Specyfikacja i weryfikacja programów operujących na wskaźnikowych strukturach danych

Adam Bałaban, Marcin Kubica*
Instytut Informatyki, Uniwersytet Warszawski

Streszczenie

W niniejszej pracy zajmujemy się problemem specyfikacji i weryfikacji programów operujących na wskaźnikowych strukturach danych. Rozważamy dwa podejścia: oparte o logikę Hoare'a i logikę algorytmiczną. Jako pierwszy przedstawiamy formalizm służący do wyrażania asercji dotyczących wskaźnikowych struktur danych oraz wariant logiki Hoare'a. Następnie prezentujemy rozszerzenie logiki algorytmicznej o mechanizmy umożliwiające wnioskowanie o programach korzystających ze wskaźników.

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

*Praca finansowana z grantu KBN nr 8 T11C 015 15

zwykłym biurze. Dlatego też błędy w oprogramowaniu stawiają pod znakiem zapytania jego wartość.

Drogą do uzyskania niezawodnego oprogramowania jest zapewnienie jego poprawności na wszystkich etapach wytwarzania. 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.

Bardzo wiele systemów formalnych powstało na użytek weryfikacji programów. Do najbardziej znanych należą: logika Hoare'a [9] w olbrzymiej liczbie wersji i rozszerzeń (por. [6] oraz [1]), rachunek tzw. *transformacji predykatów* Dijkstry [7] i jego odmiany (por. np. [14]), czy wreszcie logika dynamiczna [8] z jej równie dużą liczbą odmian (por. [10]). Gdy jednak porównamy stosowane techniki programistyczne oraz znane metody specyfikacji, możemy zauważyć pewną rozbieżność. Efektywne algorytmy, por. np. [3, 4, 16], bardzo często wykorzystują wskaźnikowe struktury danych. Z drugiej strony metody specyfikacji programów pomijają kwestię wskaźników lub traktują ją w sposób niewystarczający. Znane są próby opisywania wskaźników, często jednak pozwalają one jedynie na wyrażanie własności struktur danych w logice pierwszego rzędu. Podejścia takie mają ograniczone zastosowanie. Zauważmy, że logika pierwszego rzędu nie pozwala na wyrażenie takich podstawowych własności, jak spójność grafu, czy istnienie ścieżki o zadanych końcach. Dlatego też nie można za jej pomocą opisać nawet tak podstawowej struktury danych, jak listy jednokierunkowe.

W niniejszej pracy prześledzimy problem specyfikacji i weryfikacji wskaźnikowych struktur danych i programów na nich operujących na przykładzie dwóch podejść: logiki Hoare'a i logiki algorytmicznej. W punkcie 2. przedstawiamy propozycję logiki służącej do wyrażania asercji dotyczących wskaźnikowych struktur danych oraz wariant logiki Hoare'a dostosowany do weryfikowania programów operujących na wskaźnikowych strukturach danych względem asercji wyrażonych w prezentowanym formalizmie. W punkcie 3. przedstawiamy rozszerzenie logiki algorytmicznej o mechanizmy umożliwiające wnioskowanie o programach korzystających ze wskaźników. Prezentowane formalizmy są ilustrowane prostymi przykładami.

2. Hoare'owskie specyfikacje wskaźnikowych struktur danych

W niniejszym punkcie przedstawimy po krótce propozycję podejścia do specyfikacji wskaźnikowych struktur danych opartego na logice Hoare'a. Przedstawione podejście obejmuje propozycję języka asercji oraz wariant logiki Hoare'a. Szczegółowy opis prezentowanego podejścia można znaleźć w [11].

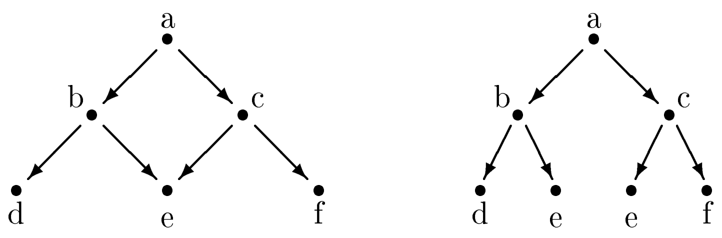
2.1. Język asercji

Okazuje się, że logika pierwszego rzędu jest zbyt słaba, aby opisywać wskaźnikowe struktury danych. Jak wiadomo, nie można wyrazić w niej takich własności, jak spójność grafu, czy istnienie ścieżki łączącej dwa określone wierzchołki (zobacz np. [15]). Uniemożliwia to opisanie najprostszej struktury wskaźnikowej jaką jest lista jednokierunkowa. Dlatego też, chcąc zastosować logikę Hoare'a do weryfikacji programów operujących na wskaźnikowych strukturach danych musimy użyć jako języka asercji silniejszej logiki.

