia wiekszego niż 3. Tak więc dla każdych dwóch krawędzi o etykietach `l` i/lub `p` należących do G_h^n, mających wspólny wierzchołek, pochodzących odpowiednio z i-tej i j-tej ($i < j$) kopii G_h mamy $j-i \leq l+m$. Podobnie pokazujemy, że jeżeli pewna krawędź o etykiecie `l` lub `p` należąca do G_h^n pochodzi z i-tej kopii G_h, jeden z jej końców jest uchwytem, lub należy do h_n, to odpowiednio mamy $i \leq l+m$, lub $i \geq n-(l+m)+1$.

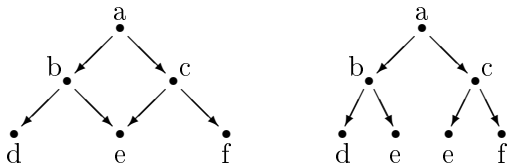
Z $\lfloor \frac{n}{2} \rfloor$-tej kopii G_h tworzącej G_h^n pochodzi pewna krawędź o etykiecie `l` lub `p`. Krawedź ta leży na pewnej ścieżce prostej łączącej ją z pewnym uchwytem G_h^n lub z pewnym wierzchołkiem należącym do h_n. Jednak długość tej ścieżki wynosi przynajmniej $\frac{\lfloor \frac{n}{2} \rfloor}{l+m}$. Tak więc G_h^n zawiera ścieżkę prostą długości rzędu $\Theta(n)$. Oczywiście żadna ścieżka prosta zawarta w G_h^n nie może mieć długości większego rzędu niż $\Theta(n)$, gdyż G_h^n zawiera $\Theta(n)$ wierzchołków. Stąd drzewa postaci $G'[h' \to G_h^n][h_i \to G'']$ mają $\Theta(n)$ wierzchołków oraz najdłuższe ścieżki proste długości rzędu $\Theta(n)$. Uzyskujemy stąd sprzeczność, gdyż w drzewach AVL długość najdłuższej ścieżki prostej w zależności od liczby wierzchołków n wynosi $\Theta(\log n)$. ■

Dzięki „rysunkowemu" charakterowi, specyfikacje zapisane w *HRG* wydają się w miarę czytelne. Dodatkowo gramatyki podmiany hiperkrawędzi dostarczają mechanizmu, który możnaby zastosować do definiowania funkcji abstrakcji. Niestety twierdzenia 1 i 2 pokazują, że zastosowanie tego formalizmu do specyfikowania wskaźnikowych struktur danych jest bardzo ograniczone.

4.5 Modalne specyfikacje wskaźnikowych struktur danych

Patrząc na wskaźnikowe struktury danych jak na grafy, nasuwa się analogia z modelami Kriepkego lub systemami tranzytywnymi. Do ich opisu wykorzystywane są logiki modalne (a w szczególności temporalne). Stąd też pojawia się naturalne pytanie o przydatność logik modalnych do specyfikowania wskaźnikowych struktur danych. Ogólna idea wykorzystania logik modalnych do specyfikacji typów danych nie jest nowa [CR97, FL90, Sza88].

Zarówno w logikach modalnych, jak i w specyfikacjach wskaźnikowych struktur danych, jednym z podstawowych pojęć jest pojęcie ścieżki. W odniesieniu do struktur wskaźnikowych ścieżka jest ciągiem rekordów połączonych wskaźnikami. W logikach modalnych ścieżka reprezentuje możliwą historię/obliczenie/ciąg stanów. Zwykle, dla prawdziwości formuły modalnej jest istotne, jakie są możliwe w danej sytuacji dalsze historie/obliczenia, a nie jest istotne np. czy dwie ścieżki wychodzące z jednego wierzchołka/świata przecinają się ponownie, czy nie. Z drugiej strony, w przypadku wskaźnikowych struktur danych, np. drzew binarnych, jest istotne, aby z korzenia drzewa do każdego wierzchołka prowadziła dokładnie jedna ścieżka. Ilustruje to następujący rysunek.



Z punktu widzenia klasycznych logik modalnych obydwie struktury są równoważne. Jednak tylko druga z nich jest poprawnie zbudowanym drzewem. Dlatego też klasyczne logiki modalne nie mają bezpośredniego zastosowania dla specyfikacji wskaźnikowych struktur danych.

Wady tej pozbawiony jest formalizm zaproponowany w [CR97] do specyfikacji dynamicznych typów danych[1]. Wersja tego formalizmu (*DDtL*), dostosowana do naszych potrzeb, jest opisana w podrozdziale 3.4. W niniejszym podrozdziale zbadamy przydatność *DDtL* do specyfikacji wskaźnikowych struktur danych.

Należy podkreślić, że choć *DDtL* ma swoje korzenie w *FOL* i logice temporalnej czasu rozgałęziającego się CTL*, to w naszym przypadku wszelkie operatory modalne nie odnoszą się do relacji następstwa w czasie, lecz do połączeń wskaźnikowych między rekordami.

Przykład 18 Deklaracja struktury danych modułu implementującego listy jednokierunkowe może wyglądać następująco (por. przykłady 2, 6 i 12):

```
TYPE
  lista = POINTER TO elem;
  elem = RECORD
    d: dane;
    nast: lista
  END;
  ppelem = POINTER TO lista;
  plista = POINTER TO lista;
```

Początki list można scharakteryzować następującą formułą:

$$head(p : \texttt{lista}) \mathrel{\hat{=}} (p\uparrow)\downarrow \wedge \forall_{q:\texttt{lista}}(q\uparrow.\texttt{nast} \neq p) \ .$$

Niezmiennik list jednokierunkowych może być opisany za pomocą następujących własności:

- zewnętrzne wskaźniki wskazują wyłącznie na początki list

$$\forall_{p:\texttt{plista}}\,((p\uparrow)\downarrow \ \wedge p\uparrow \neq \texttt{NIL} \Rightarrow head(p\uparrow)) \ \ ,$$

- każda lista jest zakończona wskaźnikiem `NIL` (por. definicja 27)

$$\forall_{p:\texttt{lista}}(head(p) \Rightarrow \triangle_{p,[]\uparrow.\texttt{nast}}\Diamond[\lambda x.\ x = \texttt{NIL}]) \ \ ,$$

- każdy alokowany rekord jest na pewnej liście

$$\forall_{p:\texttt{lista}}(p\uparrow)\downarrow \Rightarrow \exists_{q:\texttt{plista}}(head(q\uparrow) \wedge \bigtriangledown_{q\uparrow,[]\uparrow.\texttt{nast}}\Diamond[\lambda x.\ x = p]) \ \ ,$$

- do każdego węzła prowadzi co najwyżej jedna krawędź

$$\forall_{p,q,r:\texttt{lista}}(p\uparrow.\texttt{nast} = r \wedge q\uparrow.\texttt{nast} = r \wedge r \neq \texttt{NIL} \Rightarrow p = q) \ .$$

Oznaczmy przez *list* koniunkcję powyższych warunków. Rozważmy program zamieniający wszystkie wartości x na liście p na wartości y. Warunek wstępny tego programu możemy opisać następująco:

$$list \wedge head(p) \wedge x\downarrow \wedge y\downarrow$$

[1] Przypomnijmy, że tutaj określenie „dynamiczne typy danych" nie oznacza wskaźnikowych struktur danych, lecz struktury mogące ulegać zmianom.

a warunek końcowy następująco:

$$\nabla_{\mathcal{V}p,[]\uparrow.\texttt{nast}} \begin{pmatrix} \Box[\lambda q.\ q\backprime\uparrow.\texttt{d} = \backprime x \Rightarrow q\uparrow.\texttt{d} = \backprime y] \wedge \\ \forall_{q:\texttt{lista}}((q\backprime\uparrow.\texttt{d} \neq \backprime x \vee \Box[\lambda r.\ r \neq q]) \Rightarrow q\uparrow \equiv q\backprime\uparrow) \wedge \\ \forall_{r:\uparrow\texttt{data}}(\forall_{q:\texttt{lista}}\ q|\texttt{d} \neq r) \Rightarrow r\uparrow \equiv r\backprime\uparrow \end{pmatrix} \wedge \psi$$

gdzie ψ jest formułą mówiącą, że wartości zmiennych innych typów niż `elem` i `data` nie zmieniają się. □

Przykład 19 Moduł implementujący cykliczne listy dwukierunkowe może mieć następujące deklaracje typów (por. przykłady 7, 13 i 16):

```
TYPE
  lista = POINTER TO elem;
  elem = RECORD
    d: dane;
    pop, nast: lista
  END;
ppelem = POINTER TO lista;
plista = POINTER TO lista;
```

Następujący niezmiennik struktury danych określa dopuszczalne kształty list oraz wyraża fakt, że każda lista jest wskazywana przez dokładnie jeden zewnętrzny wskaźnik i vice versa:

$$\forall_{p:\texttt{lista}} \left((p\uparrow)\downarrow \Rightarrow \begin{pmatrix} p\uparrow.\texttt{nast}\uparrow.\texttt{pop} = p\ \wedge p\uparrow.\texttt{pop}\uparrow.\texttt{nast} = p \wedge \\ \triangle_{p,[]\uparrow.\texttt{nast}}\mathcal{O}\Diamond[\lambda x.\ x = p] \wedge \triangle_{p,[]\uparrow.\texttt{pop}}\mathcal{O}\Diamond[\lambda x.\ x = p] \wedge \\ \exists_{q:\texttt{plista}}\ \triangle_{q\uparrow,[]\uparrow.\texttt{nast}} \begin{pmatrix} \Diamond[\lambda x.\ x = p] \wedge \\ \forall_{r:\texttt{plista}}(r \neq q \Rightarrow\ \Box[\lambda x.\ x \neq r\uparrow]) \end{pmatrix} \end{pmatrix} \right) \wedge$$
$$\forall_{p:\texttt{plista}}\left((p\uparrow)\downarrow \wedge p \neq \texttt{NIL} \Rightarrow (p\uparrow\uparrow)\downarrow\right).$$

□

Przykład 20 Korzystając z list jednokierunkowych możemy wyspecyfikować wskaźnikową implementację grafów skierowanych (por. przykład 9). Wierzchołki grafu przechowujemy na liście jednokierunkowej. Z każdym wierzchołkiem mamy związaną jego listę incydencji — jednokierunkową listę wskaźników do tych wierzchołków, do których z danego wierzchołka prowadzą krawędzie. Deklaracje typów dla takiej struktury danych mają następującą postać:

```
TYPE
  graf = POINTER TO wierzchołek;
  sąsiedzi = POINTER TO krawędź;
  wierzchołek = RECORD
    s: sąsziedzi;
    nast: graf
  END;
  krawędź = RECORD
    w: graf;
    nast: sąsiedzi
  END;
ppwierzchołek = POINTER TO graf;
pgraf = POINTER TO graf;
```

Niezmiennik takiej struktury danych powinien wyrażać następujące własności:

- wierzchołki każdego grafu tworzą poprawne listy jednokierunkowe,
- krawędzie każdej listy incydencji tworzą poprawne listy jednokierunkowe,
- każda krawędź prowadzi do wierzchołka z tego samego grafu.

Podobnie jak w przykładzie 18 oznaczmy przez $head_w$ i $head_k$ następujące formuły:

- $head_w(p : \texttt{graf}) \mathrel{\hat{\equiv}} (p\uparrow)\downarrow \wedge \forall_{q:\texttt{graf}}\,(q\uparrow.\texttt{nast} \neq p)$,
- $head_k(p : \texttt{sąsiedzi}) \mathrel{\hat{\equiv}} (p\uparrow)\downarrow \wedge \forall_{q:\texttt{sąsiedzi}}\,(q\uparrow.\texttt{nast} \neq p)$,

mówiące, że dany wskaźnik wskazuje na pierwszy element listy, odpowiednio, wierzchołków lub krawędzi. Niezmiennik struktury danych grafu możemy zapisać następująco:

$$\forall_{p:\texttt{pgraf}}\,((p\uparrow)\downarrow \wedge\, p\uparrow \neq \texttt{NIL} \Rightarrow head_w(p\uparrow)) \wedge$$

$$\forall_{p:\texttt{graf}}\,\big(head_w(p) \Rightarrow \triangle_{p,[]\uparrow.\texttt{nast}}\Diamond[\lambda x.\ x = \texttt{NIL}]\big) \wedge$$

$$\forall_{p:\texttt{graf}}\,\big((p\uparrow)\downarrow \Rightarrow \exists_{q:\texttt{pgraf}}\,\big(head_w(q\uparrow) \wedge \triangledown_{q\uparrow,[]\uparrow.\texttt{nast}}\Diamond[\lambda x.\ x = p]\big)\big)\big) \wedge$$

$$\forall_{p,q,r:\texttt{graf}}\,(p\uparrow.\texttt{nast} = r \wedge q\uparrow.\texttt{nast} = r \wedge r \neq \texttt{NIL} \Rightarrow p = q) \wedge$$

$$\forall_{p:\texttt{graf}}\,((p\uparrow)\downarrow \wedge\, p\uparrow.\texttt{s} \neq \texttt{NIL}\ \Rightarrow head_k(p\uparrow)) \wedge$$

$$\forall_{p:\texttt{sąsiedzi}}\,\big(head_k(p) \Rightarrow \triangle_{p,[]\uparrow.\texttt{nast}}\Diamond[\lambda x.\ x = \texttt{NIL}]\big) \wedge$$

$$\forall_{p:\texttt{sąsiedzi}}\,\big((p\uparrow)\downarrow \Rightarrow \exists_{q:\texttt{graf}}\,\big(head_k(q\uparrow.\texttt{s}) \wedge \triangledown_{q\uparrow.\texttt{s},[]\uparrow.\texttt{nast}}\Diamond[\lambda x.\ x = p]\big)\big)\big) \wedge$$

$$\forall_{p,q,r:\texttt{sąsiedzi}}\,(p\uparrow.\texttt{nast} = r \wedge q\uparrow.\texttt{nast} = r \wedge r \neq \texttt{NIL} \Rightarrow p = q) \wedge$$

$$\forall_{p:\texttt{pgraf}}\left((p\uparrow)\downarrow\Rightarrow \triangledown_{p\uparrow,[]\uparrow.\texttt{nast}}\Box\left[\begin{array}{l}\lambda q.\ q \neq \texttt{NIL} \Rightarrow\\ \triangledown_{q\uparrow.\texttt{s},[]\uparrow.\texttt{nast}}\Box\left[\begin{array}{l}\lambda r.\ r \neq \texttt{NIL} \Rightarrow\\ \triangledown_{p\uparrow,[]\uparrow.\texttt{nast}}\Diamond\left[\lambda s.\ s = r\uparrow.\texttt{w}\right]\end{array}\right]\end{array}\right]\right)\quad .$$

□

Przykład 21 Deklaracja drzew *find-union* (używanych do reprezentowania rozłącznych zbiorów elementów) może być bardzo prosta (por. przykłady 8 i 14):

```
TYPE
  node = POINTER TO node;
```

Niezmiennik tej struktury danych również jest prosty, gdyż wyraża on jedynie fakt, że każda ścieżka zaczynająca się w dowolnym elemencie struktury prowadzi do elementu wskazującego na samego siebie:

$$\forall_{p:\mathtt{node}} \left((p\uparrow)\downarrow \Rightarrow \triangle_{p,[]\uparrow}\Diamond[\lambda x.\ x = x\uparrow]\right) \ .$$

Poniżej opisujemy operację *find*, która dla zadanego wskaźnika p do elementu struktury danych zmienia jego wartość tak, aby wskazywał na reprezentanta zbioru, do którego należy dany element, a przy okazji dokonuje kompresji ścieżki prowadzącej do tego reprezentanta. W warunku wstępnym tej operacji oprócz niezmiennika danych wymagamy, aby początkowa wartość p wskazywała na istniejący element: $(p\uparrow)\downarrow$. Warunek końcowy ma następującą postać:

$$p`\uparrow = p \wedge p\uparrow = p \wedge$$

$$\triangle_{`p,[]`\uparrow}(\Diamond[\lambda x.\ x = p] \wedge \Box[\lambda x.\ x\uparrow = p]) \wedge$$

$$\forall_{q:\mathtt{node}}(\triangle_{`p,[]`\uparrow}\Box[\lambda x.\ x \neq q] \Rightarrow q`\uparrow = q\uparrow) \ .$$

□

Przykład 22 Deklaracje typów dla drzew binarnych mogą mieć następującą postać (por. przyklady 10, 15 i 17):

```
TYPE
  tree = POINTER TO node;
  node = RECORD
    d: data;
    l, r: tree
  END;
  ppnode = POINTER TO tree;
  ptree = POINTER TO tree;
```

gdzie pole `d` zawiera wartość przechowywaną w węźle drzewa, a pola `l` i `r` są dowiązaniami do lewego i prawego poddrzewa. Niezmiennik drzew binarnych można opisać w następujący sposób:

$$\forall_{p:\mathtt{ptree}}\left((p\uparrow)\downarrow \wedge p\uparrow \neq \mathtt{NIL} \Rightarrow \forall_{q:\mathtt{tree}}\left(q\uparrow.\mathtt{l} \neq p\uparrow \wedge q\uparrow.\mathtt{p} \neq p\uparrow\right)\right) \wedge$$

$$\forall_{p:\mathtt{tree}}\left((p\uparrow)\downarrow \Rightarrow \exists_{q:\mathtt{ptree}}\ \triangledown_{q\uparrow,[]\uparrow.\mathtt{l},[]\uparrow.\mathtt{p}}\Diamond[\lambda x.\ x = p]\right) \wedge$$

$$\forall_{p:\mathtt{tree}}\left((p\uparrow)\downarrow \wedge p\uparrow.\mathtt{l} = p\uparrow.\mathtt{p} \Rightarrow p\uparrow.\mathtt{l} = \mathtt{NIL}\right) \wedge$$

$$\forall_{p,q,r:\mathtt{tree}}\left((p\uparrow.\mathtt{l} = r \vee p\uparrow.\mathtt{p} = r) \wedge (q\uparrow.\mathtt{l} = r \vee q\uparrow.\mathtt{p} = r) \wedge r \neq \mathtt{NIL} \Rightarrow p = q\right) \wedge$$

$$\forall_{p:\mathtt{tree}}\left((p\uparrow)\downarrow \Rightarrow \triangle_{p,[]\uparrow.\mathtt{l},[]\uparrow.\mathtt{p}}\Diamond[\lambda x.\ x = \mathtt{NIL}]\right) \ .$$

