Wykład 10.
Obliczenia i efekty

W zeszłym tygodniu osiągnęliśmy pewien sukces: nasz język programowania osiągnął dojrzałość i udało nam się w nim napisać bardzo nietrywialny program – ewaluator wyrażeń arytmetycznych. Ale nie po to pisaliśmy interpreter, żeby mieć drugiego Racketa (bo jego już akurat mamy), ale po to, by **prześcignąć Racketa**. Oczywiście nie chodzi o prędkość ewaluacji programów, ale o ciekawe konstrukcje w języku, których możemy teraz dodać sobie tyle, ile tylko zechcemy.

W tym tygodniu zajmiemy się właśnie tym, ale będziemy mieli ukryty zamysł. Zamysłem tym jest próba przekonania Państwa, że konstruując programy naprawdę warto myśleć o modularności i rozdzieleniu części programu tak, by były od siebie niezależne. Takie rozdzielanie nieraz przypomina próbę rozdzielenia porostu na glona i grzyba, ale – mam nadzieję, że sami się o tym Państwo za chwilę przekonacie – warto!

W szczególności będziemy rozbudowywać nasz język o **efekty obliczeniowe**. Będziemy mieć język, w którym można rzucić wyjątek, inny, w którym możemy niedeterministycznie wybrać jedną z kilku wartości, a także jeszcze inny, w którym mamy jedną komórkę mutowalnego stanu. Co więcej, całkowicie rozdzielimy kod interpretera od kodu, który definiuje efekt.

10.1. Procedury wbudowane

Do tej pory rozbudowywaliśmy nasz interpreter w sposób dość chaotyczny, dodając nowe konstrukcje bez zastanowienia, jak wpływają one na wcześniej zaimplementowane funkcjonalności. Przykładem są operatory arytmetyczne, które w Rackecie są procedurami, więc możemy w nim napisać np.

```
> (map + '(1 2 3) '(10 20 30))
'(11 22 33)
```

U nas tak nie można, bo + jest operatorem arytmetycznym (a więc czymś w rodzaju formy specjalnej), a nie procedurą, np.

```
> (eval (parse '((lambda (op) (op 2 3)) +)))
ERROR: Unknown identifier +
```

Oczywiście definiując + przy okazji języka wyrażeń arytmetycznych, nie mogliśmy zrobić go procedurą z oczywistych względów: nie mieliśmy jeszcze wtedy procedur. A gdy wprowadziliśmy procedury, nie chcieliśmy komplikować i majstrować przy reszcie interpretera. Ale teraz, ze względów praktycznych, nadszedł czas na porządki w interpreterze.

Trochę bardziej uporządkowaną wersję interpretera znajdą Państwo w pliku **fun.rkt**. Tutaj omówimy tylko jedną (najważniejszą) nowość: **procedury wbudowane**.

Jak zrobić + procedurą? Nie da rady zdefiniować go bez pomocy z „zewnątrz", bo samymi lambdami i if-ami nie jesteśmy w stanie ruszyć maszynowego int-a. Dlatego wprowadzamy nowy rodzaj wartości dla procedur, które nie są definiowane przez programistę, a są częścią składową interpretera. Sprawę komplikuje fakt, że uparliśmy się, by w naszym języku wszystkie procedury były jednoargumentowe, podczas gdy procedury wbudowane, żeby móc zadziałać, potrzebują konkretnej liczby argumentów (np. + potrzebuje dwóch). Dlatego będziemy reprezentować nie tyle procedury, co częściowe aplikacje procedur wbudowanych. Taka wartość musi pamiętać, jakie argumenty już dostała, a ilu jeszcze oczekuje. Dopiero aplikacja takiej wartości do ostatniego oczekiwanego argumentu uruchamia zamknięty w środku racketowy kod. Wbudowane procedury reprezentujemy przy pomocy struktury `builtin`:

fun.rkt

```
163 (struct builtin
164   (proc   ; racketowa procedura, ktora nalezy uruchomic
165    args   ; lista dotychczasowych argumentow
166    pnum)) ; liczba brakujacych argumentow
```

Przykładowo, chcemy by:

```
> (eval (parse '+))
(builtin #<procedure:+> '() 2)
> (eval (parse '(+ 5)))
(builtin #<procedure:+> '(5) 1)
> (eval (parse '(+ 5 7)))
12
```