Własności, których nie możemy wyrazić w logice pierwszego rzędu mają charakter grafowy. Jeżeli wyobrazimy sobie wskaźnikową strukturę danych jako graf, którego wierzchołkami są tworzące ją rekordy, a wskaźniki tworzą krawędzie, to zwykle chcemy wyrażać takie własności, jak osiągalność wierzchołków, istnienie tylko jednej ścieżki między dwoma wierzchołkami, czy nie przecinanie się dwóch ścieżek. Własności tego rodzaju można wyrazić w logice drugiego rzędu, czy nawet logice monadycznej drugiego rzędu. Jednak specyfikacje takie są bardzo złożone i nieczytelne.

Proponowany język asercji opiera się na analogii między strukturami wskaźnikowymi, a strukturami Kripkego. Rekordy tworzące struktury wskaźnikowe traktujemy jak „światy", a wskaźniki jak relację przejścia między światami. Przy takim spojrzeniu, pojęcie historii w strukturach Kripkego odpowiada ścieżce złożonej ze wskaźników łączących rekordy. Można by więc użyć do opisu struktur wskaźnikowych logik temporalnych.

Istnieje jednak pewien problem. Logiki temporalne służą do opisu możliwych ciągów zdarzeń. Dlatego też klasyczne logiki temporalne nie odróżnią dwóch struktur przedstawionych na poniższym rysunku — w obydwu z nich, będąc w wierzchołku a mamy przed sobą takie same możliwe historie. Z drugiej strony, jeżeli chcemy opisywać struktury wskaźnikowe, np. drzewa binarne, to tylko druga z przedstawionych struktur jest poprawnym drzewem.



Problem ten można rozwiązać adaptując wariant logiki CTL* przedstawiony w

[5]. W adaptacji tej kwantyfikując formuły ścieżkowe jawnie określamy jakiego typu rekordy tworzą światy i które z łączących je wskaźników tworzą relację przejścia między światami. Jednocześnie, wewnątrz formuł ścieżkowych mamy możliwość „uchwycenia" aktualnego świata na zmienną, za pomocą formuł postaci: $[\lambda x.\varphi(x)]$. Formalizm taki pozwala na opisanie takich struktur jak listy, drzewa i grafy. Jest jednak zbyt słaby do wyrażenia takich własności jak np. zrównoważenie drzew AVL. Tego rodzaju własności można opisywać dalej rozszerzając podany formalizm o możliwość „równoczesnego" poruszania się wzdłuż kilku ścieżek.

Dokładne przedstawienie postaci opracowanego formalizmu stanowczo wykracza poza ramy tego artykułu. Niemniej pozwalamy sobie przedstawić prosty przykład specyfikacji drzew binarnych, mając nadzieję, że będzie on zrozumiały dla czytelników zaznajomionych z logikami temporalnymi[1].

Przykład. Niezmiennik drzew binarnych:

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

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

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

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

$$\forall_{p:\texttt{tree}}\,\big((p\!\uparrow)\!\downarrow \Rightarrow \triangle_{p,[]\uparrow.\texttt{l},[]\uparrow.\texttt{p}}\Diamond[\lambda x.\ x = \texttt{nil}]\big) \wedge$$

Ważniejsze jednak od konkretnej postaci proponowanego formalizmu wydaje się to, że zastosowanie operatorów pochodzących z logik temporalnych pozwoliło na bardziej zwięzłe i intuicyjne opisywanie wskaźnikowych struktur danych. Analogia między historiami w strukturach Kripkego, a ścieżkami złożonymi ze wskaźników okazała się zgodna ze sposobem w jaki postrzegamy struktury wskaźnikowe.

