Denotacyjna inżynieria języków programowania

Prezentacja do pobrania na
http://www.moznainaczej.com.pl/inzynieria-denotacyjna

Andrzej Blikle
przy współpracy z Piotrem Chrząstowskim-Wachtlem

10 grudnia 2018

Dlaczego się na to porwałem?

Stan rzeczy w przemyśle IT

Cytat z umowy licencyjnej *Producent oprogramowania zwraca uwagę, że w oparciu o bieżący stan techniki, nie jest możliwe wyprodukowanie programu komputerowego w taki sposób, aby bezbłędnie pracował we wszystkich możliwych konfiguracjach. Producent gwarantuje w oparciu o dotychczasowych użytkowników, że do dnia zawarcia niniejszej umowy nie zna żadnych błędów w przekazywanym programowaniu.*

Stan rzeczy w nauce

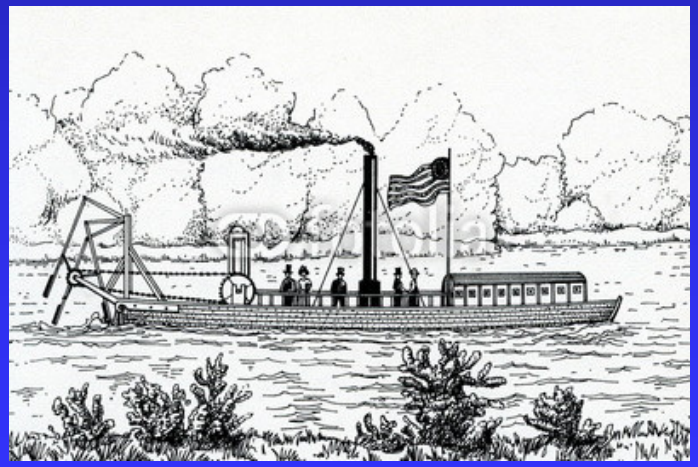
The KeY Book; From Theory to Practice (Springer 2016)
Przez wiele lat pojęcie formalnej weryfikacji było niemalże synonimem weryfikacji funkcjonalnej. W ostatnich latach staje się coraz bardziej jasne, że pełna funkcjonalna weryfikacja programu jest dla prawie wszystkich zastosowań celem iluzorycznym. (…) Prawdziwym wąskim gardłem weryfikacji funkcjonalnej jest nie weryfikacja, ale specyfikacja.

Dlaczego to się nie udawało?

Żeby - zbudować logikę programów
trzeba - semantykę języka opisać matematycznie.

Dwa historyczne podejścia do matematycznych semantyk języków programowania:

Semantyki operacyjne (VDL)
opisać wirtualną maszynę



Semantyki denotacyjne (VDM)
S : Język → Denotacje
S(P♦Q) = S(P) • S(Q)

S : AlgSkładni → AlgDenotacji

SEMANTYKA
homomorfizm
algebr wielorodzajowych

Czy semantyka denotacyjna zawsze da się napisać?

Tradycyjne podejście do budowania semantyki denotacyjnej

Najpierw składnia:
jak będziemy wyrażać treści

Później denotacje:
jakie treści będziemy wyrażać

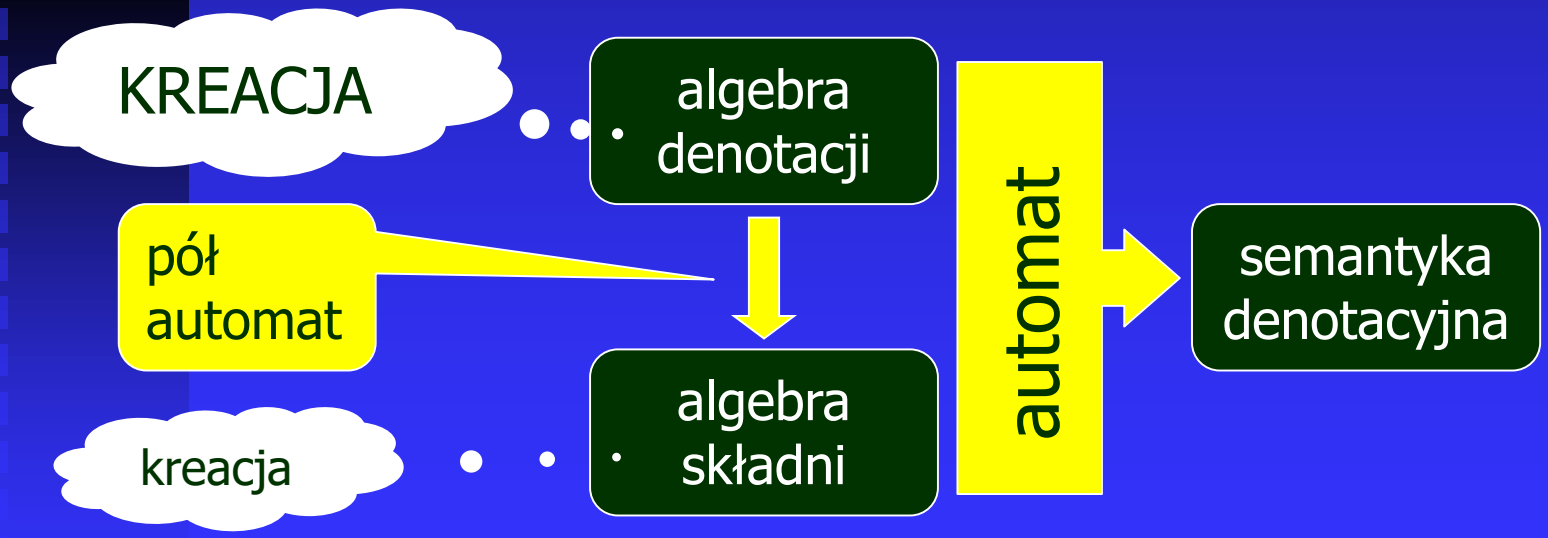
Ta kolejność wynikała z przyczyn historycznych: gdy zaczęto myśleć o semantyce, języki już istniały.

Moja hipoteza
Dla znanych mi języków programowania nie da się zbudować (praktycznych) semantyk denotacyjnych.

Odwróćmy kolejność myślenia i działania

Najpierw opiszmy w metajęzyku świat, który mają reprezentować składowe języka programowania: algebrę denotacji obejmującą programy, instrukcje, deklaracje, wyrażenia, typy, obiekty,...

