Politechnika Wrocławska
Wydział Podstawowych Problemów Techniki

PRACA DYPLOMOWA

Teoria typów z definicjami indukcyjnymi jako język programowania

Marek Łach

Praca napisana pod kierunkiem
dra Zdzisława Spławskiego

Wrocław 1998

Spis treści

1. Wstęp

Pożądaną cechą języków programowania jest możliwość dowodzenia własności programów. W przypadku większości języków jest to co najmniej trudne. Przykładem języka zaprojektowanego z troską o możliwość dowodzenia własności jest Pascal. Dzięki logice Hoare'a możliwe jest udowodnienie częściowej poprawności programu. Własność stopu musi być udowodniona oddzielnie. Trzeba podkreślić, że ręczne dowodzenie własności jest w wielu przypadkach trudne i wymaga specjalnej konstrukcji programów.

Rozwiązaniem wielu problemów związanych z dowodzeniem własności programów może być zastosowanie odpowiedniej teorii typów. W wielu przypadkach własność stopu jest cechą wszystkich poprawnych programów w systemie. Dodatkowo relacja równości, określona w systemie typów, może pozwalać na automatyczne dowodzenie odpowiednio zdefiniowanej równości programów. Dzięki temu możliwe jest automatyczne sprawdzenie czy np. program spełnia zadaną specyfikację.

Zastosowanie teorii typów jako języka programowania nastręcza wiele trudności. Wybór odpowiedniego systemu jest zawsze kompromisem między jego mocą wyrażania (zbiorem funkcji, które można zdefiniować) i własnościami teoretycznymi (które wpływają na łatwość programowania w systemie). Systemy pozwalające na automatyczne wyprowadzanie najogólniejszego typu są albo zbyt ubogie, tzn. pozwalają wyrazić zbyt małą ilość funkcji (rachunek $\lambda^{\rightarrow}$), albo nie mają pożądanych własności, np. własności mocnej normalizacji (SML). System F Girarda pozwala wyrazić dużą klasę funkcji, ale nie pozwala pozwala na automatyczne wyprowadzanie typu.

W tym świetle system ET, przedstawiony przez dra Spławskiego, jest ciekawą propozycją. Z jednej strony posiada wszystkie pożądane własności teoretyczne systemów uboższych (np. możliwość automatycznego sprawdzania typu, własność mocnej normalizacji), a z drugiej pozwala na wyrażenie bogatej klasy funkcji, choć dokładna charakteryzacja tej klasy jest problemem otwartym. Trzeba podkreślić wyjątkowe własności systemu ET. Jedyny powszechnie znany system mający takie własności – Charity[1], jest znacznie uboższy.

Własności teoretyczne ET powodują, że istnieje bezpośredni związek między programami w ET, a dowodami w logice intuicjonistycznej. Każda funkcja wyrażona w ET jest dowodem odpowiedniego twierdzenia w systemie dedukcji naturalnej. Odpowiednie przykłady zawarte są w niniejszej pracy.

Celem pracy było zapoznanie się z systemem ET i jego implementacja. Ze względu na teoretyczne własności systemu, osiągnięcie celu wymagało zapoznania się nie tylko z teorią kompilacji, ale także z podstawowymi formalizmami z teorii funkcji i typów, oraz z logiką intuicjonistyczną i systemem dedukcji naturalnej.

Ze względu na eksperymentalny charakter systemu i jego ciągły rozwój implementacja została wykonana ze szczególną dbałością o możliwość późniejszego dokładania nowych cech systemu.

Z uwagi na użyteczność i wygodę zaimplementowano w sposób efektywny liczby naturalne

[1] Charity jest językiem opartym na teorii kategorii. Powstał w 1990 r. i od tego czasu jest intensywnie rozwijany na uniwersytecie w Calgary. Informacje na temat Charity znajdują się na stronie `http://www.cpsc.ucalgary.ca/projects/charity/home.html`.

wraz z podstawowymi operacjami, bez straty własności teoretycznych.

Praca została podzielona na sześć rozdziałów.