---

[1] Formuła $t\downarrow$ oznacza, że wartość t jest dobrze określona. Symbole $\triangle$ i $\triangledown$ to, odpowiednio, uniwersalny i egzystencjalny kwantyfikator ścieżkowy. Pierwszy term występujący w indeksie kwantyfikatora wyznacza początek rozpatrywanych ścieżek, a kolejne termy (z dziurami) określają krawędzie ścieżek. Symbol $\Diamond$ to operator temporalny „kiedyś". Formuła postaci $[\lambda x.\varphi(x)]$ pozwala „uchwycić" na zmiennej x wartość „aktualnego świata na ścieżce".

2.2. Wariant logiki Hoare'a

Przedstawiona w poprzednim punkcie logika stanowi propozycję języka do specyfikacji wskaźnikowych struktur danych. Chcąc jednak weryfikować programy operujące na takich strukturach potrzebujemy czegoś więcej. W niniejszym punkcie przedstawiamy krótko odpowiedni wariant logiki Hoare'a.

Opracowując opisywany wariant trzeba było rozwiązać dwa problemy:

- opisanie pośredniego stanu struktury danych dla złożenia sekwencyjnego instrukcji, oraz
- opisanie operacji pierwotnych modyfikujących strukturę danych: przypisania, alokacji i zwalniania zmiennych dynamicznych.

W klasycznej logice Hoare'a możemy sobie radzić z pośrednimi wartościami zmiennych wprowadzając dodatkowe zmienne „matematyczne" reprezentujące takie pośrednie wartości zmiennych programistycznych. Jednak w przypadku wskaźnikowych struktur danych nie jesteśmy w stanie nazwać wszystkich zmiennych tworzących strukturę, musielibyśmy więc wprowadzić zmienne reprezentujące zawartości sterty. Podobnie, chcąc powiązać stan struktury danych sprzed i po wykonaniu programu musimy musimy powiązać ze sobą wartości zmiennych dynamicznych. Oznaczałoby to jednak wprowadzenie zmiennych drugiego rzędu — funkcji przyporządkowujących wskaźnikom wskazywane wartości. W poprzednim punkcie udało nam się uniknąć w języku asercji konstrukcji drugiego rzędu. Okazuje się, że można ich uniknąć również teraz. Rozwiązanie polega na tym, że nie musimy kwantyfikować stanów sterty. Nie muszą to więc być zmienne. Mogą to być funkcje wprowadzane w miarę potrzeb do sygnatury. Zastosowaliśmy tu konwencję nazywaną dekorowaniem. Symbol dereferencji wskaźników ($\uparrow$) traktujemy jak funkcję przyporządkowującą wskaźnikom wskazywane wartości. Jednocześnie dopuszczamy istnienie wielu takich funkcji o nazwach postaci $\uparrow^\delta$, gdzie δ jest dowolną dekoracją.

Dekorowanie funkcji $\uparrow$ jest prostym morfizmem sygnatur, możemy je więc rozszerzyć w na formuły — udekorowanie formuły oznacza udekorowanie wszystkich znajdujących się w niej symboli funkcji $\uparrow$.

Chcąc uchwycić pośredni stan struktury danych możemy użyć nowej dekoracji. Chcąc powiązać ze sobą stan struktury danych sprzed i po wykonaniu programu, w warunku końcowym reprezentujemy stan struktury danych sprzed wykonania programu używając dekoracji (tradycyjnie) zapisywanej jako $`\uparrow$.

Reguły naszej logiki mają następującą postać:

- reguła osłabiania:

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

  gdzie $\Lambda_1 \subseteq \Lambda$

- reguła wprowadzająca dekoracje:

$$\frac{\Lambda \vdash \{\varphi \wedge \mathit{Mem}_\delta\} P \{\psi\}}{\Lambda \vdash \{\varphi\} P \{\psi\}} \;,$$

  gdzie δ jest dekoracją nie występującą w φ lub ψ, różną od $\backslash$, a Mem_δ jest formułą pierwszego rzędu wyrażającą, że $\uparrow \equiv \uparrow^\delta$.