Niezmiennik drzew BST można wyrazić wzmacniając niezmiennik drzew binarnych następującą formułą (zakładając, że porządek na wartościach typu `data` opisuje relacja $\leq$):

$$\forall_{q:\mathtt{tree}}((q\uparrow.\mathtt{l}\uparrow)\downarrow \Rightarrow \triangle_{q\uparrow.\mathtt{l}\uparrow,[].\mathtt{l}\uparrow,[].\mathtt{p}\uparrow}\Box[\lambda p.\ p.\mathtt{x} \leq q\uparrow.\mathtt{x}]) \wedge$$
$$\forall_{q:\mathtt{tree}}((q\uparrow.\mathtt{p}\uparrow)\downarrow \Rightarrow \triangle_{q\uparrow.\mathtt{p}\uparrow,[].\mathtt{l}\uparrow,[].\mathtt{p}\uparrow}\Box[\lambda p.\ q\uparrow.\mathtt{x} \leq p.\mathtt{x}])\ .$$

Analogicznie, niezmiennik kopców binarnych (o minimalnej wartości przechowywanej w korzeniu) możemy uzyskać wzmacniając niezmiennik drzew binarnych następującą formułą:

$$\forall_{q:\mathtt{tree}}\,(((q\uparrow.\mathtt{l}\uparrow)\downarrow \Rightarrow q\uparrow.\mathtt{l}\uparrow.\mathtt{d} \geq q\uparrow.\mathtt{d}) \wedge ((q\uparrow.\mathtt{r}\uparrow)\downarrow \Rightarrow q\uparrow.\mathtt{r}\uparrow.\mathtt{d} \geq q\uparrow.\mathtt{d}))$$

□

Niestety okazuje się, że nie wszystkie wskaźnikowe struktury danych można wyspecyfikować w *DDtL*. Przykładem wskaźnikowych struktur danych, których nie można wyspecyfikować są różnego rodzaju drzewa zrównoważone, np. drzewa AVL. Wynika to stąd, że nie jesteśmy w stanie porównywać długości (rozłącznych) ścieżek.

Fakt 6 Rozważmy następujące deklaracje typów (por. fakt 5):

```
TYPE
  AVL = POINTER TO node;
  node = RECORD
    d: data;
    lw, pw: BOOLEAN;
    l, r: AVL
  END;
  ppnode = POINTER TO AVL;
  pAVL = POINTER TO AVL;
```

gdzie pole `d` reprezentuje wartość przechowywaną w danym węźle, pola `lw` i `pw` są równe *true* odpowiednio, gdy prawe/lewe poddrzewo ma większą wysokość, a pola `l` i `r` są dowiązaniami do lewego i prawego poddrzewa. Dla powyższych deklaracji nie istnieje formuła *DDtL* będąca niezmiennikiem drzew AVL (z powtórzeniami). □

W dowodzie powyższego faktu będzie nam pomocny następujący lemat.

Lemat 4 Niech Σ_T bedzie sygnaturą etykietowanego drzewa binarnego. Dla każdej formuły *DDtL* $\varphi \in D_{\Sigma_T}(X)$ istnieje taka formuła $\psi \in MSOL_{\Sigma_T}(X)$, że dla każdego skończonego etykietowanego Σ_T-drzewa binarnego A i wartościowania $v : X \to A$ mamy $A \models_v \varphi$ wtw., gdy $A \models_v \psi$. □

Dowód Pokażemy, w jaki sposób przetłumaczyć formułę *DDtL* φ na odpowiadającą jej formułę *MSOL* ψ. Dla uproszczenia zakładamy, że opisy krewędzi występujące w φ nie zawierają termu [] (czyli, że każda krawędź prowadzi w dół drzewa). W przeciwnym przypadku możemy zastąpić φ formułą jej równoważną, spełniającą ten warunek.

Zauważmy, że przy powyższych założeniach ścieżki w drzewach binarnych są ścieżkami prostymi i możemy je reprezentować za pomocą zbiorów wierzchołków. Przypomnijmy (por. p. 4.1), że jeżeli jakaś relacja binarna (np. połączenie wierzchołków krawędzią) jest wyrażalna w *MSOL*, to jej domknięcie zwrotno-tranzytywne też jest wyrażalne w *MSOL* [Cou90]. Fakt, iż zbiór wierzchołków P tworzy ścieżkę o początku t i krawędziach zgodnych z opisem krawędzi e, możemy opisać za pomocą następujących warunków (wyrażalnych w *MSOL*):

- wszystkie wierzchołki należące do P są osiągalne z t (poprzez krawędzie opisane w e), i to poprzez wierzchołki należące do P,
- każdy wierzchołek należący do P jest połączony krawędzią (opisaną w e) z co najwyżej jednym wierzchołkiem należącym do P,
- z żadnego wierzchołka należącego do P nie można przejść do t,
- P jest maksymalnym zbiorem wierzchołków spełniającym powyższe warunki.

Oznaczmy przez $Path(P, t, e)$ formułę $MSOL$ będącą koniunkcją powyższych warunków. Ponadto oznaczmy przez $x \leq_P y$ formułę $MSOL$ mówiącą, że x i y należą do ścieżki P, przy czym x znajduje się na tej ścieżce wcześniej niż y lub $x = y$.

Tłumaczenie formuły $\varphi \in D_{\Sigma_T}(X)$ na odpowiadającą jej formułę $\psi \in MSOL_{\Sigma_T}(X, \emptyset)$ definiujemy rekurencyjnie ze względu na budowę φ. Przy tym formuły $MSOL$ będące tłumaczeniami podformuł φ należą do $MSOL_{\Sigma_T}(X, Y)$ dla określonych zbiorów zmiennych drugiego rzędu Y, a tłumaczenie formuł ścieżkowych wchodzących w skład φ definiujemy dla danej nazwy zmiennej drugiego rzędu $P \in Y$ reprezentującej zbiór wierzchołków tworzących ścieżkę:

- jeżeli φ jest postaci $t_1 = t_2$, to $\psi = \varphi$,
- jeżeli φ jest postaci $\varphi_1 \Rightarrow \varphi_2$, to ψ jest postaci $\psi_1 \Rightarrow \psi_2$, gdzie ψ_1 jest wynikiem tłumaczenia φ_1, a ψ_2 jest wynikiem tłumaczenia φ_2; podobnie dla pozostałych operacji boolowskich,
- jeżeli φ jest postaci $\forall_{x:d}(\varphi_1)$ $(\exists_{x:d}(\varphi_1))$, to ψ jest postaci $\forall_{x:d}(\psi_1)$ $(\exists_{x:d}(\psi_1))$, gdzie ψ_1 jest wynikiem tłumaczenia φ_1,
- jeżeli φ jest postaci $\triangle_{t,e_1,\ldots,e_k}\pi$ $(\bigtriangledown_{t,e_1,\ldots,e_k}\pi)$, $P \notin X$, to ψ jest postaci

$$\forall_{P:d}(Path(P, t, \{e_1, \ldots, e_k\}) \Rightarrow \psi_1)$$

$$(\exists_{P:d}(Path(P, t, \{e_1, \ldots, e_k\}) \wedge \psi_1)) \ ,$$

  gdzie $P \notin Y$, $\psi_1 \in MSOL_{\Sigma_T}(X, Y \cup \{P : d\})$ jest wynikiem tłumaczenia π dla zmiennej predykatowej P,
- jeżeli φ jest postaci $\varphi_1 \mathcal{U} \varphi_2$ i dana jest zmienna predykatowa P, to ψ jest postaci

$$\exists_{x,Q:d} \begin{pmatrix} P(x) \wedge \exists_{y:d}(P(y) \wedge y \neq x \wedge y \leq_P x) \wedge \forall_{y:d}(Q(y) \Leftrightarrow P(y) \wedge x \leq_P y) \wedge \psi_2 \wedge \\ \forall_{y:d}(P(y) \wedge y \leq_P x \wedge y \neq x \Rightarrow \exists_{R:d}(\forall_{z:d}(R(z) \Leftrightarrow P(z) \wedge y \leq_P z) \wedge \psi_1)) \end{pmatrix} \ ,$$

  gdzie $x, y, z \notin X$, $Q, R \notin Y$, $\psi_1 \in MSOL_{\Sigma_T}(X, Y \cup \{R : d\})$ jest wynikiem tłumaczenia φ_1 dla zmiennej predykatowej R, a $\psi_2 \in MSOL_{\Sigma_T}(X, Y \cup \{Q : d\})$ jest wynikiem tłumaczenia φ_2 dla zmiennej predykatowej Q; analogicznie postępujemy dla pozostałych operatorów ścieżkowych,
- jeżeli φ jest postaci $[\lambda x.\varphi_1]$ i dana jest zmienna predykatowa P, to ψ jest postaci $\exists_{x:d}(P(x) \wedge \forall_{y:d}(y \leq_P x \Rightarrow x = y) \wedge \psi_1)$, gdzie ψ_1 jest wynikiem tłumaczenia φ_1.

Jak łatwo sprawdzić, dowodzony lemat wynika bezpośrednio z postaci tłumaczenia. ■

Dowód faktu 6 Dowód przebiega nie wprost, podobnie do dowodu faktu 5. Załóżmy przeciwnie, że istnieje formuła *DDtL* będąca niezmiennikiem struktury danych dla drzew AVL.

Wybór drzew AVL z powtórzeniami pozwala nam ominąć przypadek, gdy zbiór wartości typu `data` jest skończony, a przez to i zbiór drzew AVL bez powtórzeń jest skończony. Skoncentrujemy się przy tym na samych kształtach drzew AVL pomijając wartości pól `d`. Zauważmy, że jeżeli w dowolnym drzewie AVL zmienimy wartości pól `d` we wszystkich węzłach na tę samą, ustaloną wartość, to otrzymamy poprawne drzewo AVL (z samymi powtórzeniami). Zauważmy ponadto, że pola `lg` i `pg` są typu `BOOLEAN`, a więc mogą być reprezentowane za pomocą dwóch funkcji etykietujących. Niech Σ_T będzie sygnaturą etykietowanych drzew binarnych zawierającą dwie funkcje etykietujące węzły: $\texttt{lg},\texttt{pg} : v \to \texttt{BOOLEAN}$.

Skoro istnieje formuła *DDtL* będąca niezmiennikiem drzew AVL, to istnieje również formuła *DDtL*, nad sygnaturą Σ_T, bez zmiennych wolnych, charakteryzująca skończone etykietowane Σ_T-drzewa binarne, będące kształtami drzew AVL. Z lematu 4 otrzymujemy, że istnieje formuła *MSOL* charakteryzująca drzewa binarne będące kształtami drzew AVL. Z lematu 1 otrzymujemy sprzeczność. ■

Z lematu 4 oraz rozstrzygalności spełnialności formuł *MSOL* dla drzew binarnych (por. np. [Tho90, Tho97]) wynika następujący fakt:

Fakt 7 Spełnialność formuł *DDtL* dla skończonych etykietowanych drzew binarnych jest rozstrzygalna. □

Fakt ten można uogólnić na drzewa nieskończone.

Jak pokazują przykłady zawarte w tym podrozdziale, *DDtL* jest formalizmem pozwalającym na względnie czytelne i zwięzłe specyfikowanie wybranych przez nas wskaźnikowych struktur danych. Fakt 6 wskazuje jednak na ograniczenia w stosowaniu *DDtL* do tego celu.

4.6 Logika wielościeżkowa

W podrozdziale tym przedstawiamy propozycję logiki modalnej służącej do specyfikowania wskaźnikowych struktur danych, pozwalającej na wyspecyfikowanie szerszej klasy struktur danych niż *DDtL*. Logikę tę nazywamy wielościeżkową, w skrócie *MPL* (ang. *multi-path logic*).

Niech $\Sigma = \langle S_\Sigma, F_\Sigma \rangle$ będzie ustaloną sygnaturą, $X = \langle X_s \rangle_{s \in S_\Sigma}$ będzie zbiorem nazw zmiennych. Intuicyjnie, zmienne danego rodzaju reprezentują ścieżki złożone z elementów tego rodzaju, przy czym w danej chwili wartością zmiennej jest pierwszy element odpowiadającej jej ścieżki. Zbiór formuł wielościeżkowych $MPL_\Sigma(X)$ definiujemy rekurencyjnie.

Def. 30 $MPL_\Sigma(X)$ to najmniejszy taki zbiór, że:

- jeżeli $s \in S_\Sigma$, $t_1, t_2 \in T_\Sigma(X)_s$, to $t_1 = t_2 \in MPL_\Sigma(X)$,
- jeżeli $\varphi_1, \varphi_2 \in MPL_\Sigma(X)$, to: $\varphi_1 \Rightarrow \varphi_2, \neg\varphi_1 \in MPL_\Sigma(X)$,
- jeżeli $s \in S_\Sigma$, $\varphi \in MPL_\Sigma(X \cup \{x : s\})$, to $\forall_{x:s}(\varphi), \exists_{x:s}(\varphi) \in MPL_\Sigma(X)$,
- jeżeli $s \in S_\Sigma$, $t \in T_\Sigma(X)_s$, $e_1, \ldots, e_n \in T_\Sigma(X \cup \{[] : s\})_s$, $\varphi_1 \in MPL_\Sigma(X \cup \{x : s\})$, to $\triangle_{x:s,t,e_1,\ldots,e_n}(\varphi_1), \triangledown_{x:s,t,e_1,\ldots,e_n}(\varphi_1) \in MPL_\Sigma(X)$,
- jeżeli $\varphi_1, \varphi_2 \in MPL_\Sigma(X)$, to: $\varphi_1 \mathcal{U} \varphi_2 \in MPL_\Sigma(X)$.

□

Formuły wielościeżkowe możemy budować używając operatorów pierwszego rzędu. Formuła $\triangle_{x:s,t,e_1,\ldots,e_k}$ reprezentuje uniwersalną, a $\triangledown_{x:s,t,e_1,\ldots,e_k}$ egzystencjalną kwantyfikację ścieżek o początku w t i krawędziach opisanych przez $e_1,\ldots,e_k$. Odpowiadają one modalnościom „konieczne" i „możliwe". Operator ścieżkowy $\mathcal{U}$ (*„until"*) jest interpretowany wzdłuż ścieżek przypisanych zmiennym.

Def. 31 Niech A będzie Σ-algebrą częściową, X będzie zbiorem nazw zmiennych. Ścieżkowym wartościowaniem zmiennych z X w A będziemy nazywać każdą taką fukcję $v : X \to |A|^+ \cup |A|^\omega$, że $v(x) \in |A|_s^+ \cup |A|_s^\omega$ dla $x \in X_s$.

Przez $v_{|i}$ oznaczamy następujące ścieżkowe wartościowanie zmiennych: $v_{|i}(x) \mathrel{\hat{=}} v(x)_{|i}$. Mówimy, że $v_{|i}$ jest określone, gdy $v_{|i}(x)$ jest określone dla wszystkich $x \in X$.

Jeśli v jest ścieżkowym wartościowaniem zmiennych, to przez $B(v)$ oznaczamy następujące wartościowanie zmiennych: $B(v)(x) = B(v(x))$.

Jeśli $v_1 : X \to |A|$ jest (zwykłym) wartościowaniem zmiennych, to ścieżkowym rozszerzeniem v_1 nazywamy ścieżkowe wartościowanie $\tilde{v_1}$ postaci $\tilde{v_1}(x) \mathrel{\hat{=}} \langle v_1(x), v_1(x), \ldots\rangle$ (dla $x \in X$). □

Formuły wielościeżkowe interpretujemy przy zadanym ścieżkowym wartościowaniach zmiennych. Intuicyjnie, zmienne przebiegają kolejne elementy przypisanych im ścieżek. Semantykę formuł wielościeżkowych definiujemy rekurencyjnie ze względu na ich budowę.

Def. 32 Niech $\varphi \in MPL_\Sigma(X)$, A będzie Σ-algebrą częściową, v będzie ścieżkowym wartościowaniem zmiennych z X w A. Mówimy, że formuła wielościeżkowa φ jest spełniona w A dla v, co oznaczamy przez $A \models_v \varphi$, wtw., gdy:

- $A \models_v t_1 = t_2$ wtw., gdy $B(v)^A(t_1) = B(v)^A(t_2)$,
- $A \models_v \varphi_1 \Rightarrow \varphi_2$ wtw., gdy jeżeli $A \models_v \varphi_1$, to $A \models_v \varphi_2$,
- $A \models_v \neg\varphi_1$ wtw., gdy $A \not\models_v \varphi_1$,
- $A \models_v \forall_{x:s}\varphi_1$ ($A \models_v \exists_{x:s}\varphi_1$) wtw., gdy dla każdego (dla pewnego) $a \in |A|_s$ mamy $A \models_{v[x\to\langle a,a,\ldots\rangle]} \varphi_1$,
- $A \models_v \triangle_{x:s,t,e_1,\ldots,e_n}\varphi_1$ ($A \models_v \triangledown_{x:s,t,e_1,\ldots,e_n}\varphi_1$) wtw., gdy $B(v)^A(t)$ jest określone i dla każdej (dla pewnej) ścieżki $p \in Path(A, s, B(v), \{e_1,\ldots,e_n\})$ o początku $B(p) = B(v)^A(t)$ mamy $A \models_{v[x\to p]} \varphi_1$,
- $A \models_v \varphi_1 \mathcal{U} \varphi_2$ wtw., gdy istnieje takie $j > 0$, że $v_{|j}$ jest określone, dla każdego $0 < i < j$ zachodzi $A \models_{v_{|i}} \varphi_1$ oraz $A \models_{v_{|j}} \varphi_2$.