Rozdział 2 zawiera pobieżny opis formalizmów, których znajomość jest niezbędna do zapoznania się z systemem ET. Przedstawione są podstawowe formalizmy z teorii funkcji i typów: beztypowy rachunek λ, rachunek λ z typami, system T Gödla i system F Girarda. Kolejno opisane są: logika intuicjonistyczna i i system dedukcji naturalnej. Ostatni podrozdział zawiera izomorfizm Curry'ego-Howarda formalizujący związek między logiką intuicjonistyczną i rachunkiem λ z typami.

W rozdziale 3 przedstawiony jest system ET. Na początku jest opisana najprostsza postać ET. W kolejnych podrozdziałach znajdują się reguły rozszerzające system o możliwość definiowania typów i kotypów.

Rozdział 4 zawiera opis i przykłady użycia języka ET. Przykłady są demonstracją siły wyrażania ET i zwracają uwagę na teoretyczne własności programów.

W rozdziale 5 przedstawiono schemat działania kompilatora ET. Kolejne podrozdziały zawierają opis, podstawy teoretyczne i rozwiązanie kolejnych etapów kompilacji.

Rozdział 6 jest podsumowaniem całości pracy.

Ze względu na ogromną ilość pozycji źródłowych z dziedzin wykorzystywanych w pracy, spis literatury zawiera tylko reprezentatywne pozycje monograficzne.

2. Wprowadzenie

System ET jest jednocześnie systemem logicznym i językiem programowania . W niniejszym rozdziale opisane są formalizmy, na których jest oparta definicja systemu ET. Podrozdziały 2.1, 2.2, 2.3 i 2.4 zawierają definicje podstawowych formalizmów z teorii funkcji i typów: beztypowego rachunku λ, rachunku λ z typami, systemu T Gödla i systemu F Girarda. Podrozdziały 2.5 i 2.6 przedstawiają logikę intuicjonistyczną i system dedukcji naturalnej. W ostatnim podrozdziale 2.8 opisany jest związek między logiką intuicjonistyczną i rachunkiem λ z typami.

Z uwagi na pobieżny charakter wprowadzenia, twierdzenia przedstawione są bez dowodów. Dowody wraz z bardziej szczegółowym opisem poszczególnych teorii znajdują się w cytowanej literaturze.

2.1. Rachunek λ

Rachunek λ został zaproponowany przez Churcha w latach 30-tych. Jest formalnym systemem umożliwiającym definiowanie funkcji, rozumianych nie jako zbiór par (argument i odpowiadający mu wynik funkcji), ale jako pewien algorytm. W tym ujęciu funkcja jest operacją, którą można zaaplikować do argumentu by otrzymać wynik. Udowodniono równoważność rachunku λ z innymi modelami obliczalności, tzn. że przy pomocy λ-termów — wyrażeń w rachunku λ — można wyrazić dokładnie te same funkcje, które można wyrazić przy pomocy maszyny Turinga, algorytmów Markowa czy funkcji rekurencyjnych.

Definicja 2.1. Zbiór λ-termów Λ *definiuje się przy użyciu nieskończonego, przeliczalnego zbioru zmiennych* $\mathcal{V} = \{v, v', v'', \dots\}$ *i dwóch podstawowych operacji — aplikacji i abstrakcji funkcyjnej.*

$$\mathcal{V} ::= v \mid \mathcal{V}'$$
$$\Lambda ::= \mathcal{V} \mid (\Lambda\Lambda) \mid (\lambda\mathcal{V}.\Lambda)$$

Dla uproszczenia zapisu stosuje się następujące reguły notacyjne.

Notacja 2.2.

1. Małe litery (np. x,y,x_1) oznaczają zmienne.
2. Wielkie litery (np. M,N,P) oznaczają λ termy.
3. $\lambda x_1 \dots x_n.M$ oznacza[1] $(\lambda x_1.(\lambda x_2. \dots (\lambda x_n.M) \dots))$.
4. $M_1 \dots M_n$ oznacza[2] $(\dots (M_1 M_2) \dots M_n)$.

[1] Innymi słowy abstrakcja wiąże w prawo.
[2] Aplikacja wiąże w lewo.