- reguła 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 $\backslash$, a σ jest morfizmem sygnatur zmieniającym dekoracje $\backslash$ na δ,

- reguła instrukcji warunkowej:

$$\frac{\Lambda \vdash \{\varphi \wedge \alpha\} P \{\psi\}, \Lambda \vdash \{\varphi \wedge \neg\alpha\} Q \{\psi\}}{\Lambda \vdash \{\varphi\} \texttt{if } \alpha \texttt{ then } P \texttt{ else } Q \ \{\psi\}}$$

- reguła pętli:

$$\frac{\Lambda \vdash \{\varphi \wedge \alpha\} P \{\varphi\}}{\Lambda \vdash \{\varphi\} \texttt{while } \alpha \texttt{ do } P \ \{\varphi \wedge \neg\alpha\}}$$

Aksjomaty opisujące operacje pierwotne różnią się tym od klasycznych aksjomatów, że działają „w przód", a nie „wstecz", tzn. dla danego warunku początkowego określają warunek końcowy spełniony po wykonaniu operacji. Rozwiązanie takie wybrano ze względu na zjawisko utożsamiania zmiennych o różnych nazwach (ang. *aliasing*) wynikające z użycia wskaźników. Aksjomaty te opisują przypisanie, alokowanie zmiennych dynamicznych oraz instrukcję pustą. Aksjomat operacji pustej ma postać:

$$\vdash \{\varphi\} \varepsilon \{{}^{\backslash}\varphi \wedge \mathit{Mem}_{\backslash}\} \;,$$

gdzie $\mathit{Mem}_{\backslash}$ jest formułą wyrażającą, że $\uparrow \equiv {}^{\backslash}\uparrow$. Aksjomaty opisujące przypisanie i alokowanie zmiennych dynamicznych pominiemy tutaj, gdyż zawierają one wiele technikalii.

Opisywany wariant logiki Hoare'a jest poprawny i pełny w sensie Cook'a. Dowody tych faktów można znaleźć w [11].

3. Logika algorytmiczna „ze wskaźnikami”

W tym punkcie, z uwagi na wielkość tej publikacji, przedstawimy jedynie krótki opis języka rozszerzonej logiki algorytmicznej. Szczegółowy opis tego formalizmu znajdzie się w pracy [2].

Logika alogrytmiczna zdefiniowana w pracach [12, 13] posługuje się formułami postaci $M(\alpha)$, gdzie M jest programem, zaś α formułą pierwszego rzędu. Formuła $M(\alpha)$ jest prawdziwa w określonej interpretacji, jeśli wykonanie programu M zakończy się pomyślnie i po tym wykonaniu formuła α jest spełniona. Logika algorytmiczna rozszerza więc klasyczną Logikę Hoare'a o możliwość wnioskowania o zakończeniu programu. Oba formalizmy są określone dla dość wąskiej klasy tzw. *w*hile-programów. Prezentowana logika (oznaczana w dalszym ciągu przez *C-AL*) została zdefiniowana dla klasy programów dopuszczających następujące konstrukcje oraz operacje:

- wskaźniki, w tym wskaźniki wielokrotne (wskaźniki do wskaźników),
- aliasy zmiennych,
- tablice i rekordy,
- przydział i zwalnianie pamięci (alokację i dealokację).

Przyjrzyjmy się teraz nieco bardziej szczegółowo poszczególnym konstrukcjom języka *C-AL*. Podczas prac nad tym formalizmem przekonaliśmy się, jak istotny jest sposób w jaki wskaźniki mają być implementowane w języku — jakiego modelu pamięci używamy, jaki jest zestaw dostępnych operacji itp. To wszystko razem dopiero decyduje o przydatności konkretnego formalizmu.

3.1. Język i jego semantyka

Klasa programów, które mogą występować w formułach, bazuje na składni i semantyce języka C. Wybraliśmy ten język m.in. dlatego, że w sposób bardzo elastyczny i jednocześnie zwięzły i konsekwentny operuje on wskaźnikami. Nie było jednak naszym celem zdefiniowanie pełnej semantyki języka C (z rzutowaniem typów i bogatym językiem wyrażeń z efektami ubocznymi).

Rozważmy teraz deklarację zmiennej `max` typu `int` w języku C.

```
int max;
```

Wyrażenie języka C postaci `&max` będzie wtedy oznaczać adres realizacji tej zmiennej w pamięci (typu `int*`). W naszej logice, zwykłej zmiennej nierejestrowej występującej w programie, np. `max`, będzie odpowiadać pojedyncza zmienna indywiduowa „$\&max$” (znak „&” traktujemy jako integralną część nazwy tej zmiennej), przechowująca adres zmiennej `max`. Aby więc odwołać się do wartości zmiennej programowej `max` języka C, w języku *C-AL* posługiwać się będziemy wyrażeniami postaci „$*(\&max)$", gdzie $*(\_)$ jest również odpowiednim operatorem referencji w logice. Przyjęta interpretacja gwarantuje zachowanie jednolitego podejścia do zmiennych i ich adresów realizacji.