Później zbudujmy formalną składnię dla języka programowania opisującego ten świat: algebrę składni



Gdy mamy już język z semantyką denotacyjną, można pomyśleć o dowodzeniu własności programów.

Czy jednak dowodzenie własności programów to dobra droga?

Dwa problemy:

1. Dowód twierdzenia jest zwykle dłuższy od twierdzenia.
2. Programy zwykle nie są poprawne.

Ponownie odwróćmy kolejność myślenia i działania

Matematyk
Najpierw twierdzenie (hipoteza) później dowód.

Inżynier (informatyk)
Najpierw projekt (dowód) później produkt, np. most.

Reguły logiki programów ustępują regułom budowania programów poprawnych

Seria języków **Lingua**

Lingua-A - aplikatywny fundament serii (wyrażenia i typy)
Lingua-1 - Lingua-A + instrukcje i definicje typów (bez procedur)
Lingua-2 - Lingua-1 + procedury z multirekursją
Lingua-3 - Lingua-2 + programowanie obiektowe
Lingua-SQL - Lingua-3 + SQL-owe bazy danych (API)

Istituto di elaborazione
dell'informazione del CNR
Pisa 1969

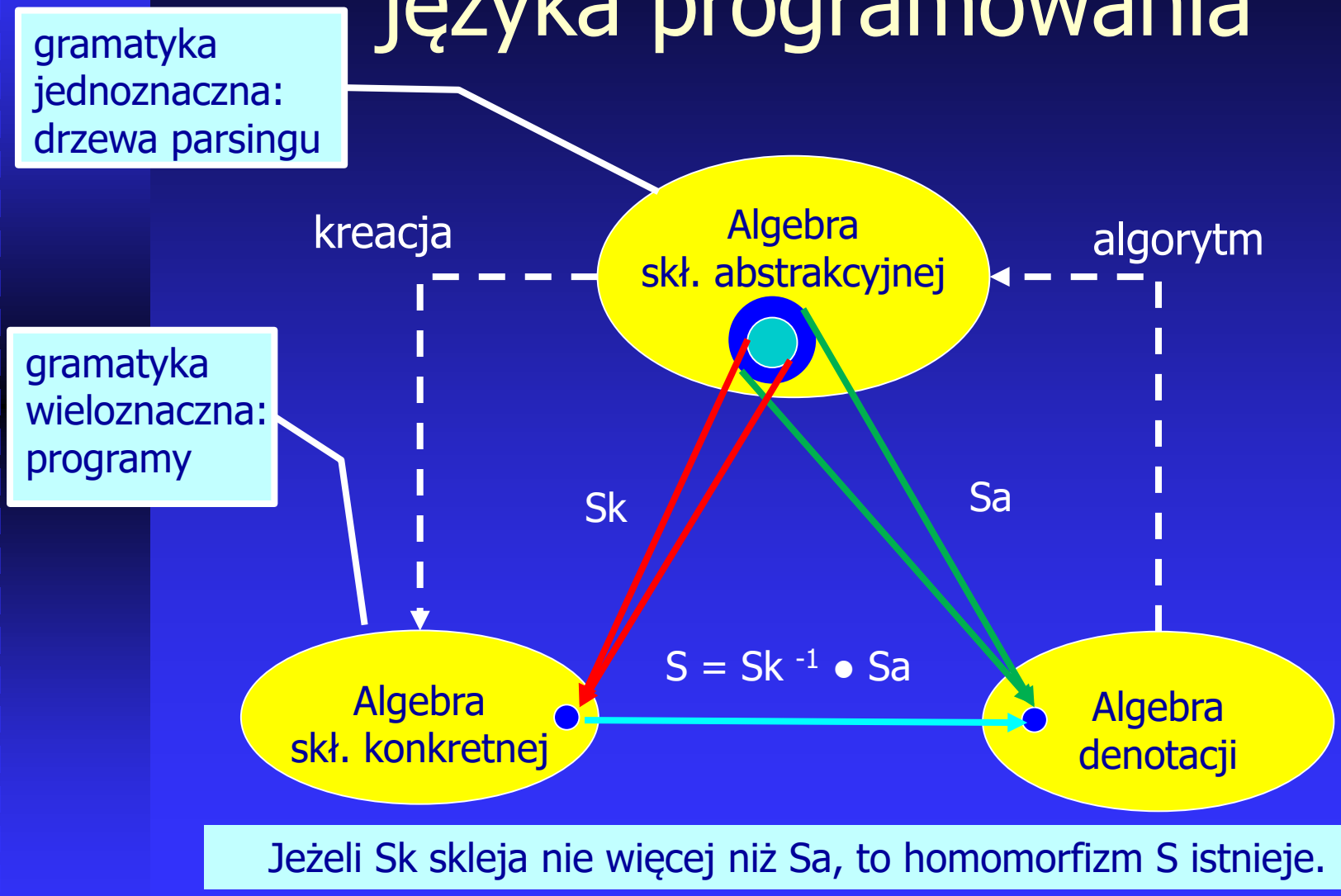
Algorithmically definable functions. A contribution towards the semantics of programming languages,
Dissertationes Mathematicae, LXXXV,
PWN, Warszawa 1971

Język Lingua nie jest propozycją standardu, a jedynie przykładem służącym do ilustrowania metody:

- od denotacji do składni,
- budowanie programów poprawnych zamiast dowodzenia poprawności

Narzędzia matematyczne budowniczego modeli denotacyjnych

Denotacyjny model języka programowania



Jeżeli Sk skleja nie więcej niż Sa, to homomorfizm S istnieje.