Definiujemy środowisko początkowe, które zawiera interesujące nas procedury wbudowane:

fun.rkt

```
166 (define (builtin/1 p)
167   (builtin p null 1))
168 (define (builtin/2 p)
169   (builtin p null 2))
170 ...
196 (define start-env
197   (foldl (lambda (p env) (env-add (first p) (second p) env))
198          env-empty
199          `((+ ,(builtin/2 +))
200            (- ,(builtin/2 -)) ... )))
```

Interpreter wywołujemy nie z pustym środowiskiem, a z `start-env`:

fun.rkt

```
263 (define (eval e)
264   (eval-env e start-env))
```

Pozostaje rozszerzyć interpreter o przypadek, gdy w aplikacji wartością aplikowanego wyrażenia jest procedura wbudowana. W procedurze `eval-env` dodajemy nowy przypadek w wyrażeniu `match`:

fun.rkt

```
224 (define (eval-env e env)
225   (match e
226     ...
251     [(app-expr ef ea)
252      (let ((vf (eval-env ef env))
253            (va (eval-env ea env)))
254        (match vf
255          [(clo x e env)
256           (eval-env e (env-add x va env))]
257          [(builtin p args nm)
258           (if (= nm 1)
259               (apply p (reverse (cons va args)))
260               (builtin p (cons va args) (- nm 1)))]))]))
```

Wiersz 259 to przypadek, gdy procedura wbudowana dostała właśnie swój ostatni argument i możemy wykonać jej ciało. Używamy do tego racketowej procedury `apply`, która aplikuje procedurę do argumentów zgromadzonych na liście, np.:

```
(apply f (list x y z))
```

Wyrażenie to oznacza to samo, co

```
(f x y z)
```

Wiersz 260 to przypadek, gdy argument, do którego aplikujemy, nie jest ostatnim argumentem (czyli częściowa aplikacja wciąż jest częściowa). Wtedy wynikiem jest nowa wartość `builtin` z zapamiętaną wartością kolejnego argumentu.

10.2. Programowanie z błędem (podejście pierwsze)

Gdy coś pójdzie nie tak, interpreter z pliku **fun.rkt** wywołuje racketową procedurę `error`, co zrywa działanie całego programu.[1] Teraz zrobimy tak, by interpreter umiał kończyć się błędem bez użycia zakulisowych działań (czyli efektów wbudowanych w Racketa).

Zrobimy to w ten sposób, że interpreter będzie zwracał wartość lub strukturę `runtime-error` (chociaż może „`user-error`" byłoby lepszą nazwą). Do środowiska dodajemy też wbudowaną procedurę, która służy właśnie do zwracania tej struktury.[2] Taką racketową wartość, która jest albo wartością naszego języka albo błędem nazwiemy **obliczeniem** (ang. *computation*). W kolejnych sekcjach pojęcie to uogólnimy.

error-naive.rkt

```
195 (struct runtime-error (reason))
196 ...
199 (define start-env
200 ...
223   (error ,(builtin/1 runtime-error)) ... )
```

Teraz musimy zmodyfikować interpreter tak, by brał pod uwagę, że interpretacja podwyrażenia może skończyć się błędem. Wówczas interpretacja całego wyrażenia kończy się (tym samym) błędem. Przykładowo, by zinterpretować

[1] W pliku **fun.rkt** znajduje się też interaktywny interpreter (REPL), który te błędy przechwytuje i się nie wywraca, gdy w interpretowanym programie zachodzi błąd, ale dziś będziemy robić takie rzeczy czysto funkcyjnie.

[2] Proszę zwrócić uwagę, że trochę brakuje nam tu nomenklatury, bo normalnie nie powiedzielibyśmy „zwraca strukturę `runtime-error`", ale że zwraca **wartość** `runtime-error`. Niestety, słowa „wartość" używamy zarówno do określenia wartości w Rackecie, jak i do specjalnych racketowych wartości, które są wynikiem działania naszego interpretera. Proszę zachować czujność!