Podobnie jak w języku C, w języku logiki *C-AL* będziemy używać pojęcia *l-wyrażenia*, czyli wyrażenia, które może znaleźć się po lewej stronie instrukcji przypisania. W tej roli mogą występować zarówno zmienne indywiduowe, jak i odpowiednio zbudowane wyrażenia adresowe (poprzedzone operatorem referencji), nazywane *l-wyrażeniem pamięciowym*.

Dodatkowo niezwykle użyteczne okazały się być zmienne rejestrowe (postaci $\overline{g}, \overline{h}, \overline{tmp}$, itd.), które są odpowiednikami zmiennych rejestrowych języka C, co do których operator `&` nie ma zastosowania. Tylko dla tych zmiennych mamy gwarancję, że nie są one aliasami innych zmiennych czy też wyrażeń, co znacznie upraszcza proces dowodzenia wszelkich własności.

Istotnym elementem skonstruowanego języka jest model pamięci. W tej pracy rozważamy model pamięci, w którym na dowolnym adresie a można wykonywać operację *przesunięcia* o kolejne stałe całkowite ($a \oplus i, i \in Z$), dla uzyskania kolejnych, sąsiadujących adresów. Sąsiedzi a, to:

$$\ldots \quad , a \oplus -2, \quad a \oplus -1, \quad a \oplus 0 = a, \quad a \oplus 1, \quad a \oplus 2, \quad \ldots .$$

Zauważmy teraz, że w najprostszym przypadku, gdyby przyjąć, że pamięć zajmuje jeden ciągły obszar adresowy, prawdziwe musiałoby być stwierdzenie, że każde dwa adresy a_1, a_2 są odlegle jedynie o pewne przesunięcie $i \in Z$ (a_1 = $a_2 \oplus i$). To stwierdzenie nie zawsze jest jednak prawdziwe w rzeczywistości (poprzez zastosowanie pamięci wirtualnej, rejestrów segmentowych, itp.) a więc programy nie powinny z tego faktu korzystać bez potrzeby, chociażby ze względu na przenośność kodu programów. W opisywanym modelu pamięci moglibyśmy jednak wykazywać prawdziwość tego faktu. Również standardowe rozwiązanie problemu przydziału i zwolnienia pamięci polegające na wprowadzeniu pewnej puli adresów, które mogą zmieniać swój status (przydzielony/wolny) nie jest pozbawione błędów. Wyobraźmy sobie np. program, który przy całkowicie zajętej pamięci zwalnia jedną komórkę pamięci wskazywaną przez wskaźnik `p`. W takiej sytuacji, po ponownej alokacji jednej komórki pamięci na wskaźnik `q` (przy założeniu pełności systemu dowodowego), moglibyśmy w tym modelu pamięci udowodnić, że `p = q` (a więc, że `p` jest nadal sensownym wskaźnikiem). Tymczasem dalsze używanie przez program wskaźnika `p` jest błędem (gdy np. nasz program działa w środowisku wieloprogramowym). W naszym modelu pamięci unikniemy tych pułapek — jest to model w którym pamięć jest podzielona na segmenty, a operacja przydziału pamięci przekazuje zawsze świeży (nowy) segment.

Dodatkowo, obok przydziału i zwolnienia pamięci, umożliwiamy również przetestowanie stanu jej zajętości. Predykat $heap(n)$ wskazuje na ilość wolnej pamięci, tzn. $heap(n)$ jest prawdziwy ($heap(n) = \top$), gdy mamy co najmniej n komórek dostępnych. W przypadku, gdy nasza pamięć jest nieograniczona, formula $\forall\, n.\, heap(n)$ jest prawdziwa, w przeciwnym razie $\exists\, n.\, \neg heap(n)$. W tym ostatnim przypadku, każda operacja przydziału pamięci (`malloc`) zmniejsza liczbę dostępnych komórek, zaś operacja zwolnienia (`free`) — na odwrót. Operator $\triangle(w_a)$ bedzie definiował określoność termu adresowego w_a oraz fakt zaalokowania adresu, który jest wartością tego termu (czyli fakt, że operacja odwołania się do pamięci pod adresem w_a poprzez term $*w_a$ jest poprawna).