Algebra denotacji

Nośniki

Ide = {x, y, z,…}

DenWyr = Stan → Liczba

DenIns = Stan → Stan

Stan = Ide ⟹ Liczba

Konstruktory

zmienna : Ide ↦ DenWyr

plus : DenWyr x DenWyr ↦ DenWyr

razy : DenWyr x DenWyr ↦ DenWyr

przypisz : Ide x DenWyr ↦ DenIns

sekwencja : DenIns x DenIns ↦ DenIns

Bardzo uproszczony przykład cz. 1

Oznaczenia:

A → B; f. częściowe

A ↦ B; f. całkowite

A ⟹ B; f. skończone

AUTOMAT

Algebra (gramatyka) składni abstrakcyjnej

Ide = {x, y, z,…}

Wyr = zm(Ide) | plus(Wyr, Wyr) | razy(Wyr, Wyr)

Ins = przypisz(Ide, Wyr) | sekwencja(Ins, Ins)

AUTOMAT

Semantyka składni abstrakcyjnej (Sa)

Sid : Ide ↦ Ide (identycznościowo)

Swy : Wyr ↦ DenWyr

Sin : Ins ↦ DenIns

Bardzo uproszczony przykład cz. 2

Algebra (gramatyka) składni abstrakcyjnej

Ide = {x, y, z}
Wyr = zm(Ide) | plus(Wyr, Wyr) | razy(Wyr, Wyr)
Ins = przypisz(Ide, Wyr) | sekwencja(Ins, Ins)

KREACJA
wspomagana

Algebra (gramatyka) składni konkretnej

Ide = {x, y, z}
Wyr = Ide | (Wyr + Wyr) | (Wyr * Wyr)
Ins = Ide := Wyr | Ins ; Ins

dopuszczalna wieloznaczność

Algebra (gramatyka) składni kolokwialnej

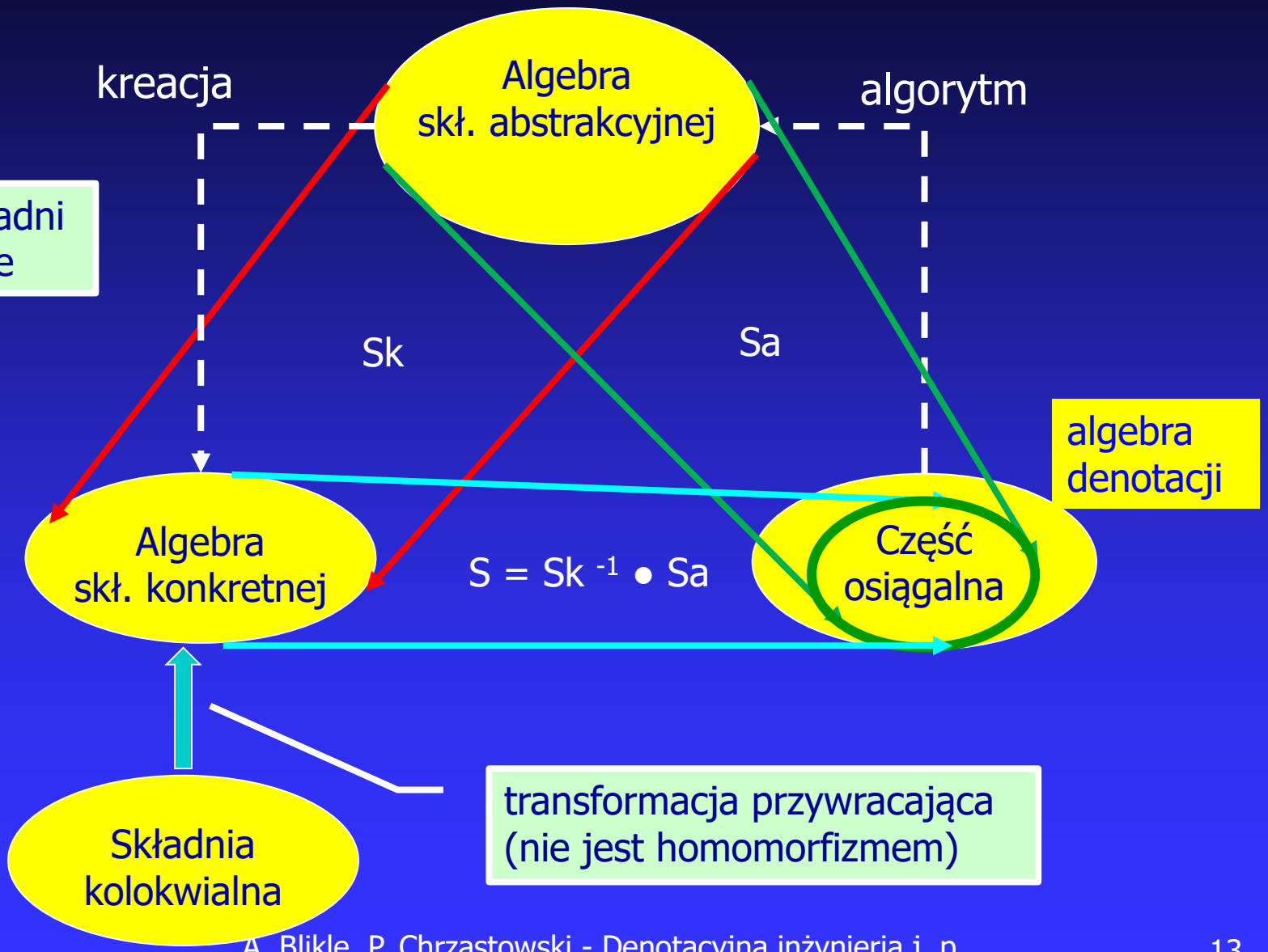
Ide = {x, y, z}
Wyr = Ide | (Wyr + Wyr) | (Wyr * Wyr)
Wyr + Wyr | Wyr * Wyr
Ins = Ide := Wyr | Ins ; Ins

KREACJA
wspomagana

Alg. skł. kolokw. nie jest podobna do alg. denot.
Nie istnieje dla niej semantyka denotacyjna!

niedopuszczalna wieloznaczność

Model ze składnią kolokwialną



Poprawność programów w języku algebry relacji

Stan – zbiór stanów
A, B ⊂ Stan – własności stanów (predykaty)
F : Stan → Stan – funkcja częściowa (program)
A•F = {b | (∃ a : A) F(a) = b} – obraz zbioru A skr. AF
F•B = {a | (∃ b : B) F(a) = b} – przeciwobraz zbioru B skr. FB

AF ⊂ B – częściowa poprawność F względem pre. A i post. B
A ⊂ FB – całkowita poprawność F względem pre. A i post. B

Każde obliczenie zaczynające się w A, jeżeli się zakończy, to zakończy się w B (ale może się nie zakończyć);
F jest na A funkcją częściową

Każde obliczenie zaczynające się w A zakończy się w B;
F jest na A funkcją całkowitą

Budowanie metaprogramów poprawnych

metaprogram – program zawierający w sobie (tekstowo) pre- i post-warunki oraz asercje (niezmienniki)

mp. poprawny – spełniający zawartą w nim specyfikację

notacja: pre(A) F post(B) oznacza $A \subset FB$

Przykłady reguł budowania metaprogramów poprawnych

pre(A) F post(B)
pre(C) G post(D)
$B \subset C$

---

pre(A) F;G post(D)

pre($A \cap C$) F post(B)
pre($A \cap \neg C$) G post(B)
$A \subset C \mid \neg C,\ C \cap \neg C = \emptyset$

---

pre(A) if C then F else G fi post(B)

W logice klasycznej (i logice Hoare'a) te warunki są tautologiami (są zawsze spełnione). W Lingua mamy 3-wart rach. predykatów.