Definicja 2.3. Zbiór zmiennych wolnych termu M, *oznaczany przez* $FV(M)$, *i* zbiór zmiennych związanych, *oznaczany przez* $BV(M)$, *definiuje się przez indukcję po strukturze termu:*

$$\begin{aligned} FV(x) &= \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \end{aligned}$$

$$\begin{aligned} BV(x) &= \emptyset \\ BV(MN) &= BV(M) \cup BV(N) \\ BV(\lambda x.M) &= BV(M) \cup \{x\} \end{aligned}$$

***Przykład* 2.4.** $FV(\lambda x.xy) = y$, $BV(\lambda x.xy) = x$.

***Notacja* 2.5.** $M \equiv N$ oznacza tekstową identyczność termów M i N z dokładnością do zamiany nazw zmiennych związanych.

Definicja 2.6. *Wynik podstawiania* N *za wolne wystąpienia zmiennej* x *w termie* M*, oznaczany przez* $M[x := N]$*, można zdefiniować indukcyjnie przez:*

$$\begin{aligned} x[x := N] &\equiv N \\ y[x := N] &\equiv y \\ (P\ Q)[x := N] &\equiv (P[x := N])(Q[x := N]) \\ (\lambda x.P)[x := N] &\equiv \lambda x.P \\ (\lambda y.P)[x := N] &\equiv \lambda y.(P[x := N]) \quad \textit{jeśli } y \notin FV(N) \textit{ lub } x \notin FV(P) \\ (\lambda y.P)[x := N] &\equiv \lambda z.(P[y := z][x := N]) \quad \textit{jeśli } y \in FV(N) \textit{ i } x \in FV(P), \\ & \qquad \textit{gdzie } z \textit{ jest dowolną zmienną taką,} \\ & \qquad \textit{że } z \notin FV(N) \cup FV(P) \end{aligned}$$

Reguły wnioskowania przedstawione poniżej umożliwiają dowodzenie równości termów w rachunku λ. Reguły przedstawione są w postaci

$$\frac{P_1 \quad P_2 \quad \ldots \quad P_n}{W}\,\mathrm{Reg},$$

gdzie P_i oznaczają przesłanki, które muszą być dowiedzione, by przy pomocy reguły Reg wyciągnąć wniosek W. Reguły nie posiadające przesłanek to aksjomaty. Dowód reprezentowany jest przez drzewo wywodu zbudowane przez zastępowanie przesłanek w regułach wnioskowania przez ich dowody. Prosty dowód znajduje się w przykładzie 2.8.

Definicja 2.7. *Reguły wnioskowania dla rachunku λ*

$$\frac{}{\lambda x.M \;=\; \lambda y.M[x := y]}\,\alpha \quad \text{o ile } y \notin FV(M) \cup BV(M)$$

$$\frac{}{(\lambda x.M)\;N \;=\; M[x := N]}\,\beta$$

$$\frac{}{(\lambda x.M)\;x \;=\; M}\,\eta \quad \text{o ile } x \notin FV(M)$$

$$\frac{}{M \;=\; M}\,\text{Refl} \qquad\qquad \frac{N \;=\; M}{M \;=\; N}\,\text{Sym}$$

$$\frac{K \;=\; M \qquad M \;=\; N}{K \;=\; N}\,\text{Trans}$$

$$\frac{K \;=\; L \qquad M \;=\; N}{K\;M \;=\; L\;N}\,\text{MonApp} \qquad\qquad \frac{M \;=\; N}{\lambda x.M \;=\; \lambda x.N}\,\text{MonAbs}$$

***Przykład* 2.8.** Dowód równości termów $(\lambda xy.x)\;(\lambda z.z)$ i $(\lambda x.x)\;(\lambda yz.z)$.

$$\dfrac{\dfrac{}{(\lambda xy.x)\;(\lambda z.z) = \lambda yz.z}\,\beta \qquad \dfrac{\dfrac{}{(\lambda x.x)\;(\lambda yz.z) = \lambda yz.z}\,\beta}{\lambda yz.z = (\lambda x.x)\;(\lambda yz.z)}\,\text{Sym}}{(\lambda xy.x)\;(\lambda z.z) = (\lambda x.x)\;(\lambda yz.z)}\,\text{Trans}$$