Przez $A \models \varphi$ oznaczamy fakt, że dla każdego (zwykłego) wartościowania $v_1 : X \to |A|$ zachodzi $A \models_{\tilde{v_1}} \varphi$. □

W powyższej definicji $B(v)$ jest (zwykłym) wartościowaniem zmiennych, a $B(v)^A$ jego rozszerzeniem na termy. Intuicyjnie $B(v)^A(t)$ reprezentuje „aktualną" wartość termu t.

Operatory boolowskie $\wedge$, $\vee$ i $\Leftrightarrow$ definiujemy w standardowy sposób za pomocą $\Rightarrow$ i $\neg$. Podobnie, za pomocą $\mathcal{U}$ definiujemy operatory ścieżkowe $\Box$ (dla każdej odległości od początku ścieżek), $\Diamond$ (dla pewnej odległości od początku ścieżek), oraz $\mathcal{O}$ (począwszy od kolejnych elementów ścieżek). Zwróćmy uwagę, że operatory ścieżkowe sięgają „tak daleko" jaka jest długość najkrótszej ścieżki w wartościowaniu.

Nie proponujemy tutaj żadnego systemu dowodowego dla *MPL*. Wynika to stąd, że nie istnieje skończony pełny system dowodowy dla *MPL*. Jak łatwo zauważyć, dla zadanej maszyny Turinga, za pomocą formuły *MPL* można opisać strukturę danych reprezentującą kończące się pomyślnie obliczenie tej maszyny. Tak więc problem stopu sprowadza się do problemu spełnialności formuł *MPL*. Gdyby istniał skończony system dowodowy dla *MPL*, to problem stopu byłby rozstrzygalny, co jak wiadamo nie ma miejsca. Argument ten dotyczy zresztą wszystkich logik nadających się do specyfikowania wskaźnikowych struktur danych.

Następujący fakt pozwala nam traktować *MPL* jako rozszerzenie *DDtL*.

Fakt 8 Dla każdej formuły $\varphi \in D_\Sigma(X)$ istnieje taka formuła $\psi \in MPL_\Sigma(X)$, że $A \models_v \varphi$ wtw., gdy $A \models_{\tilde{v}} \psi$. □

Dowód Niech p będzie ustaloną nazwą zmiennej, taką, która nie pojawia się w φ. Formułę ψ definiujemy rekurencyjnie ze względu na strukture φ, następująco:

- jeżeli φ jest postaci $t_1 = t_2$, to $\psi = \varphi$,
- jeżeli φ jest postai $\varphi_1 \Rightarrow \varphi_2$, to ψ jest postaci $\psi_1 \Rightarrow \psi_2$, gdzie formule φ_1 odpowiada ψ_1, a φ_2 odpowiada ψ_2,
- jeżeli φ jest postaci $\neg\varphi_1$, to ψ jest postaci $\neg\psi_1$, gdzie formule φ_1 odpowiada formuła ψ_1,
- jeżeli φ jest postaci $\forall_{x:s}(\varphi_1)$ $(\exists_{x:s}(\varphi_1))$, to ψ jest postaci $\forall_{x:s}(\psi_1)$ $(\exists_{x:s}(\psi_1))$, gdzie formule φ_1 odpowiada ψ_1,
- jeżeli φ jest postaci $\triangle_{t,e_1,\ldots,e_n}(\varphi_1)$ $(\triangledown_{t,e_1,\ldots,e_n}(\varphi_1))$, s jest rodzajem t, to ψ jest postaci $\triangle_{p:s,t,e_1,\ldots,e_n}(\psi_1)$ $(\triangledown_{p:s,t,e_1,\ldots,e_n}(\psi_1))$, gdzie formule φ_1 odpowiada ψ_1,
- jeżeli φ jest postaci $[\lambda x.\varphi_1]$, to ψ jest postaci $\exists_{x:s}(x = p \wedge \psi_1)$, gdzie s jest rodzajem x, a formule φ_1 odpowiada ψ_1,
- jeżeli φ jest postaci $\varphi_1 \mathcal{U} \varphi_2$, to ψ jest postaci $\psi_1 \mathcal{U} \psi_2$, gdzie formule φ_1 odpowiada ψ_1, a φ_2 odpowiada ψ_2.

Jak łatwo sprawdzić, powyższe tłumaczenie zachowuje relację spełniania formuł. ■

Z powyższego faktu wynika, że każdą wskaźnikową strukturę danych, którą można wyspecyfikować w *DDtL* można również wyspecyfikować w *MPL*. Przy tym opisane wyżej tłumaczenie nie zmniejsza czytelności specyfikacji. Ilustruje to następujący przykład.

Przykład 23 Cykliczne listy dwukierunkowe możemy wyspecyfikować analogicznie jak w *DDtL*, por. przykład 19 (a także przykłady 7, 13 i 16). Deklaracje typów nie ulegają zmianie:

```
TYPE
  lista = POINTER TO elem;
  elem = RECORD
    d: dane;
    pop, nast: lista
  END;
ppelem = POINTER TO lista;
plista = POINTER TO lista;
```

Następujący niezmiennik struktury danych określa dopuszczalne kształty list oraz wyraża fakt, że każda lista jest wskazywana przez dokładnie jeden zewnętrzny wskaźnik i vice versa:

$$\forall_{p:\texttt{lista}} \left((p\uparrow)\downarrow \Rightarrow \begin{pmatrix} p\uparrow.\texttt{nast}\uparrow.\texttt{pop} = p \ \wedge p\uparrow.\texttt{pop}\uparrow.\texttt{nast} = p \wedge \\ \triangle_{q,p,[]\uparrow.\texttt{nast}} \mathcal{O}\Diamond(q = p) \wedge \triangle_{q,p,[]\uparrow.\texttt{pop}} \mathcal{O}\Diamond(q = p) \wedge \\ \exists_{q:\texttt{plista}} \triangle_{s,q\uparrow,[]\uparrow.\texttt{nast}} \begin{pmatrix} \Diamond(s = p) \wedge \\ \forall_{r:\texttt{plista}} (r \neq q \Rightarrow \ \Box(s \neq r\uparrow)) \end{pmatrix} \end{pmatrix} \right) .$$
$$\forall_{p:\texttt{plista}} \left((p\uparrow)\downarrow \wedge p \neq \texttt{NIL} \Rightarrow (p\uparrow\uparrow)\downarrow \right)$$

□

Poniższy przykład pokazuje, jak można w logice wielościeżkowej wyspecyfikować niezmiennik drzew AVL. Istotnym elementem jest tu możliwość równoczesnego przebiegania dwóch ścieżek. Zwróćmy przy tym uwagę na to, że poniższe specyfikacje są czytelne w takim samym stopniu jak ich odpowiedniki zapisane w *DDtL*.

Przykład 24 Deklaracje typów drzew AVL mogą mieć następującą postać (por. przyklady 11, 15 i 22):

```
TYPE
  tree = POINTER TO node;
  node = RECORD
    d: data;
    lw, pw: BOOLEAN;
    l, r: AVL
  END;
  ppnode = POINTER TO tree;
  ptree = POINTER TO tree;
```

Niezmiennik drzew binarnych (por. przykład 22) może mieć postać:

$$\forall_{p:\texttt{ptree}} \left((p\uparrow\uparrow)\downarrow \wedge p\uparrow \neq \texttt{NIL} \Rightarrow \forall_{q:\texttt{tree}} \left(q\uparrow.\texttt{l} \neq p\uparrow \wedge q\uparrow.\texttt{p} \neq p\uparrow \right) \right) \wedge$$

$$\forall_{p:\texttt{tree}} \left((p\uparrow)\downarrow \Rightarrow \exists_{q:\texttt{ptree}} \bigtriangledown_{r:\texttt{tree},q\uparrow,[]\uparrow.\texttt{l},[]\uparrow.\texttt{p}} \Diamond p = r \right) \wedge$$

$$\forall_{p:\texttt{tree}} \left((p\uparrow)\downarrow \wedge p\uparrow.\texttt{l} = p\uparrow.\texttt{p} \Rightarrow p\uparrow.\texttt{l} = \texttt{NIL} \right) \wedge$$

$$\forall_{p,q,r:\texttt{tree}} \left((p\uparrow.\texttt{l} = r \vee p\uparrow.\texttt{p} = r) \wedge (q\uparrow.\texttt{l} = r \vee q\uparrow.\texttt{p} = r) \wedge r \neq \texttt{NIL} \Rightarrow p = q \right) \wedge$$

$$\forall_{p:\texttt{tree}} \left((p\uparrow)\downarrow \Rightarrow \triangle_{q:\texttt{tree},p,[]\uparrow.\texttt{l},[]\uparrow.\texttt{p}} \Diamond q = \texttt{NIL} \right) .$$

Niezmiennik drzew BST można uzyskać wzmacniając niezmiennik drzew binarnych następującą formułą (zakładając, że porządek na wartościach typu `data` opisuje relacja $\leq$):

$$\forall_{q:\texttt{tree}} \left((q\uparrow.\texttt{l}\uparrow)\downarrow \Rightarrow \triangle_{p:\texttt{tree},q\uparrow.\texttt{l}\uparrow,[].\texttt{l}\uparrow,[].\texttt{p}\uparrow} \Box p.\texttt{x} \leq q\uparrow.\texttt{x} \right) \wedge$$

$$\forall_{q:\texttt{tree}} \left((q\uparrow.\texttt{p}\uparrow)\downarrow \Rightarrow \triangle_{p:\texttt{tree},q\uparrow.\texttt{p}\uparrow,[].\texttt{l}\uparrow,[].\texttt{p}\uparrow} \Box q\uparrow.\texttt{x} \leq p.\texttt{x} \right) .$$

Niezmiennik drzew AVL można uzyskać wzmacniając niezmiennik drzew BST następującą formułą:

$$\forall_{p:\mathtt{tree}} \left((p\uparrow)\downarrow \Rightarrow \begin{pmatrix} (p\uparrow.\mathtt{l} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p} = \mathtt{NIL}) \vee \\ (p\uparrow.\mathtt{l} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p}\uparrow.\mathtt{l} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p}\uparrow.\mathtt{p} = \mathtt{NIL}) \vee \\ (p\uparrow.\mathtt{l}\uparrow.\mathtt{l} = \mathtt{NIL}) \wedge p\uparrow.\mathtt{l}\uparrow.\mathtt{p} = \mathtt{NIL} \wedge p\uparrow.\mathtt{p} = \mathtt{NIL}) \vee \\ \begin{pmatrix} \triangle_{q:\mathtt{tree},p\uparrow.\mathtt{p},[]\uparrow.\mathtt{l},[]\uparrow.\mathtt{p}} \bigtriangledown_{r:\mathtt{tree},p,[]\uparrow.\mathtt{l},[]\uparrow.\mathtt{p}} (\mathcal{O}(r = p\uparrow.\mathtt{l}) \wedge \Diamond q = \mathtt{NIL}) \wedge \\ \triangle_{q:\mathtt{tree},p\uparrow.\mathtt{l},[]\uparrow.\mathtt{l},[]\uparrow.\mathtt{p}} \bigtriangledown_{r:\mathtt{tree},p,[]\uparrow.\mathtt{l},[]\uparrow.\mathtt{p}} (\mathcal{O}(r = p\uparrow.\mathtt{p}) \wedge \Diamond q = \mathtt{NIL}) \end{pmatrix} \end{pmatrix} \right) .$$

Kolejne alternatywy w powyższej formule odpowiadają następującym przypadkom:

- p jest liściem,
- p ma tylko prawego syna, który jest liściem,
- p ma tylko lewego syna, który jest liściem,
- p ma lewego i prawego syna, przy czym wysokość prawego poddrzewa nie jest większa od wysokości lewego poddrzewa o więcej niż 1, a wysokość lewego poddrzewa nie jest większa od wysokości prawego poddrzewa o więcej niż 1.

□

Jak pokazują przykłady zawarte w tym podrozdziale, *MPL* jest formalizmem pozwalającym na względnie czytelne i zwięzłe specyfikowanie wybranych przez nas wskaźnikowych struktur danych.

4.7 Porównanie

Porównamy teraz analizowane formalizmy pod kątem ich przydatności do specyfikowania wskaźnikowych struktur danych, na podstawie przedstawionych w tym rozdziale przykładów specyfikacji wybranych przez nas struktur. Ze względu na ograniczoną objętość niniejszej pracy oraz bogactwo stosowanych wskaźnikowych struktur danych, porównanie to nie wyczerpuje wszystkich możliwych zastosowań analizowanych formalizmów do specyfikowania wskaźnikowych struktur danych. Jednakże przedstawione przykłady specyfikacji oraz fakty dotyczące siły wyrazu badanych formalizmów wyraźnie je różnicują.

Przyjęliśmy dwa kryteria oceny przydatności badanych formalizmów: siłę wyrazu oraz zwięzłość i czytelność specyfikacji. Siłę wyrazu analizowanych formalizmów ilustruje poniższa tabelka, w której zaznaczono, za pomocą których formalizmów można wyspecyfikować poszczególne struktury danych.

	FOL	*HRG*	*MSOL*	*DDtL*	*SOL*	*FPL*	*MPL*
Listy jednokierunkowe	−	+	+	+	+	+	+
Listy cykliczne	−	+	+	+	+	+	+
Drzewa binarne	−	+	+	+	+	+	+
Drzewa BST	−	+	+	+	+	+	+
Drzewa *find-union*	−	+	+	+	+	+	+
Kopce binarne	−	+	+	+	+	+	+
Grafy	−	−	+	+	+	+	+
Drzewa AVL	−	−	−	−	+	+	+

Tak więc wystarczającą siłę wyrazu do opisywania wybranych przez nas wskaźnikowych struktur danych posiadają: logika drugiego rzędu, logika stałopunktowa i logika wielościeżkowa.

Kwestia czytelności i zwięzłości specyfikacji jest częściowo subiektywna. Żaden z przedstawionych przykładów specyfijacji nie jest tak prosty i czytelny, jak byśmy tego chcieli. Mimo to przykłady te pozwalają na porównanie analizowanych formalizmów pod kątem zwięzłości i czytelności zapisanych w nich specyfikacji. Ilustruje to poniższa tabelka.

MSOL	*SOL*	*FPL*	*HRG*	*DDtL*	*MPL*
$-$	$-$	$+/-$	$+$	$+$	$+$

Tak więc spośród badanych przez nas formalizmów najlepiej nadaje się do specyfikowania wskaźnikowych struktur danych logika wielościeżkowa *MPL*. Dlatego też w dalszej części pracy skoncentrujemy się na tym formalizmie.

Rozdział 5

Weryfikacja implementacji

W rozdziale tym zajmujemy się problemem weryfikacji implementacji względem projektu implementacji dla modułów operujących na wskaźnikowych strukturach danych. Polega ona na pokazaniu dla każdej operacji udostępnianej przez moduł, że jeżeli wywołamy tę operację w stanie struktury danych spełniającym niezmiennik, oraz będzie spełniony warunek początkowy, to niezmiennik struktury danych zostanie zachowany i będzie spełniony warunek końcowy operacji.

Znanych jest wiele formalizmów służących do weryfikacji programów. Do najważniejszych należą, kilkakrotnie już wcześniej wspomniane: logika Hoare'a, transformatory predykatów Dijkstry oraz logika algorytmiczna. W pracy tej jako metodę weryfikacji przyjmujemy logikę Hoare'a. Należy zaznaczyć, że wybieramy ją jako metodę przykładową. Jest to oczywiście pewne uproszczenie, gdyż za pomocą logiki Hoare'a można wykazać jedynie częściową poprawność programów.

Należy zaznaczyć, że logiki algorytmicznej można użyć zarówno do zapisu projektu implementacji jak i do weryfikacji implementacji (por. [Bał]).

Przyjmujemy, że asercje opisujące działanie operacji są wyrażone w logice wielościeżkowej (por. p. 4.6). Prezentowany formalizm można, bez zmian, zastosować dla asercji wyrażonych w dowolnej z przedstawionych wcześniej logik — korzystamy tu tylko z tego, że język asercji zawiera w sobie konstrukcje logiki pierwszego rzędu.

Rozpatrując implementacje operacji ograniczymy się do *while*-programów. Głównym mankamentem takiego ograniczenia jest pominięcie implementacji rekurencyjnych. Jest ono o tyle akceptowalne, że każdy program rekurencyjny operujący na wskaźnikowych strukturach danych można przepisać na nierekurencyjny. Jendocześnie prezentowany tu wariant logiki Hoare'a można dalej rozszerzać, np. o procedury i właśnie rekurencję.

Przypomnijmy, że na potrzeby tej pracy jako język implementacji przyjęliśmy język Modula 2. Składnia i semantyka *while*-programów zapisanych w Moduli została przedstawiona w podrozdziale 2.9.