program `(let [x e1] e2)` musimy najpierw zinterpretować `e1`. Jeśli interpretacją `e1` jest `runtime-error`, jest to też wynik interpretacji całego let-wyrażenia. Jeśli rezultatem interpretacji `e1` jest wartość, kontynuujemy tak jak dotychczas:

error-naive.rkt

```
229 (define (eval-env e env)
230   (match e
231     ...
237     [(let-expr x e1 e2)
238      (let ((v1 (eval-env e1 env)))
239        (if (runtime-error? v1)
240            v1
241            (eval-env e2 (env-add x v1 env))))] ... ))
```

W przypadku aplikacji robi się bardziej tłoczno, bo musimy najpierw obliczyć wartość dwóch podwyrażeń. Każde z nich może zakończyć się błędem i musimy wziąć to pod uwagę:

error-naive.rkt

```
261 (define (eval-env e env)
262   (match e
263     ...
237     [(app-expr ef ea)
238      (let ((vf (eval-env ef env)))
239        (if (runtime-error? vf)
240            vf
241            (let ((va (eval-env ea env)))
242              (if (runtime-error? va)
243                  va
244                  (match vf
245                    [(clo x e env)
246                     (eval-env e (env-add x va env))]
247                    [(builtin p args nm)
248                     (if (= nm 1)
249                         (apply p (reverse (cons va args)))
250                         (builtin p (cons va args) (- nm
                                1)))] ... ))
```

Et cetera, et cetera... W każdym miejscu interpretera musimy badać wynik ewaluacji podwyrażeń, żeby propagować ewentualny błąd. Gdy tylko widzimy takie mechaniczne ozdabianie programu powtarzalnym kodem, intuicyjnie czujemy, że można napisać nasz interpreter trochę lepiej.

10.3. Programownie z błędem (podjeście drugie)

Zanim zrobimy lepiej, powróćmy do uwagi z wykładu 7., że let-wyrażenia w języku z procedurami wydają się niczym więcej niż lukrem syntaktycznym. Bardziej szczegółowo, racketowe wyrażenie

```
(let ([x e1]) e2)
```

można rozumieć jako lukier dla wyrażenia

```
((lambda (x) e2) e1)
```

Jak ktoś chce, może tę intuicję „sformalizować" i powołać do życia procedurę do robienia let-obliczeń:

```
(define (my-let c k) (k c))
```

Można użyć jej do zapisania let-wyrażenia w następujący sposób:

```
(my-let e1 (lambda (x) e2))
```

Semantycznie nie zyskujemy nic ciekawego, a czytelność kodu tylko się pogorszyła. Więc po co nam taka procedura? Dlatego, że jest ona czymś, co można uogólnić i użyć do rozwiązania naszego problemu.

Główną obserwacją jest to, że w naszym interpreterze wszystkie wyrażenia kształtu

```
(let ([x (eval-env ...)]) e
```

zamieniliśmy na

```
(let ([x (eval-env ...)])
  (if (runtime-error? x)
      x
      e))
```

Musieliśmy też powyciągać wywołania procedury `eval-env` z wewnątrz wyrażeń i zamknąć je w takie let-if-y. Nasz problem z eksplozją brzydoty kodu interpretera rozwiązujemy zmieniając zwykłe let-wyrażenia na takie, które sami zdefiniujemy. Taki `let` interpretuje najpierw pierwsze wyrażenie, a następnie sprawdza, czy skończyło się ono `runtime-error`-em. Jeśli tak, propaguje błąd, jeśli nie, kontynuuje z drugim wyrażeniem (tak jak zwykły `let`):

error-composition.rkt

```
197 (define (let-error c k)
198   (if (runtime-error? c) c (k c)))
```

Teraz możemy w interpreterze użyć procedury `let-error` tak jak zwykłego let-wyrażenia (ale zapisanego w stylu `my-let`). Zajmie się ona ewentualną propagacją błędu:

error-composition.rkt

```
232 (define (eval-env e env)
233   (match e
234     ...
240     [(let-expr x e1 e2)
241      (let-error (eval-env e1 env) (lambda (v1)
242        (eval-env e2 (env-add x v1 env))))]
243     ...
261     [(app-expr ef ea)
262      (let-error (eval-env ef env) (lambda (vf)
263      (let-error (eval-env ea env) (lambda (va)
264        (match vf
265          [(clo x e env)
266           (eval-env e (env-add x va env))]
267          [(builtin p args nm)
268           (if (= nm 1)
269               (apply p (reverse (cons va args)))
270               (builtin p (cons va args) (- nm 1)))])))))]
271      ... ))
```