***Uwagi* 2.9.**

1. Dla uniknięcia wątpliwości stosuje się zapis $\lambda \vdash M = N$ by podkreślić, że równość termów M i N można udowodnić w rachunku λ.
2. Reguły wnioskowania można podzielić na aksjomaty — reguły α, β i η konwersji i reguły zapewniające, że relacja równości „=" jest zwrotna (Refl), symetryczna (Sym) i przechodnia (Trans) oraz, że zachowuje podstawowe operacje — abstrakcję (MonAbs) i aplikację (MonApp).
3. Równość w rachunku λ można zdefiniować pomijając regułę η konwersji. Dla rozróżnienia, rachunek zawierający regułę η oznacza się przez $\lambda\eta$ lub $\lambda\beta\eta$, natomiast rachunek bez tej reguły oznacza się jako $\lambda\beta$.
4. Najczęściej pomija się regułę α konwersji przenosząc regułę równości termów różniących się tylko nazwami zmiennych do metasystemu (przyjmuje się „automatycznie" równość np. $\lambda x.x$ i $\lambda y.y$).
5. Należy zauważyć, że podczas stosowania reguły β konwersji, kiedy przy podstawieniu $(\lambda y.P)[x := N]$ zachodzą warunki $y \in FV(N)$ i $x \in FV(P)$, tzn. w przypadku opisanym w definicji 2.6 w linii 6, niezbędna jest zmiana nazw zmiennych związanych. W definicji 2.6 wolne wystąpienia zmiennej y w P zastępowane są nową zmienną z. Konieczność wprowadzania nowych zmiennych ilustruje następujący przykład. Niech $M := \lambda x.xx$ i $N := \lambda yz.yz$. Wtedy nieformalny zapis dowodu równości $MN = N$ może wyglądać następująco (równość poszczególnych termów wynika z reguły β konwersji i przechodniości równości)

$$MN \equiv (\lambda x.xx)(\lambda yz.yz) = (\lambda yz.yz)(\lambda yz.yz) = \lambda z.(\lambda yz.yz)z \equiv$$

teraz konieczne jest rozróżnienie wystąpień par zmiennej z przez zmianę nazw (zmienna z zostaje zamieniona na zmienną w)

$$\equiv \lambda w.(\lambda yz.yz)w = \lambda wz.wz \equiv N$$

Problem zmiany nazw zmiennych, a więc także α konwersji, rozwiązuje notacja de Bruijna[3]. W tej notacji nie używa się zmiennych, wstawiając za ich miejsce numer określający, które wystąpienie λ je wiąże. 0 oznacza najbliższe, 1 — λ o poziom wyżej itd. W tym zapisie $\lambda x.\lambda y.x(\lambda z.\lambda y.yzx)yx$ zapisuje się jako $\lambda\lambda 1(\lambda\lambda 013)01$. Notacja de Bruijna została zastosowana w implementacji ET. Uwagi na temat implementacji ET znajdują się w podrozdziale 5.6.

2.1.1. Redukcja

Reguły wnioskowania dla rachunku λ wprowadzają, oprócz definicji równości λ termów, także ideę obliczania wartości funkcji. W uwagach 2.9 pokazany jest przykład równości termów $\lambda \vdash MN = N$. Lewa strona równości jest funkcją zaaplikowaną do argumentu. Prawa jest wynikiem obliczania wartości funkcji M dla argumentu N. Każdy dowód równości termów, przy pomocy reguły β konwersji, jest w istocie dowodem równości między funkcją aplikowaną do argumentu (lewą stroną równości), a wynikiem tej aplikacji (prawą stroną równości). Równość $\lambda \vdash MN = N$ oznacza także, że MN redukuje się do N, tzn. że N jest wynikiem wyrażenia MN. Relacja redukcji jest formalnym opisem sposobu obliczania. Funkcja zaaplikowana do argumentu redukuje się do wartości tej funkcji dla tego argumentu.