5.1 Logika Hoare'a dla *MPL*

Przedstawimy teraz wariant logiki Hoare'a dla asercji wyrażonych w *MPL* (w skrócie *HMPL*). Przypomnijmy, że *MPL* można by tu zastąpić dowolną z omawianych wcześniej logik, zachowując prezentowane rezultaty.

Formuła Hoare'owska składa się z dwóch asercji (formuł logicznych): warunku początkowego i końcowego, oraz z programu. Warunek początkowy odnosi się do stanu struktury danych sprzed wykonania programu, a końcowy po jego wykonaniu. Zwykle jednak nie chcemy wyrazić tylko tego, że program kończąc się jest w określonym stanie, ale również to, że stan ten jest w określony sposób powiązany z początkowym stanem struktury danych. W klasycznej logice Hoare'a efekt ten uzyskuje się poprzez wprowadzenie dodatkowych zmiennych logicznych. Wartość takiej zmiennej nie jest zmieniana przez program. Przechowuje ona początkową wartość pewnej zmiennej programistycznej. Dzięki temu w warunku końcowym poprzez tę dodatkową zmienną można się odwołać do początkowej wartości zmiennej programistycznej. Jednak w przypadku wskaźnikowych struktur danych takie dodatkowe zmienne powinny przechowywać wartości wszystkich alokowanych zmiennych. Wymaga to wprowadzenia (do języka asercji) zmiennych drugiego rzędu. Przykład takiej wersji logiki Hoare'a można znaleźć w pracy [BS95].

W proponowanej wersji logiki Hoare'a postaramy się uniknąć wprowadzania asercji w logice drugiego rzędu. Zauważmy, że można obejść się bez kwantyfikowania owych dodatkowych zmiennych. Tak więc nie muszą być to koniecznie zmienne. Zamiast tego pozwalamy na rozszerzenie sygnatury i wprowadzenie dodatkowych symboli funkcyjnych $\uparrow$ z różnymi dekoracjami (por. p. 2.8). Przyjmujemy następującą konwencję:

- niedekorowane symbole funkcyjne $\uparrow$ odnoszą się do „aktualnego" stanu struktury danych,
- symbole funkcyjne postaci $\uparrow'$ mogą występować w warunku końcowym i odnoszą się one do stanu struktury danych sprzed wykonania programu,
- symbole funkcyjne $\uparrow$ z innymi dekoracjami mają charakter pomocniczy.

Na potrzeby tego podrozdziału niech $\Sigma \in \mathit{Decor}(\mathit{Int}_t)$, a X będzie zbiorem zmiennych.

Def. 33 Formułami Hoare'owskimi (nad sygnaturą Σ i zbiorem zmiennych X) dla MPL nazywamy wszystkie takie trójki $\{\varphi_1\}P\{\varphi_2\}$, że $\varphi_1 \in \mathit{MPL}_\Sigma(X)$, $\varphi_2 \in \mathit{MPL}_{\Delta\Sigma}(X)$, $P \in \mathit{Prog}_t(X)$. Zbiór wszystkich takich formuł dla zadanego Σ, Int_t oraz X oznaczamy przez $\mathit{HMPL}_{\Sigma,t}(X)$. □

Algebry częściowe należące do $\mathit{DStr}(\Sigma)$ określają zarówno stan struktury danych jak i interpretację symboli funkcyjnych nie ulegających zmianie na skutek wykonania programu. Dlatego też spełnianie formuł Hoare'owskich zdefiniujemy dla zbiorów struktur relacyjnych, których wspólnym mianownikiem są nośniki i interpretacja symboli nie zmienianych przez programy.

Def. 34 Niech $A \in \mathit{DStr}(\Delta\Sigma)$, $v : X \to |A|$ wartościowanie, $\varphi \in \mathit{MPL}_\Sigma(X)$, $\psi \in \mathit{MPL}_{\Delta\Sigma}(X)$, $P \in \mathit{Prog}_t(X)$. Przez $\mathit{Post}(\overline{A}, v, \varphi, P)$ oznaczamy nastepujący zbiór:

$$\mathit{Post}(\overline{A}, v, \varphi, P) = \{A_1 \in \overline{A} \mid A_1 \models_{\tilde{v}} {\uparrow'}\varphi, A_{1\,|\varepsilon} \in [\![P]\!]_v(A_{1\,|\Lambda})\} \ .$$

Przez $\mathit{Pre}(\overline{A}, v, \psi, P)$ oznaczamy nastepujący zbiór:

$$Pre(\overline{A}, v, \psi, P) = \left\{ A_1 \in \overline{A} \;\middle|\; \forall_{A_2 \in \overline{A}} \left(\left(\begin{array}{l} A_2 \text{ różni się od } A_1 \text{ wy-} \\ \text{łącznie interpretacją sym-} \\ \text{boli funkcyjnych postaci } \uparrow \\ \text{oraz } A_{2\,|\varepsilon} \in [\![P]\!]_v(A_{2\,|\backslash}) \end{array} \right) \Rightarrow A_2 \models_{\tilde{v}} \psi \right) \right\} .$$

□

Intuicyjnie, *Post* jest zbiorem struktur reprezentujących wyniki wykonania programu P zaczynające się w stanach spełniających φ, *Pre* jest zbiorem struktur reprezentujących początkowe stany struktur danych, dla których wykonania programu P jeśli się kończą, to prowadzą do stanów spełniających ψ.

Def. 35 Mówimy, że formuła Hoare'owska $\{\varphi\}P\{\psi\} \in HMPL_{\Sigma,t}(X)$ jest spełniona dla $\overline{A}$ ($A \in DStr(\Delta\Sigma)$) i wartościowania $v : X \to |\overline{A}|$, co oznaczamy przez $\overline{A} \models_v \{\varphi\}P\{\psi\}$, gdy dla każdego $A_1 \in Post(\overline{A}, v, \varphi, P)$, mamy $A_1 \models_{\tilde{v}} \psi$. □

Fakt 9 Niech $\overline{A} \models_v \{\varphi\}P\{\psi\}$. Wówczas:

- jeśli $A_1 \models_{\tilde{v}} \backslash\varphi$, to $A_1 \in Pre(\overline{A}, v, \psi, P)$,
- jeśli istnieje takie $\pi \in MPL_\Sigma(X)$, że $Pre(\overline{A}, v, \psi, P) = \{A_1 \in \overline{A} | A_1 \models_{\tilde{v}} \backslash\pi\}$, to dla każdego $A_1 \in \overline{A}$, $A_1 \models_{\tilde{v}} \varphi \Rightarrow \pi$.

□

Dowód Niech $A_1 \in \overline{A}$ takie, że $A_1 \models_{\tilde{v}} \backslash\varphi$. Załóżmy, że $A_2 \in \overline{A}$ dowolne takie, że różni się od A_1 tylko interpretacją symboli funkcyjnych postaci $\uparrow$, oraz $A_{2\,|\varepsilon} \in [\![P]\!]_v(A_{1\,|\backslash})$. Mamy $A_2 \models_{\tilde{v}} \backslash\varphi$ oraz $A_2 \models_{\tilde{v}} \psi$. Stąd $A_1 \in Pre(\overline{A}, v, \psi, P)$.

Niech $\pi \in MPL_\Sigma(X)$ takie, że $Pre(\overline{A}, v, \psi, P) = \{A_1 \in \overline{A} | A_1 \models_{\tilde{v}} \backslash\pi\}$. Tak więc jeśli $A_1 \models_{\tilde{v}} \backslash\varphi$, to $A_1 \models_{\tilde{v}} \backslash\pi$. Czyli $A_1 \models_{\tilde{v}} \backslash(\varphi \Rightarrow \pi)$. Stąd otrzymujemy, że dla każdego $A_1 \in \overline{A}$ mamy $A_1 \models_{\tilde{v}} \backslash\varphi$. ■

Fakt 10 Jeśli $\pi \in MPL_\Sigma(X)$ oraz $\psi, \psi_1 \in MPL_{\Delta\Sigma}(X)$ są takie, że:

$$Pre(\overline{A}, v, \psi, P) \;\; ,= \{A_1 \in \overline{A} | A_1 \models_{\tilde{v}} \backslash\pi\}$$

$$Post(\overline{A}, v, \pi, P) \;\; ,= \{A_1 \in \overline{A} | A_1 \models_{\tilde{v}} \psi_1\}$$

to $\forall_{A_1 \in \overline{A}} A_1 \models_{\tilde{v}} \psi_1 \Rightarrow \psi$. □

Dowód Załóżmy, że $A_1 \models_{\tilde{v}} \psi_1$, czyli $A_1 \in Post(\overline{A}, v, \pi, P)$. Stąd $A_1 \models_{\tilde{v}} \backslash\pi$ oraz $A_{1\,|\varepsilon} \in [\![P]\!]_v(A_{1\backslash})$. Tak więc $A_1 \in Pre(\overline{A}, v, \psi, P)$. Stąd $A_1 \models_{\tilde{v}} \psi$. Z ogólności A_1 mamy $\forall_{A_1 \in \overline{A}} A_1 \models_{\tilde{v}} \psi_1 \Rightarrow \psi$. ■

Niech $\delta \in \mathcal{D}$ będzie symbolem dekoracji, $t_1, \ldots, t_k$ będą rodzajami wskaźnikowymi. Przez $Chng(\delta, t_1, \ldots, t_k)$ oznaczamy formułę *FOL* mówiącą, że stany struktury danych opisane przez $\uparrow$ i $\uparrow^\delta$ mogą się różnić tylko wartościami zmiennych o adresach rodzai $t_1, \ldots, t_k$. Jeżeli t_i jest rodzajem wskaźnikowym do typu rekordowego, to stany struktury danych mogą sie również różnić wartościami zmiennych tworzących pola rekordów o adresach rodzaju t_i (a jeśli te pola są rekordami, to również wartościami zmiennych tworzących pola tych rekordów, itd.).

Podobnie, jeżeli $t_i = \uparrow u_i$, t jest typem rekordowym zawierającym pole(a) typu u_i, to stany struktury danych mogą się różnić wartościami zmiennych typu t, ale tylko na polach typu(ów) u_i.

Przedstawmy teraz zestaw aksjomatów i reguł wnioskowania dla *HMPL*.

Def. 36 Niech $\Lambda \subseteq MPL_{\Delta\Sigma}(X)$ będzie takim zbiorem formuł, że dla każdego morfizmu sygnatur $\sigma : \Delta\Sigma \to \Delta\Sigma$ permutującego dekoracje, jeśli $\alpha \in \Lambda$, to $\sigma(\alpha) \in \Lambda$. Niech $\{\varphi\}P\{\psi\} \in HMPL_{\Sigma,t}(X)$. Przez $\Lambda \vdash \{\varphi\}P\{\psi\}$ oznaczamy, że na podstawie założeń zawartych w Λ, w opisanym poniżej systemie dowodzenia można udowodnić formułę $\{\varphi\}P\{\psi\}$. Zamiast $\emptyset \vdash \{\varphi\}P\{\psi\}$ piszemy $\vdash \{\varphi\}P\{\psi\}$. Poniższy system dowodzenia nazywamy w skrócie *HMPL*.

- Aksjomaty:
  - aksjomat instrukcji pustej:

$$\vdash \{\varphi\}\varepsilon\{\backprime\varphi \wedge Chng(\backprime)\} \ ,$$

  - aksjomat przypisania:

$$\vdash \{\varphi\}o\,{:=}\,e \left\{ \begin{array}{l} \backprime\varphi \wedge (\backprime o)\downarrow \wedge (\backprime e)\downarrow \wedge \backprime(\&(o))\uparrow = \backprime e \wedge \\ Chng(\backprime, t) \wedge \forall_{x:t}(x \neq \backprime(\&(o)) \Rightarrow x\uparrow \equiv x\backprime\uparrow) \end{array} \right\} \ ,$$

    gdzie t jest rodzajem $\&(o)$,
  - aksjomat alokacji:

$$\vdash \{\varphi\}\texttt{NEW}(o) \left\{ \begin{array}{l} \backprime\varphi \wedge (\backprime o)\downarrow \wedge \neg(\backprime(\&(o))\uparrow \backprime\uparrow)\downarrow \wedge (\backprime(\&(o))\uparrow\uparrow)\downarrow \ \wedge \\ Chng(\backprime, t_1, t_2) \wedge \forall_{x:t_1}(x \neq \backprime(\&(o)) \Rightarrow x\uparrow \equiv x\backprime\uparrow) \wedge \\ \forall_{x:t_2}(x \neq \backprime(\&(o))\uparrow \Rightarrow x\uparrow \equiv x\backprime\uparrow) \wedge loc(\backprime(\&(o))\uparrow) \end{array} \right\} \ ,$$

    o ile $t1 \neq t_2$, gdzie t_1 jest rodzajem $\&(o)$, t_2 jest rodzajem o, t_2 jest rodzajem wskaźnikowym do t_3 — w przypadku, gdy $t_1 = t_2 = t_3$, powyższy aksjomat należy zastąpić następującym:

$$\vdash \{\varphi\}\texttt{NEW}(o) \left\{ \begin{array}{l} \backprime\varphi \wedge \backprime(o)\downarrow \wedge \neg(\backprime(\&(o))\uparrow \backprime\uparrow)\downarrow \wedge (\backprime(\&(o))\uparrow\uparrow)\downarrow \ \wedge \\ Chng(\backprime, t_1) \wedge loc(\backprime(\&(o))\uparrow) \wedge \\ \forall_{x:t_1}((x \neq \backprime(\&(o)) \wedge x \neq \backprime(\&(o))\uparrow) \Rightarrow x\uparrow \equiv x\backprime\uparrow) \end{array} \right\} \ ,$$

- Reguły:
  - osłabiania:

$$\frac{\varphi \Rightarrow \varphi_1 \in \Lambda, \Lambda_1 \vdash \{\varphi_1\}P\{\psi_1\}, (\psi_1 \wedge \backprime\varphi) \Rightarrow \psi \in \Lambda}{\Lambda \vdash \{\varphi\}P\{\psi\}} \ ,$$

    gdzie $\Lambda_1 \subseteq \Lambda$
  - dekorowania:

$$\frac{\Lambda \vdash \{\varphi \wedge Chng(\delta)\}P\{\psi\}}{\Lambda \vdash \{\varphi\}P\{\psi\}} \ ,$$

    gdzie $\delta \in \mathcal{D}$ jest dekoracją nie występującą w φ lub ψ, różną od $\backprime$.

– zmiennych lokalnych:

$$\frac{\Lambda \vdash \{\varphi_1\}\, P_1\, \{\psi_1\}}{\Lambda \vdash \{\varphi\}\texttt{VAR } x_1 : u_1; \ldots x_k : u_k;\ \texttt{BEGIN } P_1 \texttt{ END};\{\psi\}}\ ,$$

gdzie

$$\varphi_1 \mathrel{\hat{\equiv}} \begin{pmatrix} \varphi^\delta \wedge \bigwedge_{i=1,\ldots,k} \left(loc(\&x_i) \wedge \neg(\&x_i\uparrow^\delta)\downarrow \wedge \neg(\&x_i\uparrow^\tau)\downarrow \wedge(\&x_1\uparrow)\downarrow\right) \wedge \\ \bigwedge_{\substack{1\le i<j\le k \\ u_i=u_j}} (p_i \neq p_j \wedge Chng(\delta, \uparrow u_1, \ldots, \uparrow u_k)) \ \wedge \\ \bigwedge_{i=1,\ldots,k} \forall_{p:\uparrow u_i} \left(\bigwedge_{\substack{j=1,\ldots,k \\ u_i=u_j}} p \neq \&x_j \Rightarrow p\uparrow \equiv p\uparrow^\delta \right) \end{pmatrix}\ ,$$

$$\psi_1 \mathrel{\hat{\equiv}} \left(\begin{pmatrix} Chng(\tau, \uparrow u_1, \ldots, \uparrow u_k) \wedge \\ \bigwedge_{i=1,\ldots,k} \forall_{p:u_i} \left(\bigwedge_{\substack{j=1,\ldots,k \\ u_i=u_j}} p \neq \&x_j \Rightarrow p\uparrow \equiv p\uparrow^\tau \right) \end{pmatrix} \Rightarrow \psi^\tau \right)$$

i gdzie δ i τ są (różnymi) dekoracjami nie występującymi w φ lub ψ, różnymi od $\setminus$, zmienne $\&x_1, \ldots, \&x_n$ nie występują ani w φ, ani w ψ,

– złożenia sekwencyjnego:

$$\frac{\Lambda \vdash \{\varphi\}P\{\xi\}, \Lambda \vdash \{\sigma(\xi)\}Q\{\sigma(\psi)\}}{\Lambda \vdash \{\varphi\}P;Q\{\psi\}}$$

gdzie δ jest dekoracją nie występującą w φ lub ψ, różną od $\setminus$, a $\sigma : \Delta\Sigma \to \Delta\Sigma$ jest morfizmem sygnatur zmieniającym dekoracje $\setminus$ na δ,

– instrukcji warunkowej:

$$\frac{\Lambda \vdash \{\varphi \wedge OK(\alpha) \wedge \alpha\}P\{\psi\}, \Lambda \vdash \{\varphi \wedge OK(\alpha) \wedge \neg\alpha\}Q\{\psi\}}{\Lambda \vdash \{\varphi\}\texttt{IF } \alpha \texttt{ THEN } P \texttt{ ELSE } Q \texttt{ END}\{\psi\}}$$

– pętli:

$$\frac{\Lambda \vdash \{\varphi \wedge OK(\alpha) \wedge \alpha\}P\{\varphi\}}{\Lambda \vdash \{\varphi\}\texttt{WHILE } \alpha \texttt{ DO } P \texttt{ END}\{\varphi \wedge OK(\alpha) \wedge \neg\alpha\}}$$

□

Intuicyjnie Λ reprezentuje wszystkie znane twierdzenia dotyczące struktur danych nie zależące jednak od konkretnych stanów struktury danych. Przykładowo, za Λ można podstawiać zbiór wszystkich twierdzeń *MPL* prawdziwych dla zadanego $\overline{A}$ i wartościowania v:

$$Th^{MPL}_{\Delta\Sigma}(\overline{A}, \tilde{v}) = \left\{ \varphi \in MPL_{\Delta\Sigma}(X) \ \middle|\ \forall_{A_1 \in \overline{A}} A_1 \models_{\tilde{v}} \varphi \right\}\ .$$

5.2 Poprawność systemu dowodzenia *HMPL*

Pokażemy, że przedstawiony system dowodzenia *HMPL* jest poprawny.