Błędy abstrakcyjne

Przykłady sytuacji, w których zostaje przerwane wykonanie programu:

- próba dzielenie przez zero,
- próba wyliczenia a[i], gdy i jest spoza dziedziny tablicy a,
- próba wyliczenia x+y, gdy suma przekracza dopuszczalny rozmiar,
- próba wyliczenia x+y, gdy x nie zostało zadeklarowane,
- etc.

Dana = ... - dziedzina danych
DanaB = Dana | Błąd

op : $Dana_1$ x ... x $Dana_n$ → Dana
opb : $DanaB_1$ x ... x $DanaB_n$ → DanaB

wyświetlenie błędu i wstrzymanie obliczenia

jeżeli (∃ i) (d_i : Błąd) to op.(d_1,...,d_n) : Błąd - transparentność w.b.

Błędy abstrakcyjne nie obejmują sytuacji niekończących się obliczeń!

Błędy abstrakcyjne mogą inicjować akcję interwencyjną

Trójwartościowy rachunek predykatów

```
if x > 0 oraz 1/x < 10
   then y := x + y
   else  y := x – y
fi
```

Logika programów pozostaje nadal dwuwartościowa (klasyczna).

Jak zachowa się ten program dla x = 0?

Aby w semantyce programów opisać zachowanie programów w takich sytuacjach oraz dowodzić ich własności trzeba wprowadzić **trójwartościowy (wielowartościowy) rachunek predykatów**.

bb – błąd lub niekończące się obliczenie

lub	**pp**	**ff**	**bb**
pp	pp	pp	pp
ff	pp	ff	bb
bb	bb	bb	bb

oraz	**pp**	**ff**	**bb**
pp	pp	ff	bb
ff	ff	ff	ff
bb	bb	bb	bb

nie	**pp**
pp	ff
ff	pp
bb	bb

Leniwa ewaluacja wg. McCarthy' ego: a lub b ≠ b lub a

Lingua
eksperyment z budowaniem denotacyjnego modelu języka programowania

Lingua nie jest propozycją praktycznego języka programowania, a jedynie przykładem służącym do ilustrowania ekspresywności modeli denotacyjnych.

Lingua-A – algebra danych

Nośniki algebry danych:

ide : Identyfikator = …
dan : Dana = Boolowska | Liczba | Słowo | Lista | Tablica | Rekord

gdzie

boo : Boolowska = {pp, ff}
lic : Liczba = …
sło : Słowo = …
lis : Lista = Dana^{c*}
tab : Tablica = Liczba ⟹ Dana - funkcje skończone
rek : Rekord = Identyfikator ⟹ Dana - funkcje skończone

Tu jeszcze nie ma błędów abstrakcyjnych

tu są też rekordy

Konstruktor algebry danych (przykład):

wymień-w-rek : Rekord x Identyfikator x Dana ⟼ Rekord

wymień-w-re.(rek, ide, dan) =
  rek.ide = ? ➔ ?
  prawda ➔ rek[ide/dan]

Notacja wg. MetaSoft oparta na VDM (IPI PAN 1980-89)

Lingua-A – algebra korpusów

kor : Korpus =
{('boolowska')} | {('liczba')} | {('słowo')} | - korpusy proste
{'L'} x Korpus | - korpusy list
{'T'} x Korpus | - korpusy tablic
{'R'} x (Identyfikator ⟹ Korpus) - korpusy rekordów

korpus danej opisuje jej strukturę

Nośniki algebry korpusów
ide : Identyfikator
kor : KorpusB = Korpus | Błąd

Błąd – sk. zbiór słów (komunikatów błędów)

Klany korpusów
KLAN-Kr : Korpus ⟼ Pod.Dana
KLAN-Kr.('T', kor) = Liczba ⟹ KLAN-Kr.kor

podzbiory zbioru Dana

Nie wszystkie dane mają korpusy, ale mają je wszystkie dane osiągalne. Np. nie mają korpusu listy elementów o różnych korpusach.

KOR : Dana → Korpus - określona dla danych mających korpusy
jeżeli dan : KLAN-Kr.kor to KOR.dan = kor

Lingua-A – algebra korpusów cd.

Konstruktor algebry korpusów (przykład):

Kr[wymień-w-re].(kor-r, ide, kor-d) =
  kor-i : Błąd → kor-i dla i = r, d
  rodzaj.kor-r ≠ ‘R’ → ‘oczekiwany-rekord’
  niech
    (‘R’, rko) = kor-r
  rko.ide = ? → ‘atrybut-nie-występuje-w-rekordzie’
  kor-d ≠ rko.ide → ‘niezgodne-korpusy’
  prawda → kor-r[ide/kor-d]

Każdej operacji na danych ope zostaje przypisana transparentna wzg. błędów i adekwatna dla niej operacja na korpusach Kr[ope]

KOR.(ope.(arg-1,…,arg-n)) = Kr[ope].(KOR.arg-1,…,KOR.arg-n)
(gdy wszystko określone i bez błędów; KOR – „częściowy homomorfizm”

Lingua-A – algebra kompozytów

kom : Kompozyt ⊂ Dana x Korpus
kom : Kompozyt = {(dan, kor} | dan : KLAN-Kr.kor}

wartości przyszłych wyrażeń będą kompozytami

Nośniki algebry kompozytów:
ide : Identyfikator = …
kom : KompozytB = Kompozyt | Błąd

Konstruktory algebry kompozytów (poza boolowskimi)
Km[ope].(dan, kor) = (ope.dan, Kr[ope].kor) - w dużym uproszczeniu

Wszystkie poza boolowskimi silnie transparentne wzg. błędów

Konstruktory boolowskie z leniwą ewaluacją:

oraz-K.(kom-1, kom-2) =

kom-1 : Błąd	➔ kom-1
rodzaj.kom-1 ≠ (‘boolowska’)	➔ ‘oczekiwana-boolowska’
dana.kom-1 = ff	➔ (ff, (‘boolowska’))
kom-2 : Błąd	➔ kom-2
rodzaj.kom-2 ≠ (‘boolowska’)	➔ ‘oczekiwana-boolowska’
prawda	➔ (dana.kom-2, (‘boolowska’))

Lingua-A – jarzma

Korpusy określają strukturę danych.
Jarzma opisują inne właściwości danych, np.

all-list (**row**.pensja + **row**.premia < 7000) - własność listy rekordów

tra : Transfer ⊂ KompozytB ⟼ KompozytB
tra.błd = błd dla błd : Błąd - transparentność w.b.

jar : Jarzmo ⊂ KompozytB ⟼ KompozytBooB
jar.błd = błd dla błd : Błąd - transparentność w.b.