Definicja 2.10. Relację redukcji w jednym kroku $\rightsquigarrow$ *definiuje się indukcyjnie następująco:*

$$\frac{}{(\lambda x.M)N \rightsquigarrow M[x := N]} \qquad (\beta \text{ redukcja})$$

$$\frac{}{\lambda x.Mx \rightsquigarrow M} \text{ jeśli } x \notin FV(M) \qquad (\eta \text{ redukcja})$$

$$\frac{M \rightsquigarrow N}{MP \rightsquigarrow NP} \qquad \frac{M \rightsquigarrow N}{PM \rightsquigarrow PN}$$

$$\frac{M \rightsquigarrow N}{\lambda x.M \rightsquigarrow \lambda x.N}$$

Relacja redukcji $\overset{*}{\rightsquigarrow}$ *jest zwrotnym i przechodnim domknięciem* $\rightsquigarrow$*, a* relacja konwersji $\approx$ *jest zwrotnym, symetrycznym i przechodnim domknięciem* $\rightsquigarrow$*.*

Twierdzenie 2.11. *Termy M i N są w relacji konwersji wtedy i tylko wtedy, jeśli w rachunku λ można udowodnić ich równość.*

$$M \approx N \iff \lambda \vdash M = N$$

Należy zwrócić uwagę, że reguły wprowadzające relację redukcji są podobne do reguł wprowadzających relację równości. Różnica polega na pominięciu w definicji relacji redukcji reguł Refl, Sym, Trans i rozbiciu reguły MonApp na równoważne jej reguły

$$\frac{M = N}{MP = NP} \quad \text{i} \quad \frac{M = N}{PM = PN}.$$

W dalszej części pracy, przy omawianiu kolejnych formalizmów, definiowana będzie tylko relacja równości. Definicja relacji redukcji powstaje analogicznie jak w rachunku λ, czyli przez „przeniesienie” reguł definiujących relację równości.

Następujące definicje określają pojęcia związane ze stanem obliczania termu, np. term w postaci normalnej to term, którego nie da się już bardziej zredukować, interpretowany jako wynik obliczeń.

Definicja 2.12.

1. *Term jest* β redeksem, *jeśli jest postaci* $(\lambda x.M)N$ *(może być zredukowany do* $M[x := N]$ *za pomocą* β *redukcji).*
2. *Term jest* η redeksem, *jeśli jest postaci* $(\lambda x.M)x$, *gdzie* $x \notin FV(M)$ *(może być zredukowany do* M *za pomocą* η *redukcji).*
3. *Term* jest w postaci normalnej, *jeśli nie zawiera* β *ani* η *redeksów.*
4. *Term* M posiada postać normalną *(jest normalizowalny), jeśli istnieje* N *takie że* $M \approx N$ *i* N *jest w postaci normalnej.*
5. *Term* M jest mocno normalizowalny, *jeśli każdy ciąg redukcji w jednym kroku rozpoczynający się od* M *jest skończony.*

Twierdzenie 2.13 (Twierdzenie Churcha–Rossera)**.** *Jeżeli* $M \overset{*}{\leadsto} N_1$ *i* $M \overset{*}{\leadsto} N_2$ *to istnieje term* L *taki, że* $N_1 \overset{*}{\leadsto} L$ *i* $N_2 \overset{*}{\leadsto} L$.

Twierdzenie Churcha–Rossera jest kluczowe ze względu na zastosowania rachunku λ jako modelu obliczania. Wynika z niego wiele własności, z których część przedstawiona jest w postaci następującego lematu.

Lemat 2.14. *Jeśli* λ *term* M *posiada postać normalną* N *to:*

1. N *jest jedyną postacią normalną* M *(wynik obliczeń może być tylko jeden);*
2. $M \overset{*}{\leadsto} N$ *(wynik obliczeń jest osiągany przez zastosowanie redukcji);*
3. *każda strategia redukcji, czyli strategia wyboru kolejności redukowania redeksów w termie, prowadzi do* N *(wynik nie zależy od sposobu obliczania).*

Dodatkowe informacje na temat rachunku λ, wraz z dowodami przedstawionych twierdzeń, znajdują się w [2].

2.2. Rachunek $\lambda^{\rightarrow}$

Definicje funkcji w rachunku λ nie zawierają informacji o typie argumentów, do których funkcja powinna być zaaplikowana. Możliwe jest zatem np. wyliczenie wartości[3] (redukcja do postaci normalnej) wyrażenia `7 + true`, mimo iż wynikiem będzie λ term, który nie ma intuicyjnego, matematycznego znaczenia.