Obowiązkowa prezentacja działania:

```
> (eval (parse '(+ 4 (error "tu jest sobie blond"))))
(runtime-error "tu jest sobie blond")
```

10.4. Niedeterminizm

Innym efektem, o który można rozbudować język z pliku **fun.rkt**, jest **niedeterminizm**. Język rozbudowujemy o wbudowaną procedurę `choose`, która pozwala niedeterministycznie wybrać wartość spośród dwóch możliwości, a interpreter zwróci nam listę wszystkich możliwych wyników działania programu. Tego typu konstrukcje, które pozwalają nam łatwo eksplorować konsekwencje możliwych wyborów, przydają się do pisania programów rozwiązujących zagadki typu sudoku czy problem ośmiu hetmanów. Być może przykład rozjaśni wątpliwości:

```
> (eval (parse '(+ (choose 1 2) (choose 10 20))))
'(11 21 12 22)
```

Żeby zaimplementować taki język, musimy inaczej zdefiniować pojęcie **obliczenia**. Tym razem jest to lista wartości.

Oznacza to, że modyfikując interpreter, musimy pamiętać, że wynikiem interpretacji podwyrażenia jest teraz lista wartości i że sami musimy zwrócić listę wartości. Przykładowo, obliczenie stałej to już nie po prostu ta stała, ale jednoelementowa lista zawierająca tą stałą:

nondeterminism-composition.rkt

```
237 (define (eval-env e env)
238   (match e
239     [(const n)
240      (list n)] … ))
```

Żeby elegancko poradzić sobie z komponowaniem obliczeń zwracających wiele wyników, użyjemy znanej już Państwu procedury `concat-map`:

nondeterminism-composition.rkt

```
234 (define (concat-map xs f) (append-map f xs))
```

Przypisuje ona każdemu elementowi `x` listy `xs` listę `(f x)`, a następnie konkatenuje wynikowe listy. Na przykład, zakładając, że `(factors n)` to lista dzielników liczby `n` w kolejności rosnącej:

```
> (concat-map '(2 6 4) factors)
'(1 2 1 2 3 6 1 2 4)
```

Intuicyjnie, jeśli `xs` to lista możliwych wartości podwyrażenia, a `(f x)` to lista możliwych wartości programu przy założeniu, że wartością podwyrażenia jest `x`, to `(concat-map xs f)` daje nam listę wszystkich możliwych wyników programu. Rozważmy, jak nasz interpreter interpretuje let-wyrażenia:

nondeterminism-composition.rkt

```
244 (define (eval-env e env)
245   (match e
246     …
245     [(let-expr x e1 e2)
246      (concat-map (eval-env e1 env) (lambda (v1)
247        (eval-env e2 (env-add x v1 env))))] … ))
```

W powyższym programie:

- `(eval-env e1 env)` to lista zawierająca możliwe wartości wyrażenia `e1`
- `(eval-env e2 (env-add x v1 env))` to lista zawierająca możliwe wartości wyrażenia `e2` przy założeniu, że `v1` jest wartością wyrażenia `e1`.
- Procedura `concat-map` umie odpowiednio połączyć te wyrażenia i zakumulować możliwe wartości.

Proszę zwrócić uwagę, że w ciele procedury

```
(lambda (v1)
  (eval-env e2 (env-add x v1 env)))
```

nie przejmujemy się tym, że `e1` mogło zwrócić wiele wartości. Nas interesuje tylko wartość `v1`. Procedura `concat-map` wywoła powyższą procedurę dla każdej wartości w wyniku ewaluacji wyrażenia `e1` i skonkatenuje rezultaty.

Ten sam mechanizm zadziała w innych przypadkach obliczania wartości podwyrażeń. Np. klauzulę dla aplikacji można wyrazić jak następuje:

nondeterminism-composition.rkt

```
237 (define (eval-env e env)
238   (match e
239     ...
265     [(app-expr ef ea)
266      (concat-map (eval-env ef env) (lambda (vf)
267      (concat-map (eval-env ea env) (lambda (va)
268        (match vf
269          [(clo x e env)
270           (eval-env e (env-add x va env))]
271          [(builtin p args nm)
272           (if (= nm 1)
273               (apply p (reverse (cons va args)))
274               (list (builtin p (cons va args) (- nm 1))))]
275     ... ))
```