Nośniki algebry transferów
ide : Identyfikator = ...
tra : Transfer = ...

transfer boolowski

KLAN-Tr : Transfer ⟼ Pod.Kompozyt
KLAN-Tr.tra = {kom | tra.kom = (pp, ('boolowska'))}

Lingua-A – typy

typ : Typ = Korpus x Transfer

KLAN-Ty : Typ ⟼ Pod.Kompozyt
KLAN-Ty.(kor, tra) =
  {(dan, kor) | dan : KLAN-Kr.kor **oraz** (dan, kor) : KLAN-Tr.tra}

Nośniki algebry typów

ide : Identyfikator = ...
tra : Transfer = ...
typ : TypB = Typ | Błąd

typy są składowalne w pamięci dla ich wielokrotnego wykorzystania oraz dla budowania ich wstępująco

Konstruktory algebry typów

1. wszystkie konstruktory identyfikatorów,
2. wszystkie konstruktory algebry transferów,
3. specyficzne konstruktory typów.

Lingua-A – typy cd.

Dołożenie nowego atrybutu do typu rekordowego (przykład)

Ky[dołóż-do-re] : TypB x Identyfikator x TypB ⟼ TypB

Ky[dołóż-do-re].(typ-r, ide, typ-d) =
  typ-i : Błąd ➔ typ-i dla i = r, d
  niech
    (kor-i, tra-i) = typ-i dla i = r, d
    nowy-kor = Kr[dołóż-do-re].(kor-r, ide, kor-d)
    nowy-tra = oraz-T.(tra-r, Kt[weź-z-re].ide • tra-i)
  nowy-kor : Błąd ➔ nowy-kor
  prawda ➔ (nowy-kor, nowy-tra)

Lingua-A – stany

dut : DanUty = (Dana | {Ω}) x Typ - dana utypowiona
war : Wartość = {(dan, typ) | dan = Ω lub dan : KLAN-Ty.typ}

sta : Stan = Środ x Skład - stan
śro : Środ = ŚroTyp x ŚroPro - środowisko
skł : Skład = Wart x (Błąd | {'OK'}) - skład
wrt : Wart = Identyfikator ⟹ Wartość - wartościowanie
srt : ŚroTyp = Identyfikator ⟹ Typ - środowisko typów
srp : ŚroPro = Identyfikator ⟹ Procedura | Funkcja - środowisko proc.

Typy są składowalne w środowisku, ale korpusy i transfery nie są. To decyzja inżynierska, a nie matematyczna konieczność.

Lingua-A
algebra denotacji wyrażeń danologicznych

dwd : DenWyrDan = Stan → Kompozyt | Błąd - f. częściowe!
dwd.(śro, (wrt, błd)) = błd gdy błd ≠ OK - transparentność w. b.

Nośniki algebry denotacji wyrażeń danologicznych
Identyfikator
DenWyrDan

Konstruktory algebry AlgDenWyrDan
1. konstruktory identyfikatorów,
2. konstruktor zmiennych
3. konstruktory pochodne konstruktorom kompozytów,
4. konstruktor denotacji wyrażenia warunkowego.

Lingua-A; AlgDenWyrDan cd.

zmienna-dan : Identyfikator ⟼ DenWyrDan

zmienna-dan.ide.sta =
  jest-błąd.sta ➔ błąd.sta
  niech
    (śro, (wrt, 'OK')) = sta
  wrt.ide = ? ➔ 'zmienna-niezadeklarowana'
  niech
    ((dan, kor), jar) = wrt.ide
  dan = Ω ➔ 'zmienna niezainicjalizowana'
  prawda ➔ (dan, kor)

Kdd[Km[ope]].(arg-1,…,arg-n).sta =
  analiza błędów i określoności
  niech
    kom-i = arg-i.sta
  prawda ➔ Km[ope].(kom-1,…,kom-n)