Termy w rachunku $\lambda^{\rightarrow}$ poza informacją o obliczaniu zawierają także informację o typie argumentów i wyników. W powyższym przykładzie operacja dodawania została zaaplikowana do argumentu złego typu, co powoduje, że w rachunku $\lambda^{\rightarrow}$ to wyrażenie nie jest poprawne.

Typy funkcji definiowanych w rachunku $\lambda^{\rightarrow}$ mogą zawierać zmienne przebiegające typy (zmienne typowe). Funkcja identyczności $\lambda x.x$ jest typu $V \rightarrow V$, gdzie V jest zmienną typową, za którą można podstawić dowolny typ ($U \rightarrow W$ oznacza, że funkcja bierze argument typu U i zwraca wynik typu W). Tak zdefiniowaną funkcję można zaaplikować do każdego argumentu (argumentu dowolnego typu), ponieważ jej definicja nie narzuca wyboru określonego typu argumentu i wyniku. Funkcje posiadające taką właściwość nazywa się funkcjami polimorficznymi, a język

[3]Po wcześniejszym zdefiniowaniu w rachunku λ liczb naturalnych, operacji dodawania i stałych logicznych.

(rachunek) który pozwala na takie definicje językiem polimorficznym[4]. W językach monomorficznych (jak np. C, Pascal) funkcja ma przypisany jeden, konkretny typ (bez zmiennych przebiegających typy). W takich językach konieczne jest oddzielne zdefiniowanie funkcji dla każdego typu.

Definicja 2.15. Zbiór wyrażeń typowych $\mathcal{T}$ *definiuje się przy pomocy przeliczalnego zbioru zmiennych typowych* $\mathcal{V}_\mathcal{T} = \{V, V', V'', \ldots\}$ *i konstruktora* $\rightarrow$ *następująco:*

$$\begin{aligned} \mathcal{V}_\mathcal{T} &::= V \mid \mathcal{V}_\mathcal{T}' \\ \mathcal{T} &::= \mathcal{V}_\mathcal{T} \mid \mathcal{T} \rightarrow \mathcal{T} \end{aligned}$$

Dla uproszczenie zapisu stosuje się następującą notację.

***Notacja* 2.16.**

1. Wielkie litery z końca alfabetu (np. X,Y,Z) oznaczają zmienne typowe;
2. Małe litery greckie (np. σ,τ,ξ) oznaczają typy;
3. $\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n$ oznacza[5] $(\tau_1 \rightarrow (\tau_2 \rightarrow \ldots \rightarrow (\tau_{n-1} \rightarrow \tau_n) \ldots))$

Ciąg poniższych definicji wprowadza zbiór termów w rachunku $\lambda^{\rightarrow}$.

Definicja 2.17.

1. *Niech* zbiór quasitermów Λ *będzie zdefiniowany jak w definicji 2.1.*
2. Środowisko typizujące *jest zbiorem deklaracji* $x : \sigma$ *(zmienna* x *jest typu* σ*), w którym każda zmienna występuje tylko raz. Środowisko typizujące będzie oznaczane przez* Γ *lub* Δ. Dziedziną środowiska typizującego Γ, *oznaczaną przez* $\mathrm{dom}(\Gamma)$, *jest zbiór zmiennych, których typy są zadeklarowane w środowisku. Zapis* $\Gamma, x : \sigma$, *gdzie* $x \notin \mathrm{dom}(\Gamma)$ *oznacza* $\Gamma \cup \{x : \sigma\}$. *Zapis* $\sigma[X := \tau]$ *oznacza typ* σ, *w którym za wszystkie wolne wystąpienia zmiennej* X *podstawiono typ* τ[6]. *Zapis* $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}[X := \tau]$ *oznacza* $\{x_1 : \sigma_1[X := \tau], \ldots, x_n : \sigma_n[X := \tau]\}$. *Zapis* $\sigma[\vec{X} := \vec{\tau}]$ *oznacza ciąg podstawień* $\sigma[X_1 := \tau_1] \ldots [X_n := \tau_n]$.
3. Termem *w rachunku* $\lambda^{\rightarrow}$ *jest quasiterm* M, *dla którego można znaleźć typ, tzn. istnieje takie* Γ *i* σ, *że przy pomocy reguł zdefiniowanych w (2.18) można wyprowadzić* $\Gamma \triangleright M : \sigma$, *co oznacza, że term* M *jest typu* σ *w środowisku typizującym* Γ.