Tw. 3 System dowodzenia *HMPL* jest poprawny, tzn. jeśli $\Lambda \vdash \{\varphi\}P\{\psi\}$, $A \in DStr(\Sigma)$, $v : X \to |A|$ takie, że dla każdego $\alpha \in \Lambda$, $A_1 \in \overline{A}$ mamy $A_1 \models_{\tilde{v}} \alpha$, to $\overline{A} \models_v \{\varphi\}P\{\psi\}$. □

Dowód Załóżmy, że $\Lambda \vdash \{\varphi\}P\{\psi\}$, $A \in DStr(\Sigma)$, $v : X \to |\overline{A}|$ takie, że dla każdego $\alpha \in \Lambda$, $A_1 \in \overline{A}$, mamy $A_1 \models_{\tilde{v}} \alpha$. Mamy pokazać, że $\overline{A} \models_v \{\varphi\}P\{\psi\}$. Załóżmy dalej, że $A \in Post(\overline{A}, v, \varphi, P)$. Należy więc pokazać, że $A \models_{\tilde{v}} \psi$. Dowód twierdzenia przebiega indukcyjnie ze względu na strukturę dowodu $\Lambda \vdash \{\varphi\}P\{\psi\}$.

- Jeżeli $\vdash \{\varphi\}\varepsilon\{\psi\}$ jest aksjomatem instrukcji pustej, to w oczywisty sposób mamy $A \models_{\tilde{v}} \psi$.

- Jeżeli $\vdash \{\varphi\}d\texttt{:=}\ t\{\psi\}$ jest aksjomatem przypisania, to mamy $A_{|\setminus} \models_{\tilde{v}} d\downarrow \wedge\, t\downarrow$, $A_{|\varepsilon} = Store(A_{|\setminus}, v^{A_{|\setminus}}(d), v^{A_{|\setminus}}(t))$. Stąd $A \models_{\tilde{v}} \setminus(d)\downarrow \wedge\, \setminus(t)\downarrow$. Niech s będzie rodzajem $\&(d)$. Z definicji *Store* otrzymujemy

$$A \models_{\tilde{v}} Chng(\setminus, s) \wedge \forall_{x:s}(x \neq \setminus(\&(d)) \Rightarrow x\uparrow \equiv x\setminus\uparrow) \wedge \setminus(\&(d))\uparrow = \setminus(t)\ .$$

  Stąd otrzymujemy $A \models_{\tilde{v}} \psi$.

- Jeżeli $\vdash \{\varphi\}\texttt{NEW}(d)\{\psi\}$ jest aksjomatem alokacji, s_1 jest rodzajem $\&(d)$, s_2 jest rodzajem d, s_2 jest rodzajem wskaźnikowym do s_3, $s_1 \neq s_2$, to $A_{|\setminus} \models_{\tilde{v}} d\downarrow$, istnieje taki $x \in |A|_{s_2}$ oraz $y \in |A|_{s_3}$, że $x\uparrow^{A_{|\setminus}}$ jest nieokreślone oraz $A_{|\varepsilon} = Store(Store(A_{|\setminus}, v^{A_{|\setminus}}(\&(d)), x), x, y)$. Mamy więc $A \models_{\tilde{v}} (\setminus d)\downarrow$. Z definicji *Store* mamy $v^A((\setminus\&(d))\uparrow) = x$. Stąd $A \models_{\tilde{v}} \neg((\setminus\&(d))\uparrow \setminus\uparrow)\downarrow$ oraz $((\setminus\&(d))\uparrow\uparrow)\downarrow$. Z definicji *Store* mamy również:

$$A \models_{\tilde{v}} Chng(\setminus, t_1, t_2) \wedge \forall_{x:s_1}(x \neq \setminus(\&(d)) \Rightarrow x\uparrow \equiv x\setminus\uparrow) \wedge \forall_{x:s_2}(x \neq \setminus(\&(d))\uparrow \Rightarrow x\uparrow \equiv x\setminus\uparrow)\ .$$

  Stąd mamy $A \models_{\tilde{v}} \psi$. Przypadek gdy $s_1 = s_2 = s_3$ rozpatrujemy podobnie.

- Załóżmy, że ostatnią regułą dowodu jest reguła osłabiania, $\varphi \Rightarrow \varphi_1, (\psi_1 \wedge \setminus\varphi) \Rightarrow \psi \in \Lambda_1$, $\overline{A} \models_v \{\varphi_1\}P\{\psi_1\}$. Tak więc $A \models_{\tilde{v}} \setminus\varphi_1$. Stąd $A \models_{\tilde{v}} \psi_1$, a stąd $A \models_{\tilde{v}} \psi$.

- Załóżmy, że ostatnią regułą dowodu jest reguła dekorowania, $\delta \in \mathcal{D}$ jest dekoracją nie występującą w φ lub ψ, różną od $\setminus$, oraz $\overline{A} \models_v \{\varphi \wedge Chng(\delta)\}P\{\psi\}$. Oznaczmy przez $\sigma : \Delta\Sigma \to \Delta\Sigma$ morfizm sygnatur, który zamienia dekoracje δ na $\setminus$. Mamy $A \models_{\tilde{v}} \setminus(\varphi \wedge Chng(\varepsilon))$, czyli $A_{|\sigma} \models_{\tilde{v}} \setminus(\varphi \wedge Chng(\delta))$. Ponadto $A_{|\sigma} \in Post(\overline{A}, v, \setminus(\varphi \wedge Chng(\delta))$. Stąd $A_{|\sigma} \models_{\tilde{v}} \psi$ czyli $A \models_{\tilde{v}} \psi$.

- Załóżmy, że ostatnią regułą dowodu jest reguła zmiennych lokalnych, $\delta, \tau \in \mathcal{D}$ są różnymi dekoracjami, różnymi od $\setminus$ i nie występującymi ani w φ ani w ψ, P jest postaci $\texttt{VAR}\ x_1 : u_1; \ldots x_k : u_k;\ \texttt{BEGIN}\ P_1\ \texttt{END};$ oraz $\overline{A} \models_v \{\varphi_1\}P\{\psi_1\}$, gdzie φ_1 i ψ_1 są jak w regule zmiennych lokalnych. Ponieważ $A_{|\varepsilon} \in [\![P]\!]_v(A_{|\setminus})$, więc istnieją $p_1 \in |\overline{A}|_{\uparrow u_1}, \ldots, p_k \in |\overline{A}|_{\uparrow u_k}$, $y_1 \in |\overline{A}|_{u_1}, \ldots, y_k \in |\overline{A}|_{u_k}$ i $A_1 \in \overline{A}$ takie, że dla $i = 1, \ldots, k$ mamy $loc^A(p_i)$ i $p_i\uparrow^{A_{|\setminus}}$ jest nieokreślone, dla $1 \leq i < j \leq k, u_i = u_j$ mamy $p_i \neq p_j$, A_1 różni się od A wyłącznie interpretacją symboli funkcyjnych postaci: $\uparrow$, $\setminus\uparrow$, $\uparrow^\delta$ i $\uparrow^\tau$, $A_{1\,|\delta} = A_{|\setminus}$, $A_{1\,|\tau} = A_{|\varepsilon}$, oraz:

$$\begin{gathered} A_{1\,|\setminus} = Store(\ldots Store(A_{|\setminus}, p_1, y_1) \ldots, p_k, y_k)\ , \\ A_{|\varepsilon} = Dealloc(\ldots Dealloc(A_{1\,|\varepsilon}, p_1), \ldots, p_k)\ , \\ A_{1\,|\varepsilon} \in [\![P_1]\!]_{v_1}(A_{1\,|\setminus})\ , \end{gathered}$$

  gdzie $v_1 = v[\&x_1 \to p_1] \ldots [\&x_k \to p_k]$. Z definicji *Store* i *Dealloc* oraz powyższych założeń mamy $A_1 \models_{\tilde{v_1}} \setminus\varphi_1$. Stąd $A_1 \in Post(\overline{A}, v_1, \varphi_1, P_1)$, czyli $A_1 \models \psi_1$. Ponadto z

definicji *Dealloc* mamy $Chng(\tau, \uparrow u_1, \ldots, \uparrow u_k)$, oraz dla każdego $i = 1, \ldots, k$ mamy:

$$\forall_{p:u_i}(\bigwedge_{\substack{j=1,\ldots,k \\ u_i=u_j}} (p \neq \&x_j) \Rightarrow p\uparrow \equiv p\uparrow^{\tau}) \ .$$

Stąd $A_1 \models_{\tilde{v_1}} \psi^{\tau}$, czyli $A \models_{\tilde{v_1}} \psi$. Ponieważ zmienne $\&x_1, \ldots, \&x_k$ nie występują w ψ, mamy $A \models_{\tilde{v}} \psi$.

- Załóżmy, że ostatnią regułą dowodu jest reguła złożenia sekwencyjnego, P jest postaci $Q;R$, δ jest dekoracją różną od $\setminus$, nie występującą ani w φ, ani w ψ, $\sigma : \Delta\Sigma \to \Delta\Sigma$ jest morfizmem sygnatur zmieniającym dekoracje $\setminus$ na δ, $\overline{A} \models_v \{\varphi\}Q\{\xi\}$, $\overline{A} \models_v \{\sigma(\xi)\}R\{\sigma(\psi)\}$. Ponieważ $A_{|\varepsilon} \in [\![Q;R]\!]_v(A_{|\setminus})$, więc istnieje takie $A_1 \in [\![Q]\!]_v(A_{|\setminus})$, że $A_{|\varepsilon} \in [\![R]\!]_v(A_1)$. Niech $B \in \overline{A}$ różni się od A jedynie interpretacją symboli funkcyjnych postaci $\uparrow^{\delta}$ oraz $B_{|\delta} = A_1$. Mamy wówczas: $B \models_{\tilde{v}} \setminus\varphi$, $B_{|\delta} \in [\![Q]\!]_v(B_{|\setminus})$, $B_{|\varepsilon} \in [\![R]\!]_v(B_{|\delta})$. Niech $\sigma_1, \sigma_2 : \Delta\Sigma \to \Delta\Sigma$ morfizmy sygnatur: σ_1 zmienia symbole niedekorowane na dekorowane δ, zaś σ_2 zamienia dekoracje $\setminus$ na δ i odwrotnie. Mamy $B_{|\sigma_1} \models_{\tilde{v}} \setminus\varphi$, $(B_{|\sigma_1})_{|\varepsilon} \in [\![Q]\!]_v((A_{|\sigma_1})_{|\setminus})$. Stąd $B_{|\sigma_1} \models_{\tilde{v}} \xi$, czyli $B \models_{\tilde{v}} \xi^{\delta}$. Zauważmy, że $\xi^{\delta} = \sigma_2(\sigma_2(\xi^{\delta})) = \sigma_2(\setminus\sigma(\xi))$, czyli $B_{|\sigma_2} \models_{\tilde{v}} \setminus\sigma(\xi)$. Ponadto $(B_{|\sigma_2})_{|\varepsilon} = B_{|\varepsilon}$, oraz $(B_{|\sigma_2})_{|\setminus} = B_{|\delta}$. Tak więc $(B_{|\sigma_2})_{|\varepsilon} \in Post(\overline{A}, v, \sigma(\xi), R)$. Stąd $B_{|\sigma_2} \models_{\tilde{v}} \sigma(\psi)$, czyli $B_{|\sigma_2} \models_{\tilde{v}} \sigma_2(\psi)$, a stąd $A \models_{\tilde{v}} \psi$.

- Załóżmy, że ostatnią regułą dowodu jest reguła instrukcji warunkowej, P jest postaci `IF` α `THEN` Q `ELSE` R `END`, $\overline{A} \models_v \{\varphi \wedge OK(\alpha) \wedge \alpha\}Q\{\psi\}$, $\overline{A} \models_v \{\varphi \wedge OK(\alpha) \wedge \neg\alpha\}R\{\psi\}$. Ponieważ $A_{|\varepsilon} \in [\![P]\!]_v(A_{|\setminus})$, więc $A_{|\setminus} \models_v OK(\alpha)$. Rozważmy dwa przypadki:

  1. $A_{|\setminus} \models_v \alpha$, wówczas $A_{|\varepsilon} \in [\![Q]\!]_v(A_{|\setminus})$. Ponadto $A \models_{\tilde{v}} \setminus(\varphi \wedge OK(\alpha) \wedge \alpha)$. Stąd $A \models_{\tilde{v}} \psi$.
  2. $A_{|\setminus} \models_v \neg\alpha$, wówczas $A_{|\varepsilon} \in [\![R]\!]_v(A_{|\setminus})$. Ponadto $A \models_{\tilde{v}} \setminus(\varphi \wedge OK(\alpha) \wedge \neg\alpha)$. Stąd $A \models_{\tilde{v}} \psi$.

  Tak więc $A \models_{\tilde{v}} \psi$.

- Załóżmy, że ostatnią regułą dowodu jest instrukcja pętli, P jest postaci `WHILE` α `DO` Q `END`, $\overline{A} \models_v \{\varphi \wedge OK(\alpha) \wedge \alpha\}Q\{\varphi\}$. Ponieważ $A_{|\varepsilon} \in [\![P]\!]_v(A_{|\setminus})$, więc istnieją takie $A_0, \ldots, A_n \in DStr(Sig(Int_t))$, że $A_{|\setminus} = A_0$, $A_{|\varepsilon} = A_n$, dla $i = 0, \ldots, n-1$ mamy $A_i \models_v OK(\alpha) \wedge \alpha$, $A_{i+1} \in [\![Q]\!]_v(A_i)$, oraz $A_n \models_v OK(\alpha) \wedge \neg\alpha$. Niech $B_1, \ldots, B_n \in \overline{A}$ takie, że B_i różni się od A tylko interpretacją symboli funkcyjnych postaci $\uparrow$ i $\setminus\uparrow$, przy czym $B_{i|\setminus} = A_{i-1}$, $B_{i|\varepsilon} = A_i$. Pokażemy przez indukcję, że $B_i \models_{\tilde{v}} \varphi$.

  Załóżmy, że dla wszystkich $1 \leq j < i$ mamy $B_j \models_{\tilde{v}} \varphi$. Mamy więc $B_i \models_{\tilde{v}} \setminus\varphi$. Stąd $B_i \models_{\tilde{v}} \setminus(\varphi \wedge OK(\alpha) \wedge \alpha)$. Ponadto $B_{i|\varepsilon} \in [\![Q]\!]_v(B_{i|\setminus})$, czyli $B_i \models_{\tilde{v}} \varphi$.

  Tak więc $A \models_{\tilde{v}} \varphi$. Mamy więc $A \models_{\tilde{v}} \varphi \wedge OK(\alpha) \wedge \neg\alpha$. Stąd $A \models_{\tilde{v}} \psi$.

■

5.3 Pełność *HMPL* w sensie Cook'a

Pokażemy teraz, że system wnioskowania *HMPL* jest pełny w sensie Cook'a.

Def. 37 Niech $A \in DStr_{\Delta\Sigma}(X)$, $v : X \to |A|$ wartościowanie. Mówimy, że zbiór formuł $MPL_\Sigma(X)$ jest ekspresywny ze względu na $\overline{A}$, v, $Prog_t(X)$, gdy dla każdego $\psi \in MPL_{\Delta\Sigma}(X)$, $P \in Prog_t(X)$ istnieje takie $\pi \in MPL_\Sigma(X)$, że $Pre(\overline{A}, v, \psi, P) = \{A_1 \in \overline{A} \mid A_1 \models_{\tilde{v}} \backslash\pi\}$. □

Tw. 4 System dowodzenia *HMPL* jest pełny w sensie Cook'a, tzn. dla każdego $A \in DStr(\Delta\Sigma)$, $v : X \to |A|$, jeśli $MPL_\Sigma(X)$ jest ekspresywny ze względu na $\overline{A}$, v i $Prog_t(X)$, oraz $\overline{A} \models_v \{\varphi\}P\{\psi\}$, to dla $A_1 \in DStr(\Delta Sig(Int_t)^{\mathcal{D}})$, ι naturalnego włożenia $\Delta\Sigma$ w $\Delta Sig(Int_t)^{\mathcal{D}}$ takiego, że $A_{1\,|\iota} = A$ mamy $Th_{\Delta Sig(Int_t)^{\mathcal{D}}}(\overline{A_1}, \tilde{v}) \vdash \{\varphi\}P\{\psi\}$. □

Dowód Niech $A \in DStr_{\Delta\Sigma}(X)$, $v : X \to |A|$, $\overline{A} \models_v \{\varphi\}P\{\psi\}$, $MPL_\Sigma(X)$ ekspresywne ze względu na $\overline{A}$, v i $Prog_t(X)$.