uproszczony
schemat definicji
konstruktora denotacji

Lingua-A

algebra denotacji wyrażeń typologicznych i transferowych

Nośniki algebry

ide : Identyfikator = …
tra : Transfer = …
dwt : DenWyrTyp = Stan ⟼ Typ | Błąd

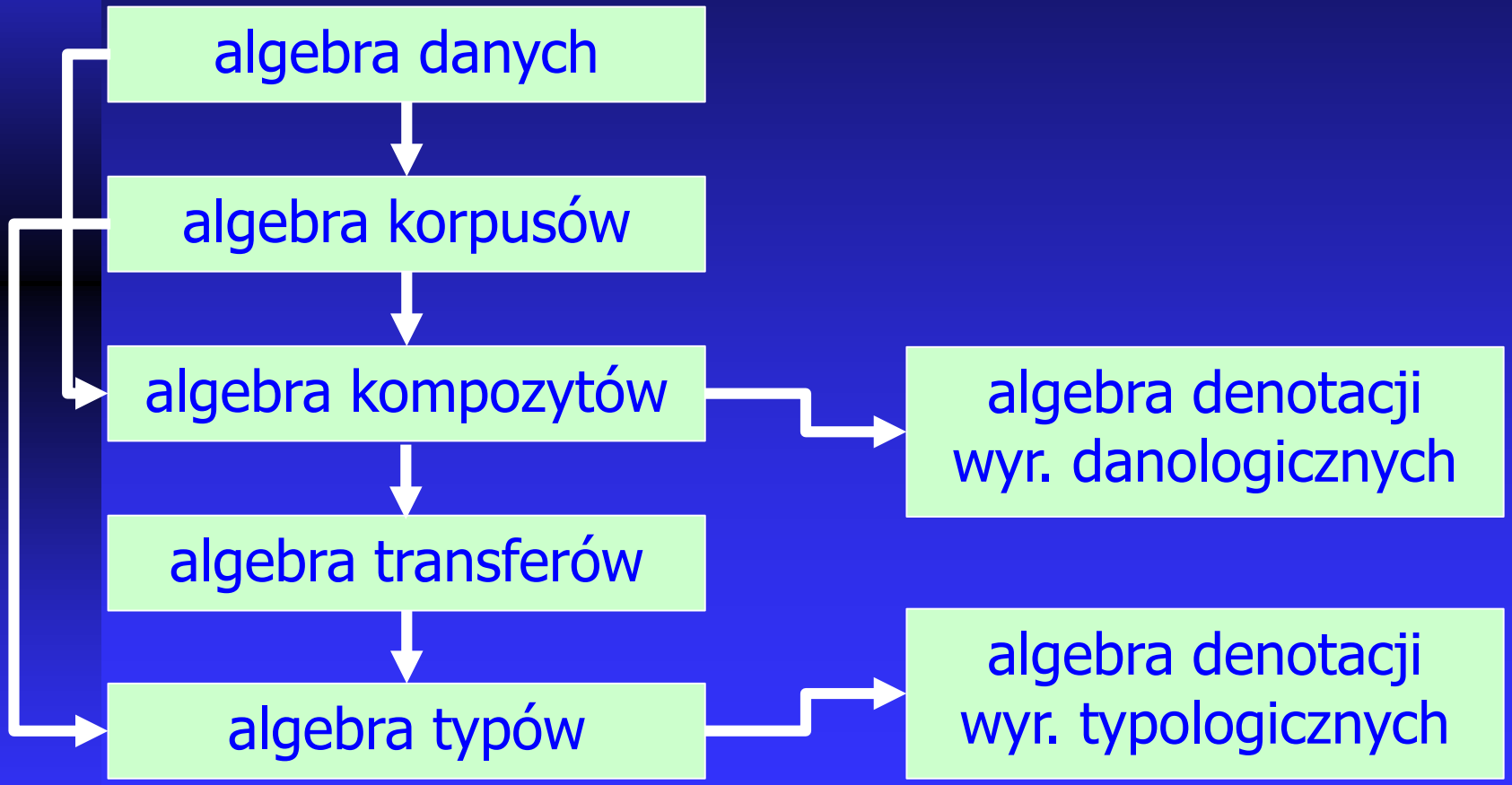
Konstruktory algebry

1. znane konstruktory identyfikatorów
2. znane konstruktory transferów
3. specyficzne konstruktory denotacji wyrażeń typologicznych

Kdt[Ky[twórz-li]] : DenWyrTyp ⟼ DenWyrTyp

Kdt[Ky[twórz-li]].dwt.sta = - wyliczanie typu listowego
analiza błędów
prawda ➔ Ky[twórz-li].(dwt.sta)

Podsumowanie
Siedem kroków do algebry denotacji wyrażeń



Lingua-1

wyrażenia, instrukcje, deklaracje zmiennych i definicje typów

Nośniki algebry denotacji

ide	: Identyfikator	= …	
dwd	: DenWyrDan	= Stan → KompozytB	- den. wyrażeń danologicznych
dwtr	: DenWyrTra	= Transfer	- den. wyrażeń transferowych
dwt	: DenWyrTyp	= Stan ↦ TypB	- den. wyrażeń typologicznych
ddz	: DenDekZmi	= Stan ↦ Stan	- den. deklaracji zmiennych
ddt	: DenDefTyp	= Stan ↦ Stan	- den. definicji stałych typ.
din	: DenIns	= Stan → Stan	- denotacje instrukcji
dep	: DenPre	= Stan → Stan	- denotacje preambuł
dpr	: DenPro	= Stan → Stan	- denotacje programów

Lingua-1
deklaracje i definicje

deklaruj-zmi-dan : Identyfikator x DenWyrTyp ⟼ DenDekZmi

deklaruj-zmi-dan.(ide, dwt).(śro, (wrt, błd)) =
  analiza błędów
  niech
    typ = dwt. (śro, (wrt, błd))
  prawda ➔ (śro, (wrt[ide/(Ω, typ)], 'OK'))

deklaracja
zmiennej danologicznej

definiuj-sta-typ : Identyfikator x DenWyrTyp ⟼ DenDefTyp

definiuj-sta-typ.(ide, dwt). ((stp, srp), skł) =
  analiza błędów
  niech
    typ = dwt. ((stp, srp), skł)
  prawda ➔ ((srt[ide/typ], srp), skł)

definicja
stałej typologicznej

Lingua-1
przypisania i wymiany transferów

przypisz : Identyfikator x DenWyrDan ⟼ DenIns

przypisz.(ide, dwd).sta =
  analiza błędów i spełnienia jarzma
  niech
    (kom, tra) = sta.ide
    (kom-n, tra) = war-n
  prawda ➔ ((srt, srp), wrt[ide/war-n], 'OK')

przypisanie
wartości do zmiennej

wymień-tr : Identyfikator x DenWyrTra ⟼ DenIns
wymienia transfer bez zmiany kompozytu

wymiana
transfery przy zmiennej

Lingua-1
pętla while i inne konstruktory

dopóki : DenWyrDan x DenIns ⟼ DenIns

pętla while-do-od

dopóki.(dwd, din).sta =
  jest-błąd.sta ➔ sta
  dwd.sta = ? ➔ ?
  dwd.sta : Błąd ➔ sta ◄ dwd.sta
  niech
    (dan, kor) = dwd.sta
  rodzaj.kor ≠ (‘boolowska’) ➔ sta ◄ ‘oczekiwana-boolowska’
  dan = ff ➔ sta
  prawda ➔ (din • [dopóki.(dwd, din)]).sta

Pozostałe konstruktory:
- złożenie sekwencyjne instrukcji, deklaracji zm. i definicji typów
- złożenie instrukcji if-then-else
- tworzenie preambuł i programów

Lingua-1

składnia konkretna instrukcji, preambuł i programów

```
ins : Instrukcja =
   Identyfikator := WyrDan                                  |
   yoke(Identyfikator):= WyrTra                             |
   if WyrDan then Instrukcja else Instrukcja fi             |
   while WyrDan do Instrukcja od                            |
   Instrukcja ; Instrukcja

pab : Preambuła =
   DefTyp                        |
   DekZmi                        |
   Preambuła ; Preambuła

prg : Program =
   begin-program Instrukcja end-program                     |
   begin-program Preambuła ; Instrukcja end-program
```

Lingua-1, cd.

przykład składni kolokwialnej dla dek. zmiennych

let x, y, z **be** number **tel**

rozwija się do

let x **be** number **tel;**
let y **be** number **tel**;
let z **be** number **tel**

Lingua-2, procedury

pfo : ParFor = (Identyfikator x DenWyrTyp)^{c*} - parametry formalne dek. p.
pak : ParAkt = Identyfikator^{c*} - parametry aktualne wyw. p.

pri : ProImp = ParAkt x ParAkt ⟼ Skład → Skład - proc. imperatywne
prf : ProFun = ParAkt x ParAkt ⟼ Skład → KompozytB - proc. fun.
pro : Procedura = ProImp | ProFun - procedury

ddp : DenDekPri = Stan ⟼ Stan - denotacje dek. proc. imp.
ddf : DenDekPrf = Stan ⟼ Stan - denotacje dek. proc. fun.