Spośród wielu pojęć, relacji i operatorów, które jeszcze wprowadzamy, na szczególną uwagę zasługują dwa. Po pierwsze wprowadzamy relację osiągalności ($\vdash\!\dashv$). Przykładowo, relacja:

$$\overline{d}{\downarrow} \vdash\!\dashv \tau$$

określa, że element τ występuje w drzewie $\overline{d}$, a dokładniej, że istnieje ścieżka adresów od $\overline{d}$ do τ. Po drugie umożliwiamy kwantyfikowanie po adresach występujących na ścieżkach adresów rozpoczynających się w pewnym konkretnym adresie. Na przykład, kwantyfikator ogólny postaci:

$$(\forall\, \overline{z} : \overline{d}{\downarrow}.\ \neg\overline{z}{\to}\texttt{wartość = } x))$$

oznacza, że w grafie o wierzchołku $\overline{d}$ nie występuje wartość x (dla każdego adresu $\overline{z}$, jeśli $\overline{d}{\downarrow} \vdash\!\dashv \overline{z}$, to $\neg\overline{z}{\to}\texttt{wartość = } x$). Dzięki tym dwu konstrukcjom język zaproponowanej przez nas logiki ma dużą moc wyrażania i jest bardzo zwięzły.

3.2. Specyfikacje struktur danych

Przedstawimy teraz zastosowanie logiki *C-AL* do definiowania struktur danych. Nie będziemy tu podawać przykładów aksjomatów lub reguł dowodowych, ponieważ system dowodowy składa się z około 100 reguł i aksjomatów, i nie miało by sensu prezentowanie ich w sposób fragmentaryczny.

3.2.1. Tablice

Choć tablice nie są wskaźnikowymi strukturami danych opisywanymi w logice *C-AL*, to jednak zauważmy, że poniższe deklaracje języka C są sobie równoważne:

```
T tablica[stała];
T* tablica; tablica=malloc(stała*sizeof(T));
```

Korzystając więc z tego drugiego sposobu możemy bez trudu posługiwać się tablicami w języku logiki *C-AL* (zakładamy w niej jedynie, że różne typy zajmują zawsze dokładnie jedną komórkę pamięci i nie potrzebujemy w związku z tym operatora `sizeof`). Oczywiście odwołania do tablicy postaci `tablica[i]` możemy interpretować korzystając z ich równoważnej postaci: `*(tablica + i*sizeof(T) )`, czyli w języku *C-AL* wyrażenia takie będą miały postać: $*(\texttt{tablica} \oplus \texttt{i})$.

3.2.2. Listy

Załóżmy teraz, że w sygnaturze języka mamy typ złożony (strukturę w języku C) o definicji: `struct Tlist { T wartość; Tlist *nast; }`.
W języku logiki zapiszemy to poprzez:

$$\texttt{Tlista} \equiv \texttt{T} \diamond \texttt{Tlista}*, \quad \texttt{Tlista.wartość} \stackrel{\text{df}}{=} 1, \quad \texttt{Tlista.nast} \stackrel{\text{df}}{=} 2,$$

oraz przykładowo: $\overline{p} \rightarrow \texttt{nast} = *(\overline{p} \oplus \texttt{Tlista.nast}) = *(\overline{p} \oplus 2)$.
Następujące formuły definiują postacie list zawierających i pozbawionych cykli:

$$\begin{aligned}
&\textit{lista-bez-cyklu}(l) \stackrel{\text{df}}{=} l = \textit{nil} \vee \exists\, \overline{p} : l{\downarrow}.\, \overline{p} \rightarrow \texttt{nast=nil},\\
&\textit{lista-cykliczna}(l) \stackrel{\text{df}}{=} \exists\, \overline{p} : l{\downarrow}.\, (\overline{p} \rightarrow \texttt{nast}){\downarrow} \longmapsto\!\!\!\dashv \overline{p},\\
&\textit{brak-cyklu}(l) \stackrel{\text{df}}{=} \forall\, \overline{p} : l{\downarrow}.\, \neg(\overline{p} \rightarrow \texttt{nast}){\downarrow} \longmapsto\!\!\!\dashv \overline{p}.
\end{aligned}$$