Niech $A_1 \in DStr(\Delta Sig(Int_t)^{\mathcal{D}})$ takie, że dla ι-naturalnego włożenia $\Delta\Sigma$ w $\Delta Sig(Int_t)^{\mathcal{D}}$ mamy $A_{1\,|\iota} = A$. Zauważmy, że $v : X \to |A_1|$, $\overline{A_1} \models_v \{\varphi\}P\{\psi\}$, $MPL_{Sig(Int_t)^{\mathcal{D}}}(X)$ ekspresywne ze względu na $\overline{A_1}$, v i $Prog_t(X)$. Możemy więc bez zmniejszenia ogólności założyć, że $\Sigma = Sig(Int_t)^{\mathcal{D}}$.

Oznaczmy $\Lambda = Th_{\Delta Sig(Int_t)^{\mathcal{D}}}(\overline{A}, \tilde{v})$. Niech $\pi \in MPL_\Sigma(X)$ takie, że $Pre(\overline{A}, v, \psi, P) = \{A_1 \in \overline{A} \mid A_1 \models_{\tilde{v}} \backslash\pi\}$. Wówczas z faktu 9 mamy $(\varphi \Rightarrow \pi) \in Th_{\Delta\Sigma}(\overline{A}, \tilde{v})$. Wystartczy więc pokazać, że $\Lambda \vdash \{\pi\}P\{\psi\}$, a z reguły osłabiania uzyskujemy $\Lambda \vdash \{\varphi\}P\{\psi\}$.

Dowód przebiega indukcyjnie ze względu na strukturę programu P.

- Załóżmy, że P jest postaci ε, $d{:=}t$ lub $\texttt{NEW}(d)$. Niech $\vdash \{\pi\}P\{\psi_1\}$ bedzie odpowiednim aksjomatem. Zauważmy, że $\{A_1 \in \overline{A} | A_1 \models_{\tilde{v}} \psi_1\} = Post(\overline{A}, v, \pi, P)$. Z faktu 10 mamy $(\psi_1 \Rightarrow \psi) \in \Lambda$. Korzystając z reguły osłabiania mamy $\Lambda \vdash \{\pi\}P\{\psi\}$.

- Załóżmy, że P jest postaci $Q; R$. Niech $\delta \in \mathcal{D}$ będzie dekoracją nie występującą ani w π ani w ψ, różną od $\backslash$, $\sigma : \Delta\Sigma \to \Delta\Sigma$ będzie morfizmem sygnatur zmieniającym dekoracje $\backslash$ na δ. Niech $\pi_1 \in MPL_\Sigma(X)$ takie, że $Pre(\overline{A}, v, \sigma(\psi), R) = \{A_1 \in \overline{A} \mid A_1 \models_{\tilde{v}} \backslash\pi_1\}$. Niech $\pi_2 \in MPL_{\Delta\Sigma}(X)$ jest wynikiem zamiany w π_1 dekoracji δ na $\backslash$. Wówczas π_2 nie zawiera dekoracji δ oraz $Pre(\overline{A}, v, \sigma(\psi), R) = \{A_1 \in \overline{A} \mid A_1 \models_{\tilde{v}} \backslash\sigma(\pi_2)\}$. Mamy $\overline{A} \models_v \{\sigma(\pi_2)\}R\{\sigma(\psi)\}$. Z założenia indukcyjnego $\Lambda \vdash \{\sigma(\pi_2)\}R\{\sigma(\psi)\}$. Pokażemy, że $\overline{A} \models_v \{\pi\}Q\{\pi_2\}$. Niech $A_1 \in \overline{A}$ takie, że $A_1 \models_{\tilde{v}} \backslash\pi$, $A'_{1\,|\varepsilon} \in [\![Q]\!]_v(A_{1\,|\backslash})$. Należy pokazać, że $A_1 \models_{\tilde{v}} \pi_2$. Rozważmy dwa przypadki:

  1. $[\![R]\!]_v(A_{1\,|\varepsilon}) = \emptyset$. Niech A_2 różni się od A_1 jedynie interpretacją symboli funkcyjnych postaci $\uparrow^\delta$ i $\backslash\uparrow$, przy czym $A_{2\,|\backslash} = A_{1\,|\varepsilon}$, $A_{2\,|\delta} = A_{1\,|\backslash}$. Wówczas $[\![R]\!](A_{2\,|\backslash}) = \emptyset$. Tak więc $A_2 \in Pre(\overline{A}, v, \sigma(\psi), R)$, czyli $A_2 \models_{\tilde{v}} \backslash\sigma(\pi_2)$. Stąd $A_1 \models_{\tilde{v}} \pi_2$.
  2. $[\![R]\!]_v(A_{1\,|\varepsilon}) \neq \emptyset$. Niech A_2 różni się od A_1 jedynie interpretacją symboli funkcyjnych postaci $\uparrow^\delta$, $\backslash\uparrow$ i $\uparrow$, przy czym $A_{2\,|\backslash} = A_{1\,|\varepsilon}$, $A_{2\,|\delta} = A_{1\,|\backslash}$, $A_{2\,|\varepsilon} \in [\![R]\!]_v(A_{2\,|\backslash})$. Mamy więc $A_{2\,|\varepsilon} \in [\![P]\!]_v(A_{2\,|\delta})$, $A_2 \models_{\tilde{v}} \pi^\delta$, czyli $(A_{2\,|\sigma})_{|\varepsilon} \in [\![P]\!]_v(A_{2\,|\sigma\,|\backslash})$, $A_{2\,|\sigma} \models_{\tilde{v}} \backslash\pi$. Stąd $A_{2\,|\sigma} \models_{\tilde{v}} \psi$, $A_2 \models_{\tilde{v}} \sigma(\psi)$. Dalej, z ogólności $A_{2\,|\varepsilon}$, mamy $A_2 \in Pre(\overline{A}, v, \sigma(\psi), R)$, czyli $A_2 \models_{\tilde{v}} \backslash\sigma(\pi_2)$. Stąd $A_1 \models_{\tilde{v}} \pi_2$.

  Tak więc $\overline{A} \models_v \{\pi\}Q\{\pi_2\}$. Z założenia indukcyjnego mamy $\Lambda \vdash \{\pi\}Q\{\pi_2\}$. Stosując regułę złożenia uzyskujemy $\Lambda \vdash \{\pi\}Q; R\{\psi\}$.

- Załóżmy, że P jest postaci `VAR` $x_1 : u_1; \ldots x_k : u_k$; `BEGIN` P_1 `END;`. Niech $\delta, \tau \in \mathcal{D}$ będą różnymi dekoracjami, różnymi od $\setminus$ nie występującymi ani w π ani w ψ, φ_1 i ψ_1 będą jak w regule zmiennych lokalnych (dla π i ψ). Pokażemy, że $\overline{A} \models_v \{\varphi_1\}P_1\{\psi_1\}$. Niech $A_1 \in \overline{A}$ dowolne takie, że $A_1 \in Post(\overline{A}, v, \varphi_1, P_1)$. Należy pokazać, że $A_1 \models_{\tilde{v}} \psi_1$. Rozważmy dwa przypadki:

  1. $A_1 \not\models_v Chng(\tau, \uparrow u_1, \ldots, \uparrow u_k) \wedge \bigwedge_{i=1,\ldots,k} \forall_{p:u_i} \left(\bigwedge_{\substack{j=1,\ldots,k \\ u_i=u_j}} p \neq \&x_j \Rightarrow p\uparrow \equiv p\uparrow^\tau \right)$
     Wówczas trywialnie $A_1 \models_{\tilde{v}} \psi_1$.

  2. $A_1 \models_v Chng(\tau, \uparrow u_1, \ldots, \uparrow u_k) \wedge \bigwedge_{i=1,\ldots,k} \forall_{p:u_i} \left(\bigwedge_{\substack{j=1,\ldots,k \\ u_i=u_j}} p \neq \&x_j \Rightarrow p\uparrow \equiv p\uparrow^\tau \right)$
     Wówczas wystarczy pokazać, że $A_1 \models_{\tilde{v}} \psi^\tau$. Zauważmy, że $A_{1\,|\delta} \in [\![P]\!]_v(A_{1\,|\tau})$. Niech $A_2 \in \overline{A}$ różni się od A_1 tylko interpretacją symboli funkcyjnych postaci $\uparrow$ i $\setminus\uparrow$, przy czym $A_{2\,|\setminus} = A_{1\,|\delta}$, $A_{2\,|\varepsilon} = A_{1\,|\tau}$. Mamy $A_2 \models_{\tilde{v}} \setminus\pi$ oraz $A_{2\,|\varepsilon} \in [\![P]\!]_v(A_{2\,|\setminus})$. Stąd $A_2 \models_{\tilde{v}} \psi$, czyli $A_1 \models_{\tilde{v}} \psi^\tau$.

  Tak więc $A_1 \models_{\tilde{v}} \psi_1$. Z ogólności A_1 mamy $\overline{A} \models_v \{\varphi_1\}P_1\{\psi_1\}$. Z założenia indukcyjnego mamy $\Lambda \vdash \{\varphi_1\}P_1\{\psi_1\}$. Stosując regułę zmiennych lokalnych otrzymujemy $\Lambda \vdash \{\pi\}P\{\psi\}$.

- Załóżmy, że P jest postaci `IF` α `THEN` Q `ELSE` R `END`. Pokażemy, że $\overline{A} \models_v \{\pi \wedge OK(\alpha) \wedge \alpha\}Q\{\phi\}$ oraz $\overline{A} \models_v \{\pi \wedge OK(\alpha) \wedge \neg\alpha\}R\{\phi\}$.

  Niech $A_1 \in \overline{A}$, $A_1 \models_{\tilde{v}} \setminus(\pi \wedge OK(\alpha) \wedge \alpha)$. Jeśli tylko $A_{1\,|\varepsilon} \in [\![Q]\!]_v(A_{1\,|\setminus})$, to $A_{1\,|\varepsilon} \in [\![P]\!]_v(A_{1\,|\setminus})$, a stąd $A_1 \models_{\tilde{v}} \psi$. Tak więc $\overline{A} \models_v \{\pi \wedge OK(\alpha) \wedge \alpha\}Q\{\phi\}$.

  Niech $A_2 \in \overline{A}$, $A_2 \models_{\tilde{v}} \setminus(\pi \wedge OK(\alpha) \wedge \neg\alpha)$. Jeśli tylko $A_{2\,\varepsilon} \in [\![R]\!]_v(A_{2\,|\setminus})$, to $A_{2\,\varepsilon} \in [\![P]\!]_v(A_{2\,|\setminus})$, a stąd $A_2 \models_{\tilde{v}} \psi$. Tak więc $\overline{A} \models_v \{\pi \wedge OK(\alpha) \wedge \neg\alpha\}R\{\phi\}$.

  Z założenia indukcyjnego mamy $\Lambda \vdash \{\pi \wedge OK(\alpha) \wedge \alpha\}Q\{\psi\}$, $\Lambda \vdash \{\pi \wedge OK(\alpha) \wedge \neg\alpha\}R\{\pi\}$. Stosując regułę instrukcji warunkowej otrzymujemy: $\Lambda \vdash \{\pi\}P\{\psi\}$.

- Załóżmy, że P jest postaci `WHILE` α `DO` Q `END`. Niech $\delta \in \mathcal{D}$ będzie dekoracją nie występującą w φ, π lub ψ. Mamy $\overline{A} \models_v \{\pi \wedge Chng(\delta)\}$`WHILE` α `DO` Q `END`$\{\psi\}$. Niech $\sigma : \Delta\Sigma \to \Delta\Sigma$ będzie morfizmem sygnatur zmieniającym dekoracje $\setminus$ na δ. Oznaczmy przez $\eta \in MPL_\Sigma(X)$ taką formułę, że $\{A_1 \in \overline{A} \mid A_1 \models_{\tilde{v}} \eta\} = Pre(\overline{A}, v, \sigma(\psi), P)$. Pokażemy, że

$$((\pi \wedge Chng(\delta)) \Rightarrow \eta) \in \Lambda$$
$$((\eta \wedge OK(\alpha) \wedge \neg\alpha) \Rightarrow \sigma(\psi)) \in \Lambda$$
$$\Lambda \vdash \{\eta \wedge OK(\alpha) \wedge \alpha\}Q\{\eta\}$$

  Zauważmy, że $\overline{A} \models_v \{\pi \wedge Chng(\delta)\}P\{\sigma(\psi)\}$. Z faktu 9 mamy $\forall_{A_1 \in \overline{A}}(A_1 \models_{\tilde{v}} (\pi \wedge Chng(\delta)) \Rightarrow \eta)$. Stąd $((\pi \wedge Chng(\delta)) \Rightarrow \eta) \in \Lambda$.

  Zauważmy, że $\overline{A} \models_v \{\eta \wedge OK(\alpha) \wedge \neg\alpha\}P\{\sigma(\psi)\}$. Niech $A_1 \in \overline{A}$ dowolne takie, że $A_1 \models_{\tilde{v}} \eta \wedge OK(\alpha) \wedge \neg\alpha$ oraz $A_2 \in \overline{A}$ takie, że różni się od A_1 tylko interpretacją symboli funkcyjnych postaci $\setminus\uparrow$, przy czym $A_{2\,|\setminus} = A_{1\,|\varepsilon}$. Mamy $A_2 \models_{\tilde{v}} \setminus(\eta \wedge OK(\alpha) \wedge \neg\alpha)$, $A_2 \in [\![P]\!]_v(A_{2\,|\setminus})$. Stąd $A_2 \models_{\tilde{v}} \sigma(\psi)$, czyli również $A_1 \models_{\tilde{v}} \sigma(\psi)$. Z ogólności A_1 mamy $((\eta \wedge OK(\alpha) \wedge \neg\alpha) \Rightarrow \sigma(\psi)) \in \Lambda$.

Zauważmy, że $\overline{A} \models_v \{\eta\}P\{\sigma(\psi)\}$. Ponadto mamy:

$$[\![P]\!]_v = [\![\texttt{IF } \alpha \texttt{ THEN } Q \texttt{ ELSE END; } P]\!]_v \ ,$$
$$Pre(\overline{A}, v, \eta, \texttt{IF } \alpha \texttt{ THEN } Q \texttt{ ELSE END}) = \{A_2 \in \overline{A} | A_2 \models_{\tilde{v}} \setminus\eta\} \ ,$$
$$\overline{A} \models_v \{\eta\}\texttt{IF } \alpha \texttt{ THEN } Q \texttt{ ELSE END}\{\eta\} \ .$$

Niech $A_1 \in \overline{A}$, $A_1 \models_{\tilde{v}} \setminus(\eta \wedge OK(\alpha) \wedge \alpha)$, $A_{1\,|\varepsilon} \in [\![Q]\!]_v(A_{1\,|\setminus})$. Wówczas również

$$A_{1\,|\varepsilon} \in [\![\texttt{IF } \alpha \texttt{ THEN } Q \texttt{ ELSE END}]\!]_v(A_{1\,|\setminus}) \ ,$$

czyli $A_1 \models_{\tilde{v}} \eta$. Z ogólności A_1 mamy $\overline{A} \models_v \{\eta \wedge OK(\alpha) \wedge \alpha\}Q\{\eta\}$. Z założenia indukcyjnego mamy $\Lambda \vdash \{\eta \wedge OK(\alpha) \wedge \alpha\}Q\{\eta\}$.

Tak więc, stosując regułę pętli i osłabiania, mamy $\Lambda \vdash \{\pi \wedge Chng(\delta)\}P\{\sigma(\psi)\}$. Zauważmy, że $(\sigma(\psi) \wedge \setminus Chng(\delta) \Rightarrow \psi) \in \Lambda$. Stosując regułę osłabiania i dekorowania mamy $\Lambda \vdash \{\pi\}P\{\psi\}$.

■

5.4 Przykład użycia *HMPL*

Na poniższym przykładzie przedstawimy zastosowanie *HMPL* do weryfikacji programu operującego na wskaźnikowych strukturach danych.

Przykład 25 Deklaracje typów występujące w projekcie implementacji Int_{bst} modułu implementującego drzewa BST mogą mieć następującą postać (por. przykłady 10, 15, 22 i 24):

```
TYPE
  tree = POINTER TO node;
  node = RECORD
    d: data;
    l, r: tree
  END;
  ppnode = POINTER TO tree;
  ptree = POINTER TO tree;
```

W prawie każdej operacji na drzewach BST występuje wyszukiwanie w drzewie wierzchołka zawierającego zadaną wartość. Oznaczmy przez P program realizujący takie właśnie wyszukiwanie. Niech działanie P zależy od dwóch zmiennych programistycznych: `p: tree` oraz `x: data`, przy czym P modyfikuje wartość zmiennej `p`. Zmienna `x` zawiera szukaną wartość. Zmienna `p` przed wykonaniem programu wskazuje na korzeń drzewa, a po jego zakończeniu wskazuje na szukany jego wierzchołek lub, gdy takiego wierzchołka nie ma, jest równa `NIL`.

Przez Σ oznaczmy sygnaturę postaci $\Sigma = \Delta Sig(Int_{\text{bst}})^{\mathcal{D}}$. W dalszej części przykładu posługujemy się formułami nad sygnaturą Σ i zbiorem zmiennych $X = \langle \&\texttt{p} : \texttt{ppnode}, \&\texttt{x} :\uparrow \texttt{data}\rangle$.