Modyfikujemy też procedury wbudowane. Nie mogą one zwracać wartości, a jednoelementową listę wartości (podobnie jak stałe):

nondeterminism-composition.rkt

```
166 (define (builtin/1 p)
167   (builtin (lambda (x) (list (p x))) null 1))
168 (define (builtin/2 p)
169   (builtin (lambda (x y) (list (p x y))) null 2))
```

Ostatnią modyfikacją, jakiej potrzebujemy, jest dodanie wbudowanej procedury służącej do dokonywania niedeterministycznego wyboru. Ponieważ nie zwraca ona listy jednoelementowej, tworzymy dla niej specjalne konstruktory, które nie opakowują wyniku w listę. Na koniec, wywołujemy interpreter z początkowym środowiskiem `effect-env`:

nondeterminism-composition.rkt

```
223 (define (effect-builtin/1 p)
224   (builtin p null 1))
225 (define (effect-builtin/2 p)
226   (builtin p null 2))
227
228 (define effect-env
229   (foldl (lambda (p env) (env-add (first p) (second p) env))
230          start-env
231          `((choose ,(effect-builtin/2 (lambda (x y) (list x
                y))))
232            )))
233 ...
277 (define (eval e)
278   (eval-env e effect-env))
```

Żeby w naszym języku rzeczywiście dało się rozwiązywać sudoku, potrzebujemy jeszcze procedury wbudowanej `fail`. Zadanie zdefiniowania jej pozostawiamy Państwu na ćwiczenia.

10.5. Abstrahujemy

Teraz dokonamy rzeczy dość zaawansowanej, czyli znajdziemy podobieństwa pomiędzy implementacjami i dokonamy abstrakcji odpowiednich fragmentów kodu. W okolicach pierwszego wykładu mówiliśmy o „abstrakcji proceduralnej" która pozwala nam pewne wspólne schematy wyodrębnić i nadać im nazwę. To właśnie zrobimy tutaj. Spójrzmy na interpreter dla programowania z błędem i programowania z niedeterminizmem. Klauzule dla let-wyrażeń:

error-composition.rkt

```
[(let-expr x e1 e2)
 (let-error (eval-env e1 env)
    (lambda (v1)
   (eval-env e2 (env-add x v1
      env))))]
```

nondeterminism-composition.rkt

```
[(let-expr x e1 e2)
 (concat-map (eval-env e1 env)
    (lambda (v1)
   (eval-env e2 (env-add x v1
      env))))]
```

Klauzule dla aplikacji:

error-composition.rkt

```
[(app-expr ef ea)
 (let-error (eval-env ef env)
    (lambda (vf)
 (let-error (eval-env ea env)
    (lambda (va)
   (match vf
     [(clo x e env)
      (eval-env e (env-add x va
          env))]
     [(builtin p args nm)
      (if (= nm 1)
          (apply p (reverse
              (cons va args)))
          (builtin p (cons va
              args) (- nm
              1)))])))))]
```

nondeterminism-composition.rkt

```
[(app-expr ef ea)
 (concat-map (eval-env ef env)
    (lambda (vf)
 (concat-map (eval-env ea env)
    (lambda (va)
   (match vf
     [(clo x e env)
      (eval-env e (env-add x va
          env))]
     [(builtin p args nm)
      (if (= nm 1)
          (apply p (reverse
              (cons va args)))
          (list (builtin p
              (cons va args) (-
              nm 1))))]
```

Łatwo zauważyć, że `concat-map` w interpreterze z niedeterminizmem pełni tę samą rolę, co `let-error` w programowaniu z błędem. To dlatego, że to tak naprawdę też rodzaj let-wyrażenia, które pozwala nam składać ze sobą obliczenia. Różnicą jest to, że interpreter dla niedeterminizmu musi owijać pojedyncze wartości (np. w stałych czy procedurach wbudowanych) w jednoelementową listę, ale ponieważ obliczenia z błędem to „suma" wartości i błędu, ta operacja **podniesienia** wartości do obliczenia jest po prostu identycznością. Przykładowo, generyczna klauzula dla aplikacji zdefiniowana jest następująco:

nondeterminism-monadic.rkt

```
269 [(app-expr ef ea)
270  (bind (eval-env ef env) (lambda (vf)
271  (bind (eval-env ea env) (lambda (va)
272    (match vf
273      [(clo x e env)
274       (eval-env e (env-add x va env))]
275      [(builtin p args nm)
276       (if (= nm 1)
277           (apply p (reverse (cons va args)))
278           (return (builtin p (cons va args) (- nm 1))))]
```