3.2.3. Struktury drzewiaste

Załóżmy deklarację następującej struktury:

`struct Tdrzewo { T wartość; Tdrzewo *lewe,*prawe; };`

W szeregu kolejnych, zwięzłych i prostych formuł definiujemy drzewa, drzewa BST oraz AVL:

$$\begin{aligned}
&\textit{drzewo}(d) \stackrel{\text{df}}{=} (\triangle(d) \vee d = \textit{nil}) \wedge\\
&\qquad \forall\, \overline{w} : d{\downarrow}.\, (\ \neg d{\downarrow} w \oplus \texttt{prawe} \longmapsto\!\!\!\dashv \overline{w} \rightarrow \texttt{prawe} \wedge\\
&\qquad\qquad \neg d{\downarrow} w \oplus \texttt{lewe} \longmapsto\!\!\!\dashv \overline{w} \rightarrow \texttt{lewe} \wedge\\
&\qquad\qquad (\triangle(\overline{w} \rightarrow \texttt{prawe}) \vee \overline{w} \rightarrow \texttt{prawe=nil}) \wedge\\
&\qquad\qquad (\triangle(\overline{w} \rightarrow \texttt{lewe}) \vee \overline{w} \rightarrow \texttt{lewe=nil})\ ).\\
&\textit{BST}(d) \stackrel{\text{df}}{=} \textit{drzewo}(d) \wedge\\
&\qquad \forall\, \overline{w} : d{\downarrow}.\\
&\qquad \forall\, \overline{z} : \overline{w}{\downarrow}.\, ((\overline{w} \rightarrow \texttt{lewe}{\downarrow} \longmapsto\!\!\!\dashv \overline{z} \Rightarrow \overline{z} \rightarrow \texttt{wartość} < \overline{w} \rightarrow \texttt{wartość}) \wedge\\
&\qquad\qquad (\overline{w} \rightarrow \texttt{prawe}{\downarrow} \longmapsto\!\!\!\dashv \overline{z} \Rightarrow \overline{z} \rightarrow \texttt{wartość} > \overline{w} \rightarrow \texttt{wartość})).\\
&\textit{wysokość-drzewa}(d, h) \stackrel{\text{df}}{=}\\
&\qquad (d = \texttt{nil} \wedge h = 0) \vee\\
&\qquad (d \rightarrow \texttt{lewy} = \texttt{nil} \wedge d \rightarrow \texttt{prawy} = \texttt{nil} \wedge h = 0) \vee\\
&\qquad ((\forall\, \overline{w} : d{\downarrow}.\, d{\downarrow} \stackrel{h}{\longmapsto\!\!\!\dashv} \overline{w}) \wedge (\exists\, \overline{w} : d{\downarrow}.\, \neg d{\downarrow} \stackrel{h-1}{\longmapsto\!\!\!\dashv} \overline{w}))\\
&\textit{AVL}(d) \stackrel{\text{df}}{=} \textit{BST}(d) \wedge\\
&\qquad \forall\, \overline{w} : d{\downarrow}.\, \forall\, h_1.\, \forall\, h_2.\\
&\qquad\quad (\ \textit{wysokość-drzewa}(\overline{w} \rightarrow \texttt{lewy}, h_1) \wedge\\
&\qquad\quad\ \ \textit{wysokość-drzewa}(\overline{w} \rightarrow \texttt{prawy}, h_2) \Rightarrow\\
&\qquad\qquad -1 \leq {+}h_1 - {+}h_2 \leq 1)
\end{aligned}$$

Rozważmy teraz problem wyszukiwania elementu x w drzewie BST. Odpowiedni program P zapisany w języku *C-AL* mógłby wyglądać następująco:

$$P \stackrel{\mathrm{df}}{=} \texttt{while(not } \overline{t}\texttt{=nil) and (not } \overline{t}\to\texttt{wartość = } x\texttt{) do}$$
$$\texttt{if } \overline{t}\to\texttt{wartość} < x \texttt{ then } \overline{t}\texttt{:=}\overline{t}\to\texttt{prawy}$$
$$\texttt{else } \overline{t}\texttt{:=}\overline{t}\to\texttt{lewy fi;}$$