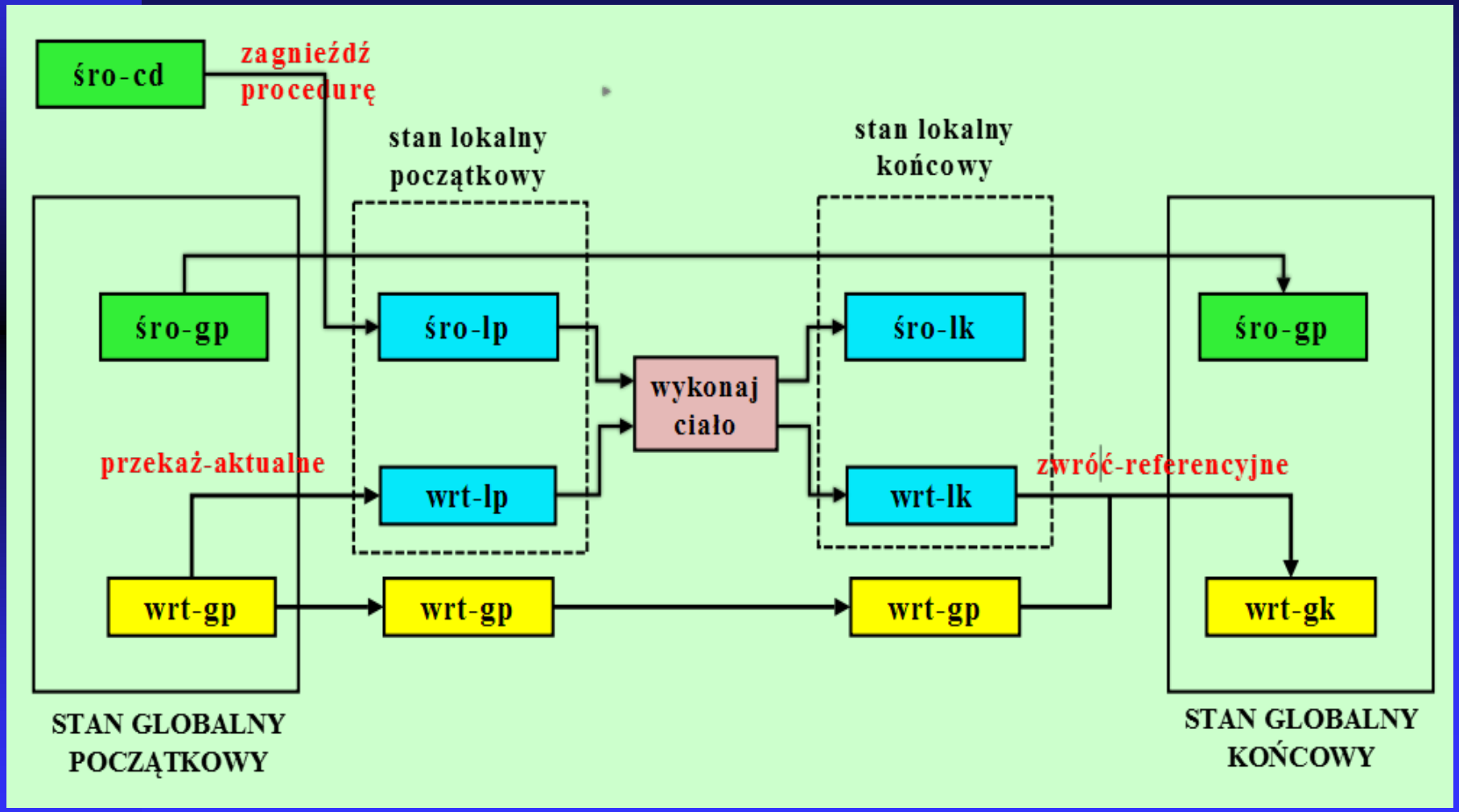
Gdyby procedury były transformacjami stanów, to układ równań

ProImp = ParAkt x ParAkt ⟼ Stan → Stan (operator → jest nieciągły)
Stan = (ŚroTyp x ŚroPro) x Skład
ŚroPro = Identyfikator ⟹ ProImp | ProFun

nie miałby rozwiązania.

A.Blikle, A.Tarlecki, *Naive denotational semantics,* IFIP 1983

Lingua-2
wywołanie procedury imperatywnej



Lingua-2
procedury jako parametry procedur

$$\text{Procedura} = \text{Parametr} \mapsto \text{Stos} \to \text{Stos}$$
$$\text{Parametr} = \text{Wartość} \mid \text{Procedura}$$

to równanie nie ma rozwiązania
bo operator $\mapsto$ nie jest ciągły

$$\text{Procedura.0} = \text{Parametr.0} \mapsto \text{Stan} \to \text{Stan}$$
$$\text{Parametr.0} = \text{Wartość}$$

Dla $n > 0$:

$$\text{Procedura.n} = \text{Parametr.n} \mapsto \text{Stan} \to \text{Stan}$$
$$\text{Parametr.n} = \text{Parametr.0} \mid \ldots \mid \text{Parametr.(n-1)}$$

Procedura nie może przyjąć samej siebie jako parametr (jak w Algolu 60).