Warunek wstępny programu wyraża następujące własności:

- jest spełniony niezmiennik drzew BST (por. przykład 24),
- `p` jest zmienną pomocniczą nie będącą polem żadnego rekordu,
- jeżeli wartość `p` jest różna od `NIL`, to `p` wskazuje na korzeń pewnego drzewa,

- zmiennym &p i &x przyporządkowane są nieskończone ciągi stałe — inaczej mówiąc, wartości tych zmiennych nie zmieniają się pod operatorami modalnymi i nie ograniczają one długości rozpatrywanych ścieżek.

Warunek końcowy tworzą następujące warunki:

- niezmiennik drzew BST jest zachowany,
- jedyną zmienną programistyczną, której wartość ulega zmianie, jest p,
- z poprzedniej wartości p można osiągnąć (poprzez wskaźniki tworzące krawędzie drzewa) aktualną wartość p,
- jeśli w drzewie, na którego korzeń wskazywała poprzednia wartość p, jest wierzchołek zawierający wartość x, to aktualna wartość p wskazuje na ten wierzchołek,
- p jest równe NIL lub wskazuje na wierzchołek zawierający wartość x.

Przez φ_{bst} oznaczmy niezmiennik drzew BST, a przez φ_{var} formułę postaci:

$$\begin{aligned}\varphi_{\text{var}} \mathrel{\hat{\equiv}} & \forall_{t:\texttt{tree}}\,(t|\texttt{l} \neq \&\texttt{p} \wedge t|\texttt{r} \neq \&\texttt{p}) \wedge (\texttt{p})\downarrow \wedge \exists_{q:\texttt{ppnode}}\,(\Box q = \&\texttt{p}) \wedge \\ & (x)\downarrow \wedge\, \exists_{y:\uparrow\texttt{data}}\,(\Box y = \&x) \wedge \Box\mathcal{O}true\end{aligned}$$

mówiącą, że p nie jest częścią żadnego rekordu, że jest alokowane i wskazuje na alokowany rekord, lub jest równe NIL, a także, że zmiennym &p i &x są przyporządkowane nieskończone ciągi stałe. Warunek wstępny φ_{pre} programu możemy zapisać formalnie w następujący sposób:

$$\varphi_{\text{pre}} \mathrel{\hat{\equiv}} \varphi_{\text{bst}} \wedge \varphi_{\text{var}} \wedge (\texttt{p} \neq \texttt{NIL} \Rightarrow (\texttt{p}\uparrow)\downarrow \wedge \forall_{t:\texttt{tree}}\,(t.\texttt{l} \neq \texttt{p} \wedge t.\texttt{r} \neq \texttt{p}))\quad .$$

Przez φ_{nc} oznaczmy formułę mówiącą, że zmianie uległa tylko wartość zmiennej p:

$$\varphi_{\text{nc}} \mathrel{\hat{\equiv}} \mathit{Chng}(\backslash, \texttt{ppnode}) \wedge \forall_{t:\texttt{ppnode}}\left(t \neq \&\texttt{p} \Rightarrow t^{\backslash}\uparrow \equiv t\uparrow\right)\quad .$$

Wówczas, warunek końcowy φ_{post} programu P możemy zapisać formalnie w następujący sposób:

$$\begin{aligned}\varphi_{\text{post}} \mathrel{\hat{\equiv}} & \varphi_{\text{bst}} \wedge \varphi_{\text{nc}} \wedge \bigtriangledown_{q,\backslash\texttt{p},[]\backslash\uparrow.\texttt{l},[]\backslash\uparrow.\texttt{r}}\Diamond(\texttt{p} = q) \wedge \\ & \left(\bigtriangledown_{q,\backslash\texttt{p},[]\backslash\uparrow.\texttt{l},[]\backslash\uparrow.\texttt{r}}\Diamond(q^{\backslash}\uparrow.\texttt{d} = \backslash\texttt{x}) \Rightarrow \texttt{p}\uparrow.\texttt{d} = \backslash\texttt{x}\right) \wedge \\ & (\texttt{p} = \texttt{NIL} \vee \texttt{p}\uparrow.\texttt{d} = \backslash\texttt{x})\end{aligned}$$

Program P może mieć następującą postać:

```
WHILE (p ≠ NIL) AND (p↑.d ≠ x) DO
  IF p↑.d < x THEN
    p := p↑.l
  ELSE
    p := p↑.r
  END
END
```

Niech $\Lambda \subseteq MPL_\Sigma(X)$ bedzie zbiorem formuł prawdziwych w każdej algebrze częściowej należącej do $DStr(\Delta Sig(Int_{\text{tree}})^{\mathcal{D}})$, w której relacja $<$ jest liniowym porządkiem. Pokażemy, że $\Lambda \vdash \{\varphi_{\text{pre}}\}P\{\varphi_{\text{post}}\}$. W tym celu, korzystając z reguły dekoracji, wystarczy pokazać że:

$$\left\{ \begin{pmatrix} \varphi_{\text{bst}} \wedge \varphi_{\text{var}} \wedge \mathit{Chng}(*) \wedge \\ \left(\mathtt{p} \neq \mathtt{NIL} \Rightarrow \begin{pmatrix} (\mathtt{p}\uparrow)\downarrow \ \wedge \\ \forall_{t:\mathtt{tree}} \begin{pmatrix} t.\mathtt{l} \neq \mathtt{p} \wedge \\ t.\mathtt{r} \neq \mathtt{p} \end{pmatrix} \end{pmatrix} \right) \end{pmatrix} \right\} P \left\{ \varphi_{\text{post}} \right\} \ .$$

Niech $\sigma : \Sigma \to \Sigma$ będzie morfizmem sygnatur zmieniającym dekoracje $\backprime$ na $*$. Następnie korzystamy z reguły osłabiania, otrzymując następującą formułę:

$$\left\{ \begin{pmatrix} \varphi_{\text{bst}} \wedge \varphi_{\text{var}} \wedge \sigma(\varphi_{\text{nc}}) \wedge \\ (\mathtt{p} \neq \mathtt{NIL} \Rightarrow (\mathtt{p}\uparrow)\downarrow) \wedge \\ \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q = \mathtt{p}) \wedge \\ \begin{pmatrix} \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \Rightarrow \\ \bigtriangledown_{q,\mathtt{p},[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \end{pmatrix} \end{pmatrix} \right\} P \left\{ \begin{pmatrix} \varphi_{\text{bst}} \wedge \varphi_{\text{var}} \wedge \sigma(\varphi_{\text{nc}}) \wedge \\ (\mathtt{p} \neq \mathtt{NIL} \Rightarrow (\mathtt{p}\uparrow)\downarrow) \wedge \\ \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q = \mathtt{p}) \wedge \\ \begin{pmatrix} \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \Rightarrow \\ \bigtriangledown_{q,\mathtt{p},[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \end{pmatrix} \wedge \\ \mathit{OK}((\mathtt{p} \neq \mathtt{NIL}) \ \mathtt{AND} \ (\mathtt{p}\uparrow.\mathtt{d} \neq \mathtt{x})) \wedge \\ (\mathtt{p} = \mathtt{NIL} \vee \mathtt{p}\uparrow.\mathtt{d} = \mathtt{x}) \end{pmatrix} \right\} \ .$$

Opieramy się przy tym na następujących obserwacjach. Zauważmy, że $\mathit{Chng}(*)$ implikuje $\sigma(\varphi_{\text{nc}})$ oraz dwa ostatnie człony koniunkcji w warunku początkowym. Klauzula $\mathit{Chng}(*)$ pozwala nam w warunku końcowym na zamianę dekoracji $\backprime$ na $*$ i vice versa. Zauważmy też, iż z tego, że $\mathtt{p} = \mathtt{NIL} \vee \mathtt{p}\uparrow.\mathtt{d} = \mathtt{x}$ oraz

$$\bigtriangledown_{q,\backprime\mathtt{p},[]\backprime\uparrow.\mathtt{l},[]\backprime\uparrow.\mathtt{r}} \Diamond(q\backprime\uparrow.\mathtt{d} = \backprime\mathtt{x}) \Rightarrow \bigtriangledown_{q,\mathtt{p},[]\backprime\uparrow.\mathtt{l},[]\backprime\uparrow.\mathtt{r}} \Diamond(q\backprime\uparrow.\mathtt{d} = \backprime\mathtt{x})$$

wynika, że

$$\bigtriangledown_{q,\backprime\mathtt{p},[]\backprime\uparrow.\mathtt{l},[]\backprime\uparrow.\mathtt{r}} \Diamond(q\backprime\uparrow.\mathtt{d} = \backprime\mathtt{x}) \Rightarrow \mathtt{p}\uparrow.\mathtt{d} = \backprime\mathtt{x} \ .$$

Dodatkowo, z φ_{nc} wynika, że $x = \backprime x$.

Zauważmy, że powyższy warunek początkowy jest niezmiennikiem pętli programu. Przez I oznaczmy program tworzący jej wnętrze. Korzystając z reguły pętli wystarczy pokazać, że:

$$\left\{ \begin{pmatrix} \varphi_{\text{bst}} \wedge \varphi_{\text{var}} \wedge \sigma(\varphi_{\text{nc}}) \wedge \\ (\mathtt{p} \neq \mathtt{NIL} \Rightarrow (\mathtt{p}\uparrow)\downarrow) \wedge \\ \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q = \mathtt{p}) \wedge \\ \begin{pmatrix} \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \Rightarrow \\ \bigtriangledown_{q,\mathtt{p},[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \end{pmatrix} \wedge \\ \mathit{OK}((\mathtt{p} \neq \mathtt{NIL}) \ \mathtt{AND} \ (\mathtt{p}\uparrow.\mathtt{d} \neq \mathtt{x})) \wedge \\ \mathtt{p} \neq \mathtt{NIL} \wedge \mathtt{p}\uparrow.\mathtt{d} \neq \mathtt{x} \end{pmatrix} \right\} I \left\{ \begin{pmatrix} \varphi_{\text{bst}} \wedge \varphi_{\text{var}} \wedge \sigma(\varphi_{\text{nc}}) \wedge \\ (\mathtt{p} \neq \mathtt{NIL} \Rightarrow (\mathtt{p}\uparrow)\downarrow) \wedge \\ \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q = \mathtt{p}) \wedge \\ \begin{pmatrix} \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \Rightarrow \\ \bigtriangledown_{q,\mathtt{p},[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \end{pmatrix} \end{pmatrix} \right\} \ .$$

Następnie korzystamy z reguły instrukcji warunkowej. Mamy do udowodnienia dwie formuły odpowiadające dwóm członom instrukcji warunkowej. Pierwsza z nich jest postaci:

$$\left\{ \begin{pmatrix} \varphi_{\text{bst}} \wedge \varphi_{\text{var}} \wedge \sigma(\varphi_{\text{nc}}) \wedge \\ (\mathtt{p} \neq \mathtt{NIL} \Rightarrow (\mathtt{p}\uparrow)\downarrow) \wedge \\ \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q = \mathtt{p}) \wedge \\ \begin{pmatrix} \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \Rightarrow \\ \bigtriangledown_{q,\mathtt{p},[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \end{pmatrix} \wedge \\ \mathit{OK}((\mathtt{p} \neq \mathtt{NIL}) \ \mathtt{AND} \ (\mathtt{p}\uparrow.\mathtt{d} \neq \mathtt{x})) \wedge \\ \mathtt{p} \neq \mathtt{NIL} \wedge \mathtt{p}\uparrow.\mathtt{d} \neq \mathtt{x} \wedge \\ \mathit{OK}(\mathtt{p}\uparrow.\mathtt{d} < \mathtt{x}) \wedge \mathtt{p}\uparrow.\mathtt{d} < \mathtt{x} \end{pmatrix} \right\} \mathtt{p}{:=}\mathtt{p}\uparrow.\mathtt{l} \left\{ \begin{pmatrix} \varphi_{\text{bst}} \wedge \varphi_{\text{var}} \wedge \sigma(\varphi_{\text{nc}}) \wedge \\ (\mathtt{p} \neq \mathtt{NIL} \Rightarrow (\mathtt{p}\uparrow)\downarrow) \wedge \\ \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q = \mathtt{p}) \wedge \\ \begin{pmatrix} \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow.\mathtt{d} = \mathtt{x}^*) \Rightarrow \\ \bigtriangledown_{q,\mathtt{p},[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow.\mathtt{d} = \mathtt{x}^*) \end{pmatrix} \end{pmatrix} \right\} \ .$$

Stosując regułę osłabiania uzyskujemy aksjomat przypisania postaci:

$$\left\{\begin{array}{l}\varphi_{\mathrm{bst}} \wedge \varphi_{\mathrm{var}} \wedge \sigma(\varphi_{\mathrm{nc}}) \wedge \\ (\mathtt{p} \neq \mathtt{NIL} \Rightarrow (\mathtt{p}\uparrow)\downarrow) \wedge \\ \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q = \mathtt{p}) \wedge \\ \left(\begin{array}{l}\bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \Rightarrow \\ \bigtriangledown_{q,\mathtt{p},[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*)\end{array}\right) \wedge \\ \mathtt{p}\uparrow.\mathtt{d} < \mathtt{x}\end{array}\right\} \mathtt{p}{:=}\mathtt{p}\uparrow.\mathtt{l} \left\{\begin{array}{l}\backprime\varphi_{\mathrm{bst}} \wedge \varphi_{\mathrm{var}} \wedge \backprime\sigma(\varphi_{\mathrm{nc}}) \wedge \\ (\backprime\mathtt{p} \neq \mathtt{NIL} \Rightarrow (\backprime\mathtt{p}\backprime\uparrow)\downarrow) \wedge \\ \bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q = \backprime\mathtt{p}) \wedge \\ \left(\begin{array}{l}\bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \Rightarrow \\ \bigtriangledown_{q,\backprime\mathtt{p},[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*)\end{array}\right) \wedge \\ \backprime\mathtt{p}\backprime\uparrow.\mathtt{d} < \backprime\mathtt{x} \wedge (\backprime\mathtt{p})\downarrow \wedge (\backprime\mathtt{p}\backprime\uparrow.\mathtt{l})\downarrow \ \wedge \\ \mathtt{p} = \backprime\mathtt{p}\backprime\uparrow.\mathtt{l} \wedge \varphi_{\mathrm{nc}}\end{array}\right\}.$$

Korzystamy tu z następujących obserwacji. Zauważmy, że z φ_{nc} i $\backprime\sigma(\varphi_{\mathrm{nc}})$ wynika $\sigma(\varphi_{\mathrm{nc}})$. Podobnie, φ_{nc} i $\backprime\varphi_{\mathrm{bst}}$ implikują φ_{bst}. Niezmiennik drzew BST, φ_{nc} oraz $\mathtt{p} = \backprime\mathtt{p}\backprime\uparrow.\mathtt{l}$ pociągają za sobą $\mathtt{p} \neq \mathtt{NIL} \Rightarrow (\mathtt{p}\uparrow)\downarrow$. Warunek $\bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q = \mathtt{p})$ wynika z $\mathtt{p} = \backprime\mathtt{p}\backprime\uparrow.\mathtt{l}$, $\backprime\sigma(\varphi_{\mathrm{nc}})$ oraz $\bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q = \backprime\mathtt{p})$. Zauważmy też, że

$$\bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow.\mathtt{d} = \mathtt{x}^*) \Rightarrow \bigtriangledown_{q,\mathtt{p},[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow.\mathtt{d} = \mathtt{x}^*)$$

wynika z

$$\begin{array}{l}\bigtriangledown_{q,\mathtt{p}^*,[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \Rightarrow \bigtriangledown_{q,\backprime\mathtt{p},[]\uparrow^*.\mathtt{l},[]\uparrow^*.\mathtt{r}} \Diamond(q\uparrow^*.\mathtt{d} = \mathtt{x}^*) \wedge \\ \backprime\mathtt{p}\backprime\uparrow.\mathtt{d} < \backprime\mathtt{x} \wedge \backprime\sigma(\varphi_{\mathrm{nc}}) \wedge \sigma(\varphi_{\mathrm{nc}}) \wedge \varphi_{\mathrm{bst}}\end{array},$$

oraz stąd, że $<$ jest porządkiem liniowym.

Formułę odpowiadającą drugiemu członowi instrukcji warunkowej dowodzimy podobnie. □

Powyższy przykład pokazuje, że nawet w przypadku stosunkowo prostej struktury wskaźnikowej i bardzo prostego programu dowód poprawności jest dosyć skomplikowany. Stawia to pod znakiem zapytania możliwość zastosowania *HMPL* w praktyce. Problem ten dotyczy chyba wszystkich metod weryfikacji programów — dowody poprawności są zbyt skomplikowane, aby stosowano je w praktyce. W powyższym przykładzie zwracają uwagę dosyć złożone asercje. Mimo, że *MPL* pozwala na bardziej zwięzły zapis asercji, to nadal są one bardziej skomplikowane, niż byśmy tego chcieli. Wydaje się więc, że do praktycznego zastosowania przedstawionego formalizmu jest niezbędne zastosowanie narzędzi wspomagających. Przydatne mogą być tu zarówno narzędzia wspomagające dowodzenie twierdzeń (takie jak np. PVS [COR+95, OSR93], HOL [Uni94] czy Isabelle [Pau95a, Pau95b]) jak i systemy wspomagające dowodzenie poprawności programów (takie jak np. CaProDel [DŁMJ94]).

Rozdział 6

Zakończenie

Praca ta stanowi krok ku połączeniu metod formalnych specyfikacji oraz efektywnych algorytmów i struktur danych. Analizuje ona problem specyfikacji wskaźnikowych struktur danych. Nasze rozważania zostały osadzone w ramach podejścia funkcyjnego.