Własność poprawności programu P możemy teraz sformalizować formułami:

$$(BST(\overline{d}) \wedge \overline{t}\texttt{=}\overline{d} \wedge \forall\, \overline{z} : \overline{d}{\downarrow}\texttt{.}\, \neg\overline{z}\to\texttt{wartość = } x) \Rightarrow P(\overline{t}\texttt{=nil}),$$
$$(BST(\overline{d}) \wedge \overline{t}\texttt{=}\overline{d} \wedge \overline{z}\to\texttt{wartość = } x \wedge \overline{d}{\downarrow}\longmapsto\!\!\dashv \overline{z}) \Rightarrow P(\overline{t}\texttt{=}\overline{z}),$$
$$(\overline{z}\to\texttt{wartość = } c) \Rightarrow P(\overline{z}\to\texttt{wartość = } c).$$

Sytem dowodowy opisywanej tu logiki jest poprawny i pełny (por. [2]). Powyższe własności możemy więc wykazać dowodem formalnym.

4. Podsumowanie

W niniejszej pracy zaprezentowano po krótce dwa podejścia do specyfikowania wskaźnikowych struktur danych. Pierwsze z nich jest oparte o logikę Hoare'a. Na podejście to składa się propozycja języka asercji, w którym wykorzystano konstrukcje temporalne, oraz wariant logiki Hoare'a dostosowany do weryfikowania asercji wyrażonych w proponowanym języku. Logika ta jest poprawna i pełna w sensie Cook'a. Drugie podejście stanowi rozszerzenie języka logiki algorytmicznej o wskaźniki, struktury, instrukcje przydziału i zwolnienia pamięci. System dedukcyjny dla tego języka jest poprawny i pełny.

Oba podejscia stanowią krok w kierunku formalnej weryfikacji programów operujących na stosowanych strukturach wskaźnikowych. Niestety dowody poprawności programów, choć możliwe do wyknonania są wielokrotnie bardziej skomplikowane od weryfikowanych programów. Dalsze prace w tym kierunku będą się koncentrowały na usprawnianiu procesu weryfikacji — zarówno przez dalsze prace nad językiem specyfikacji, jak i nad możliwościami zastosowania półałtomatycznych systemów dowodzenia.

Bibliografia

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

[2] Adam Bałaban. *Wnioskowanie o programach za pomocą rozszerzenia logiki algorytmicznej*. PhD thesis, Uniwersytet Warszawski, Wydział Matematyki, Informatyki i Mechaniki. W przygotowaniu.

[3] Lech Banachowski, Krzysztof Diks, and Wojciech Rytter. *Algorytmy i struktury danych*. Wydawnictwa Naukowo-Techniczne, 1996.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Wprowadzenie do algorytmów*. Wydawnictwa Naukowo-Techniczne, 1997.

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

[6] Patrick Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 15, pages 841–993. Elsevier, 1990.

[7] Edsger W. Dijkstra. *Umiejętność programowania*. Wydawnictwa Naukowo-Techniczne, 1985.

[8] David Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, volume 165 of *Synthese Library*, chapter II.10, pages 497–604. D. Reidel Publ. Co., Dordrecht, 1984.

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

[10] Dexter Kozen and J. Tiuryn. Logics of programs. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 14, pages 789–840. Elsevier, New York, N.Y., 1990.

[11] Marcin Kubica. *Formalna specyfikacja wskaźnikowych struktur danych*. PhD thesis, Wydział Matematyki, Informatyki i Mechaniki, Uniwersytet Warszawski, 1999.

[12] Grażyna Mirkowska and Andrzej Salwicki. *Algorithmic Logic*. D. Reidel Publishing Company, 1987.

[13] Grażyna Mirkowska and Andrzej Salwicki. *Logika algorytmiczna dla programistów*. Wydawnictwa Naukowo-Techniczne, 1992.

[14] Greg Nelson. A generalization of dijkstra's calculus. Technical Report 16, Digital Equipment Corporation, Systems Research Centre, 2 April 1987.

[15] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, volume 3, chapter 7. Springer-Verlag, 1997.

[16] Niklaus Wirth. *Algorytmy + struktury danych = programy*. Wydawnictwa Naukowo-Techniczne, trzecie edition, 1999.