Lingua-3
programowanie obiektowe

obi : Obiekt = Środ ⟼ Środ
bio : BibObi = Identyfikator ⟹ Obiekt

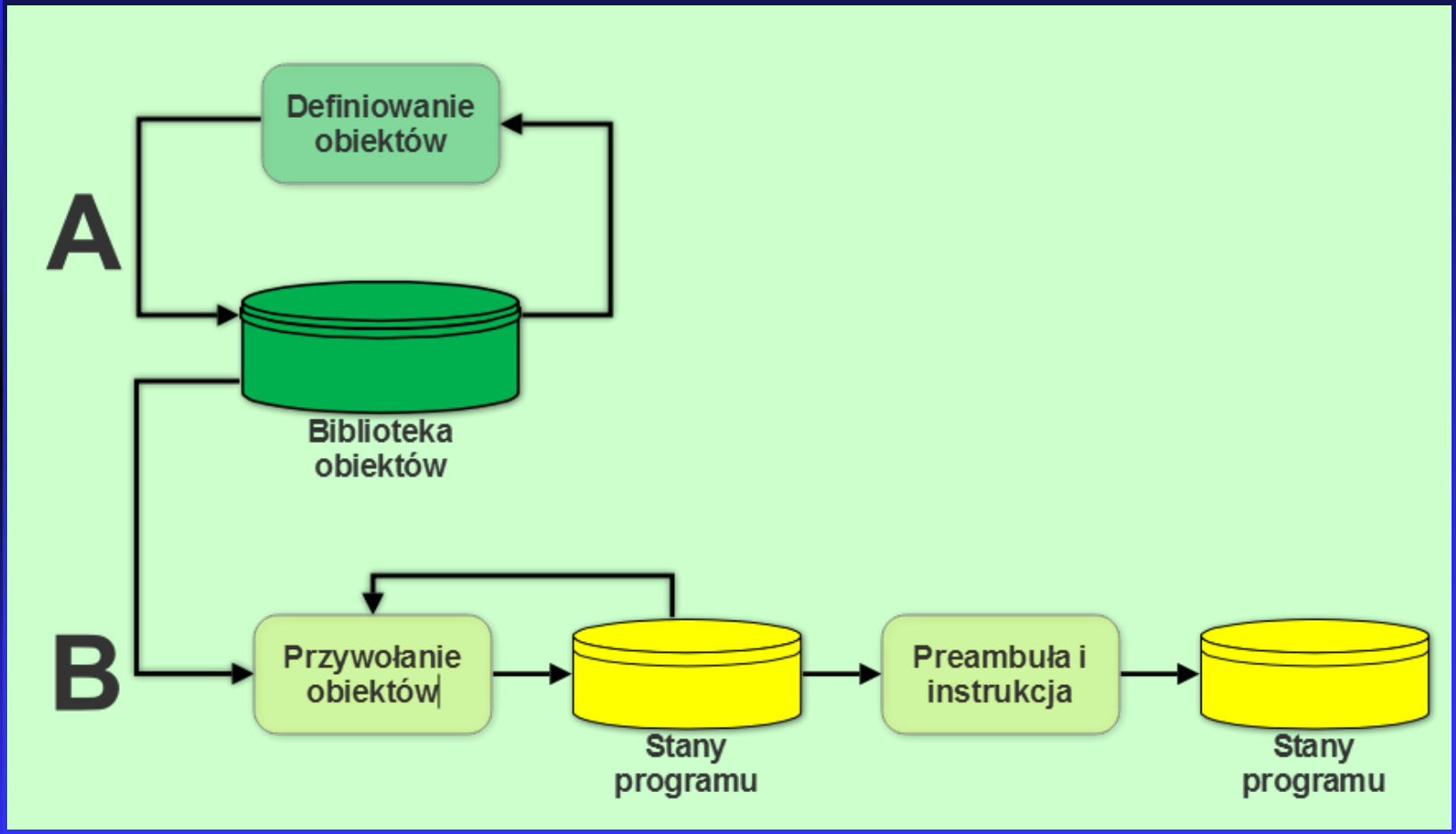
twórz-obi-def-typ : DenDefTyp ⟼ Obiekt
twórz-obi-dek-prf : DenDekPrf ⟼ Obiekt
twórz-obi-dek-pri : DenDekPri ⟼ Obiekt

sekwencja-obi : Obiekt x Obiekt ⟼ Obiekt

dwo : DenWyrObi = BibObiB ⟼ Obiekt | Błąd
ddo : DenDefObi = BibObiB ⟼ BibObiB

Lingua-3

dwa reżimy pracy przy programowaniu obiektowym



Lingua-3
denotacja przywołania obiektu

dpo : DenPrzObi = BibObi x Stan ⟼ BibObi x Stan

przywołaj-obi : Identyfikator ⟼ DenPrzObi

```
przywołaj-obi.ide.(bio, sta) =
  jest-błąd.sta ➔ (bio, sta)
  bio.ide = ?    ➔ (bio, sta ◄ `brak-obiektu-o-nazwie-`ide)
  niech
    obi          = bio.ide
    (śro, skł)   = sta
  prawda         ➔ (bio, (obi.śro, skł))
```

sekwencja-przy : DenPrzObi x DenPrzObi ⟼ DenPrzObi

Lingua-3
prefiksowanie programów

dpx : DenPrefiks = BibObi x Stan ⟼ Stan

twórz-prefiks : DenPrzObi ⟼ DenPrefiks

twórz-prefiks.dpo.(obi, sta) =
  jest-błąd.sta ➔ sta
  niech
    (obi, sta-1) = dpo.(obi, sta)
  prawda ➔ sta-1

Prefiksów nie trzeba składać sekwencyjnie bo one zawierają sekwencje przywołań

twórz-program-prefiksowany : DenPrefiks x DenPre x DenIns ⟼ DenPro
twórz-program-prefiksowany.(dpx, dpe, din) = dpx • dep • din

Przykłady problemów badawczych

Teoria i inżynieria software'owa
- modele dla języków skryptowych; HTML, TEX,...
- modele dla języków ze współbieżnością
- pełen system reguł konstruowania programów poprawnych
- pełne modele dla języków z serii Lingua

Narzędzia wspierające budowniczego języków
- automat. generator skł. abstr. z opisu algebry denotacji
- pół-aut. generator składni konkretnej
- wspomaganie tworzenia składni kolokwialnej i tr. przywracającej
- wspomaganie przy tworzeniu opisu semantyki

Narzędzia wspierające konstruowanie programów poprawnych
- języki z serii Lingua
- wspomaganie wykonywania konstruktorów programów

Eksperymentalne zastosowania, np. do mikroprogramów.

DZIĘKUJĘ ZA UWAGĘ