Pokazaliśmy, w jaki sposób stany i modyfikacje wskaźnikowych struktur danych można modelować za pomocą algebr wielosortowych z funkcjami częściowymi. Pozwala to na sprowadzenie problemu specyfikowania wskaźnikowych struktur danych do specyfikowania reprezentujących je algebr.

Zbadaliśmy przydatność kilku znanych formalizmów do specyfikowania wskaźnikowych struktur danych. Zaproponowaliśmy również własny formalizm — nazywany logiką wielościeżkową — pozwalający na zwięzłe i względnie czytelne opisywanie wskaźnikowych struktur danych. W tym formalizmie zastosowano operatory modalne do opisu połączeń wskaźnikowych łączących rekordy tworzące strukturę danych. Operatory te pozwalają uchwycić podstawowe własności „grafowe" — takie, jak spójność czy acykliczność powiązań wskaźnikowych. Jak pokazują przykłady, dzięki temu specyfikacje wskaźnikowych struktur danych są bardziej zwięzłe i czytelne. Logika wielościeżkowa pozwala też na zestawienie ze sobą kolejnych elementów tworzących różne ścieżki. Daje jej to odpowiednią siłę wyrazu i pozwala to na uchwycenie takich własności, jak np. kształty drzew AVL.

Dokonaliśmy porównania badanych formalizmów pod kątem ich przydatności do specyfikowania wskaźnikowych struktur danych. Z natury rzeczy porównanie to nie jest wyczerpujące. Wskazuje jednak ono wyraźnie, że spośród badanych formalizmów najlepszym kandydatem do specyfikowania wskaźnikowych struktur danych jest logika wielościeżkowa. Mimo to musimy jednak przyznać, że specyfikacje zapisane w tym formalizmie są bardziej skomplikowane niż byśmy tego chcieli. Logika wielościeżkowa z pewnością nie jest ostatnim słowem w dyskusji na temat specyfikacji wskaźnikowych struktur danych. Jednak z rozważań zawartych w tej pracy wynika, że odpowiednie zastosowanie operatorów modalnych pozwala na równoczesne osiągnięcie odpowiedniej siły wyrazu oraz większą zwięzłość specyfikacji.

Przedstawiliśmy również wariant logiki Hoare'a pozwalający na weryfikowanie *while*-programów względem asercji wyrażonych w logice wielościeżkowej. Zaproponowany system dowodzenia posiada podstawowe własności poprawności i pełności w sensie Cook'a. Jest to więc elementarne narzędzie do weryfikowania programów operujących na wskaźnikowych strukturach danych względem specyfikacji zapisanych w zaproponowanej przez nas logice. Jego zastosowanie zostało pokazane na prostym przykładzie. Niestety z przykładu tego wynika, że przedstawiony formalizm jest zbyt skomplikowany, aby go stosować w praktyce. Problem

ten dotyczy zresztą wszystkich metod dowodzenia poprawności programów — dowody są zbyt skomplikowane, aby były stosowane dla dużych programów. Niezbędne wydają się zarówno opracowanie narzędzi wspomagających weryfikację oprogramowania jak i praca nad dalszym uproszczeniem specyfikacji i weryfikacji programów.

Nie wszystkie idee prezentowane w pracy są oryginalnym wkładem autora. Zaproponowana w pracy logika wielościeżkowa w dużej mierze opiera się na logice zaproponowanej przez Costę i Reggio w [CR97]. Przedstawiona wersja logiki Hoare'a powstała po przeanalizowaniu różnych wersji takich logik znanych z literatury. Dowody poprawności i pełności w sensie Cook'a prezentowanego systemu dowodzenia są wzorowane na standardowych dowodach tych własności zawartych w pracy [Apt81]. Do oryginalnych osiągnięć niniejszej rozprawy można zaliczyć ideę wykorzystania operatorów modalnych do opisu powiązań wskaźnikowych tworzących strukturę danych, analizę różnych formalizmów pod kątem ich przydatności do specyfikowania stosowanych wskaźnikowych struktur danych, a także wykorzystanie wszystkich tych idei przy opracowaniu logiki wielościeżkowej oraz prezentowanego wariantu logiki Hoare'a wraz z systemem dowodzenia.

Niniejsza praca nie wyczerpuje oczywiście problematyki specyfikacji wskaźnikowych struktur danych. Dalsze badania powinny się koncentrować na kilku tematach. Przede wszystkim należy zbadać na innych przykładach praktyczną użyteczność propozycji zawartych w tej pracy. Otwarty pozostaje także problem weryfikacji specyfikacji wskaźnikowych struktur danych względem specyfikacji abstrakcyjnych typów danych. W ramach przyjętego przez nas podejścia funkcyjnego problem ten sprowadza się głównie do problemu definiowania funkcji abstrakcji dla wskaźnikowych struktur danych.

Literatura

[AL91] M. Abadi, L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[Apt81] K. R. Apt. Ten years of Hoare's logic: A survey — part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, październik 1981.

[Bał] A. Bałaban. *Wnioskowanie o programach za pomocą rozszerzenia logiki algorytmicznej.* Rozprawa doktorska, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki. W przygotowaniu.

[Bas96] D. A. Basin. Verification based on monadic logic. BRICS autumn school on verification. BRICS Notes Series NS-96-3, BRICS, Department of Computer Science, University of Aarhus, Aarhus, Denmark, październik 1996.

[BDR96] L. Banachowski, K. Diks, W. Rytter. *Algorytmy i struktury danych.* Wydawnictwa Naukowo-Techniczne, 1996.

[BH97] R. Bharadwaj, C. Heitmeyer. Model checking complete requirements specifications using abstraction. Memorandum Report NRL/MR/5540–97-7999, United States Navy, Naval Research Laboratory, Washington, DC, USA, 1997.

[BIMO94] J. Bojanowski, M. Iglewski, J. Madey, A. Obaid. Functional approach to protocols specification. W: *Proceedings of the 14th International IFIP Symposium on Protocol Specification, Testing and Verification, PSTV'94, Vancouver, B.C.*, str. 371–378, 1994.

[BKR87] L. Banachowski, A. Kreczmar, W. Rytter. *Analiza algorytmow i struktur danych.* Wydawnictwa Naukowo-Techniczne, 1987.

[BP78] W. Bartussek, D. L. Parnas. Using traces to write abstract specifications for software modules. W: *Proceedings of 2nd Conference of European Cooperation in Informatics*, LNCS, nr 65. Springer-Verlag, 1978.

[BP81] K. H. Britton, D. L. Parnas. A-7E software module guide. Memorandum Report NRL-4702, United States Navy, Naval Research Laboratory, Washington, DC, USA, 1981.

[BS95] H. Bickel, W. Struckmann. The Hoare logic of data types. Raport techniczny 95-04, Technische Universität Braunschweig, Deutschland, 1995.

[CLR97] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Wprowadzenie do algorytmów.* Wydawnictwa Naukowo-Techniczne, 1997.

[CO81] R. Cartwright, D. Oppen. The logic of aliasing. *Acta Informatica*, 15:365–384, 1981.

[Coo78] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, M. Srivas. A tutorial introduction to PVS. `http://www.csl.sri.com/sri-csl-fm.html`, kwiecień 1995.

[Cou90] B. Courcelle. *Graph Rewriting: An Algebraic and Logic Approach*, tom B, rozdział 5, str. 193–242. Elsevier, 1990.

[CP93] F. Courtois, D. L. Parnas. Formally specifying a communications protocol using the trace assertion method. CRL Report No. 269, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1993.

[CR97] G. Costa, G. Reggio. Specification of abstract dynamic-data types: A temporal approach. *Theoretical Computer Science*, 173(2):513–554, 1997.

[Dah92] O.-J. Dahl. *Verifiable Programming.* Prentice Hall, 1992.

[Dij85] E. W. Dijkstra. *Umiejętność programowania.* Wydawnictwa Naukowo-Techniczne, 1985.

[DŁMJ94] B. Dunin-Kęplicz, W. Łukaszewicz, E. Madalińska-Bugaj, J. Jabłonowski. CaProDel: A system for computer-aided program development. W: *Proceedings of the Sixth International Conference on Software Engineering and Knowledge Engineering, SEKE'94*, Jurmala, Latvia, 1994.

[EF95] H.-D. Ebbinghaus, J. Flum. *Finite Model Theory.* Springer-Verlag, 1995.

[EKM+93] M. Engel, M. Kubica, J. Madey, D. L. Parnas, A. P. Ravn, A. J. van Schouwen. A formal approach to computer systems requirements documentation. W: R. L. Grossman, A. Nerode, A. P. Ravn, H. Rischel, ed., *Hybrid Systems*, LNCS, nr 736, str. 452–474. Springer-Verlag, 1993.

[EM85] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specificaton 1.* EATCS MTCS, nr 6. Springer-Verlag, 1985.

[Eng94] J. Engelfriet. Graph grammars and tree transducers. W: S. Tison, ed., *Trees in Algebra and Programming — CAAP'94. Proceedings*, LNCS, nr 787, str. 15–36. Springer-Verlag, 1994.

[FL90] Y. Feng, J. Liu. A temporal approach to algebraic specifications. W: J. C. M. Baeten, J. W. Klop, ed., *CONCUR'90: Theories of Concurrency: Unification and Extension*, LNCS, nr 458, str. 216–229. Springer-Verlag, 1990.

[GH93] J. V. Guttag, J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[GHH+92] C. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, K. R. Wagner. *The RAISE Specification Language*. Prentice Hall, 1992.

[GHH+95] C. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, J. S. Pedersen. *The RAISE Development Method*. Prentice Hall, 1995.

[Gut77] J. V. Guttag. Abstract data types and the development of data structure. *Communications of the ACM*, 20(6):396–404, 1977.

[HJL96] C. Heitmeyer, R. D. Jeffords, B. G. Labaw. Automated consistency checking of requirements specification. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(1):271–281, 1972.

[IKM95a] M. Iglewski, M. Kubica, J. Madey. Trace specifications of non-deterministic multi-object modules. W: K. Kanchacasut, J.-J. Levy, ed., *Agorithms, Concurrency and Knowledge. Proceedings, 1995*, LNCS, nr 1023, str. 381–395. Springer-Verlag, 1995.

[IKM+95b] M. Iglewski, M. Kubica, J. Madey, J. Mincer-Daszkiewicz, K. Stencel. The Fun-project: From requirements specification to program presentation. Raport techniczny TR 95-18 (218), Instytut Informatyki, Uniwersytet Warszawski, Warszawa, 1995.

[IKM+97] M. Iglewski, M. Kubica, J. Madey, J. Mincer-Daszkiewicz, K. Stencel. TAM'97: the trace assertion method of module interface specification. Reference manual. Raport techniczny TR 97-01 (238), Instytut Informatyki, Uniwersytet Warszawski, Warszawa, 1997.

[IMD97] M. Iglewski, J. Mincer-Daszkiewicz. Internal design of modules specified in the trace assertion method. *Science of Computer Programming*, 28(2–3):139–170, 1997.

[IMM92] M. Iglewski, J. Madey, S. Matwin. *Pascal. Standard*. Wydawnictwa Naukowo-Techniczne, 1992.

[IMPK93] M. Iglewski, J. Madey, D. L. Parnas, P. C. Kelly. Documentation paradigms (a progress report). CRL Report No. 270, McMaster University, CRL, Telecommunication Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1993.

[IMS94] M. Iglewski, J. Madey, K. Stencel. On fundamentals of the trace assertion method. Raport techniczny TR 94-09, Instytut Informatyki, Uniwersytet Warszawski, Warszawa, 1994.

[Jan95] R. Janicki. On foundations of the trace assertion method. Raport techniczny 95-04, Department of Computer Science and Systems, McMaster University, Hamilton, Ontario, Canada, 1995.

[JJKS97] J. L. Jensen, M. E. Jøergensen, N. Klarlund, M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. *ACM SIGPLAN Notices*, 32(5):226–234, 1997.

[Jon84] C. B. Jones. *Konstruowanie oprogramowania metodą systematyczną*. Wydawnictwa Naukowo-Techniczne, 1984.

[JvEB77] T. M. V. Janssen, P. van Emde Boas. On the proper treatment of referencing, dereferencing and assignment. W: G. Goos, J. Hartmanis, ed., *Automata, Languages and Programming*, LNCS, nr 52, str. 282–300. Springer-Verlag, 1977.

[JW76] K. Jensen, N. Wirth. *Pascal. User Manual and Report.* LNCS, nr 18. Springer-Verlag, 1976.

[KS93] N. Klarlund, M. I. Schwartzbach. Graph types. W: *Proceedings of the 20th Symposium on Principles of Programming Languages*, str. 196–205. ACM, 1993.

[KS94] N. Klarlund, M. I. Schwartzbach. Graphs and decidable transductions based on edge constraints. W: S. Tison, ed., *Trees in Algebra and Programming — CAAP'94. Proceedings*, LNCS, nr 787, str. 187–201. Springer-Verlag, 1994.

[Kub94] M. Kubica. Metoda tropów — analiza i propozycje modyfikacji. Praca magisterska, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki, 1994.

[Lip89] W. Lipski. *Kombinatoryka dla programistów.* Wydawnictwa Naukowo-Techniczne, 1989.

[MS87] G. Mirkowska, A. Salwicki. *Algorithmic Logic.* Państwowe Wydawnictwo Naukowe, 1987.

[MS92] G. Mirkowska, A. Salwicki. *Logika algorytmiczna dla programistów.* Wydawnictwa Naukowo-Techniczne, 1992.

[Mye80] G. J. Myers. *Projektowanie niezawodnego oprogramowania.* Wydawnictwa Naukowo-Techniczne, 1980.

[OSR93] S. Owre, N. Shankar, J. Rushby. *User Guide for the PVS Specification and Verification System, Language, and Proof Checker (Beta Release).* Computer Science Laboratory, SRI International, Menlo Park, CA, USA, 1993.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, grudzień 1972.

[Pau95a] L. C. Paulson. *Introduction to Isabelle.* Computer Laboratory, University of Cambridge, 1995.

[Pau95b] L. C. Paulson. *The Isabelle Reference Manual.* Computer Laboratory, University of Cambridge, 1995.

[PCW85] D. L. Parnas, P. C. Clements, D. M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, 11:259–266, 1985.

[PM90] D. L. Parnas, J. Madey. Functional documentation for computer systems engineering. Raport techniczny 90-287, Queen's University, C&IS, Telecommunications Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, 1990.

[PM91] D. L. Parnas, J. Madey. Functional documentation for computer systems engineering. (Version 2). CRL Report 90-287, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1991.

[PM92] D. L. Parnas, J. Madey. Documentation of real-time requirements. W: K. M. Kavi, ed., *Real-Time Systems, Abstraction, Languages, and Design Methodologies*, str. 48–56. IEEE Computer Society Press, 1992.

[PM95] D. L. Parnas, J. Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995.

[PW89] D. L. Parnas, Y. Wang. The trace assertion method of module interface specification. Raport techniczny 89-261, Queen's University, C&IS, Telecommunication Research Institute of Ontario (TRIO), Kingston, Ontario, Canada, 1989.

[Ras84] H. Rasiowa. *Wstęp do matematyki współczesnej*, tom 30, *Biblioteka Matematyczna*. Państwowe Wydawnictwo Naukowe, 1984.

[RND85] E. M. Reingold, J. Nievergelt, N. Deo. *Algorytmy kombinatoryczne*. Państwowe Wydawnictwo Naukowe, 1985.

[Spi92] J. M. Spivey. *The Z notation, A Reference Manual*. Second edition. Prentice-Hall, 1992.

[ST] D. Sannella, A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development*. Cambridge University Press. to appear.

[Ste99] K. Stencel. *Abstrakcyjne specyfikacje z jawnym nośnikiem modelu*. Rozprawa doktorska, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki, 1999.

[Str97] B. Stroustrup. *Język C++*. Wydawnictwa Naukowo-Techniczne, trzecie wydanie, 1997.

[Sza88] A. Szałas. Towards the temporal approach to abstract data types. *Fundamenta Informaticae*, XI:49–63, 1988.

[Tho90] W. Thomas. Automata on infinite objects. W: *Handbook of Theoretical Computer Science*, tom B, rozdział 4, str. 133–191. Elsevier, 1990.

[Tho97] W. Thomas. Languages, automata, and logic. W: *Handbook of Formal Languages*, tom 3, rozdział 7. Springer-Verlag, 1997.

[Tiu98] J. Tiuryn. *Wstęp do teorii mnogości i logiki*. Wydział Matematyki, Informatyki i Mechaniki, Uniwersytet Warszawski, 1998. Skrypt.

[Uni94] University of Cambridge and DSTO and SRI International. *The HOL System tutorial*, version 2 wydanie, 1994.

[vSPM93] A. J. van Schouwen, D. L. Parnas, J. Madey. Documentation of requirements for computer systems. W: *Proceedings of the IEEE International Symposium on Requirements Engineering*, str. 198–207, San Diego, California, USA, 1993.

[Wan94] Y. Wang. Specifying and simulating the externally observable behavior of modules. CRL Report 292, McMaster University, CRL, Telecommunications Research Institute of Ontario (TRIO), Hamilton, Ontario, Canada, 1994.

[Wil85] R. J. Wilson. *Wprowadzenie do teorii grafów*. Państwowe Wydawnictwo Naukowe, 1985.

[Wir90] M. Wirsing. Algebraic specifications. W: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, tom B, rozdział 13, str. 676–788. Elsevier, 1990.

[Wir91] N. Wirth. *Modula 2*. Wydawnictwa Naukowo-Techniczne, 1991.

[Wir99] N. Wirth. *Algorytmy + struktury danych = programy*. Wydawnictwa Naukowo-Techniczne, trzecie wydanie, 1999.

[Wor92] J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.