Dokonujemy abstrakcji poprzez sparametryzowanie interpretera implementacją tego, co wcześniej było procedurami `let-error` i `concat-map` (nazywamy ten parametr `bind`) oraz procedurą podniesienia wartości do obliczenia, którą nazywamy `return`. Wówczas, by rozbudować taki generyczny interpreter o, powiedzmy, niedeterminizm, wystarczy dokonać instancjacji tych parametrów jak poniżej:[3]

nondeterminism-monadic.rkt

```
230 (define effect-env
231   (foldl (lambda (p env) (env-add (first p) (second p) env))
232     start-env
233     `((choose ,(effect-builtin/2 (lambda (x y) (list x y))))
234       )))
235
236 (define (bind c k) (append-map k c))
237
238 (define (return x) (list x))
```

10.6. Mutowalna komórka stanu

Jako przykład, że nasz interpreter rzeczywiście da się parametryzować różnymi efektami, sprametryzujmy go pojedynczą mutowalną komórką pamięci, ktorą implementujemy czysto funkcyjnie. Oznacza to, że obliczenie w naszym interpreterze będzie procedurą, która przyjmuje początkową wartość stanu i zwraca parę zawierającą końcową wartość i końcowy stan. Np. procedura

[3] Gdybyśmy znali racketowy system modułów, moglibyśmy naprawdę napisać generyczny interpreter naprawdę parametryzowany implementacją efektu, ale nie chcemy Państwu zbyt komplikować w tym momencie.

```
(lambda (s) (cons "abc" (+ s 1)))
```

reprezentuje obliczenie, którego wartością jest napis "abc", a które zwiększa wartość w komórce pamięci o 1.

Wartość podniesiona do obliczenia to procedura, która nie zmienia stanu i zwraca tę wartość w pierwszym elemencie pary. Procedura `bind` używa końcowego stanu pamięci pierwszego obliczenia, by użyć go jako początkowego stanu pamięciu drugiego obliczenia. Potrzebujemy też wbudowanych procedur `put` do przypisania komórce pamięci jakieś wartości i `get` do pobierania aktualnej wartości (ponieważ nie ma w naszym języku procedur zeroargumentowych, `get` bierze argument, ale nic z nim nie robi).

state-monadic.rkt

```
232 (define effect-env
233   (foldl (lambda (p env) (env-add (first p) (second p) env))
234          start-env
235          `((put ,(effect-builtin/1
236              (lambda (n) (lambda (o) (cons false n)))))
237            (get ,(effect-builtin/1
238              (lambda (f) (lambda (s) (cons s s)))))
239            )))
240
241 (define (bind c k)
242   (lambda (s)
243     (let ((r (c s)))
244       ((k (car r)) (cdr r)))))
245
246 (define (return x)
247   (lambda (s) (cons x s)))
```

Mutowalnej komórki satanu możemy użyć np. do zaimplementowania prostego generatora liczb losowych, który w komórce trzyma aktualne ziarno (ang. *seed*):

state-monadic.rkt

```
296 (define RAND-GENERATOR
297   '(letrec [rand (lambda (x)
298       (let [x (get false)]
299       (begin (put (% (* (+ x 17) 37) 397))
300              x)))]
301    (letrec [gen (lambda (n)
```

```
302       (if (= n 0)
303           null
304           (cons (rand false) (gen (- n 1)))))]
305      (gen 20))))
```

Uruchamiamy program:

```
> (eval (parse RAND-GENERATOR))
#<procedure:...ate.-monadic.rkt:242:2>
```

Wynikiem ewaluacji tego programu jest jakaś procedura. Czemu? Bo obliczenia to procedury, które przyjmują jako argument stan początkowy. Dajmy więc tej procedurze jakiś stan:

```
> ((eval (parse RAND-GENERATOR)) 233)
'((233 119 268 223 146 76 265 112 9 168 96 211 99 322 236
    230 8 131 315 374) . 175)
```

Wynikiem jest para złożona z listy pseudolosowych wartości i wartości stanu końcowego.