Definicja 2.18. *Reguły typizowania dla rachunku* $\lambda^{\rightarrow}$

$$\frac{}{\Gamma, x : \sigma \triangleright x : \sigma}\,\mathrm{Ass}$$

$$\frac{\Gamma \triangleright M : \sigma \rightarrow \tau \qquad \Gamma \triangleright N : \sigma}{\Gamma \triangleright MN : \tau}\,{\rightarrow}\mathtt{e} \qquad\qquad \frac{\Gamma, x : \sigma \triangleright M : \tau}{\Gamma \triangleright \lambda x.M : \sigma \rightarrow \tau}\,{\rightarrow}\mathtt{i}$$

Własności rachunku $\lambda^{\rightarrow}$ przedstawiają następujące lematy.

Lemat 2.19 (Lemat środowiska).

1. *Jeśli* $\Gamma \subseteq \Delta$ *to* $\Gamma \triangleright M : \sigma \Rightarrow \Delta \triangleright M : \sigma$ *(jeśli typ da się wyprowadzić w uboższym środowisku, to da się wyprowadzić także w bogatszym).*

[4] Rachunek $\lambda^{\rightarrow}$ jest językiem z płytkim polimorfizmem (bez kwantyfikatorów) w odróżnieniu od np. systemu F. W pracy, poza częścią dotyczącą systemu F, termin polimorfizm odnosi się do płytkiego polimorfizmu.

[5] $\rightarrow$ wiąże w prawo

[6] W rachunku $\lambda^{\rightarrow}$ wszystkie zmienne występujące w typie są wolne. Dopiero w systemie F zmienne typowe mogą być wiązane przez abstrakcję typową. Definicja podstawienia w postaci przedstawionej powyżej jest odpowiednia w obu systemach.

2. $\Gamma \triangleright M : \sigma \Rightarrow FV(M) \subseteq \mathrm{dom}(\Gamma)$ *(typy zmiennych wolnych muszą być zadeklarowane w środowisku).*
3. $\Gamma \triangleright M : \sigma \Rightarrow \Delta \triangleright M : \sigma$ *gdzie* $\Delta \subseteq \Gamma$ *i* $\mathrm{dom}(\Delta) = FV(M)$ *(wyprowadzenie typu dla termu wymaga deklaracji w środowisku tylko typów zmiennych wolnych).*

Lemat 2.20 (Lemat wyprowadzania typu)**.**

1. $\Gamma \triangleright x : \sigma \Rightarrow (x : \sigma) \in \Gamma$ *(typ zmiennej zadeklarowany jest w środowisku).*
2. $\Gamma \triangleright MN : \sigma \Rightarrow \exists\tau\ (\Gamma \triangleright M : \tau \to \sigma\ \wedge\ \Gamma \triangleright N : \tau)$ *(wyprowadzenie typu aplikacji wymaga wyprowadzenia typu argumentu i typu funkcji).*
3. $\Gamma \triangleright \lambda x.M : \sigma \Rightarrow \exists\tau\xi\ (\Gamma, x : \tau \triangleright M : \xi\ \wedge\ \sigma \equiv \tau \to \xi)$ *(wyprowadzenie typu abstrakcji funkcyjnej wymaga wyprowadzenia typu treści funkcji).*

Lemat 2.21 (Lemat podstawiania)**.**

1. $\Gamma \triangleright M : \sigma \Rightarrow \Gamma[X := \tau] \triangleright M : \sigma[X := \tau]$ *(postawienie za zmienną typową (X) dowolnego typu (τ) powoduje identyczną zamianę w wyprowadzonym typie (σ)).*
2. *Jeśli* $\Gamma, x : \tau \triangleright M : \sigma$ *i* $\Gamma \triangleright N : \tau$ *to* $\Gamma \triangleright M[x := N] : \sigma$ *(podstawienie za zmienną termu o identycznym typie nie zmienia wyprowadzonego typu).*

Lemat 2.22 (Lemat zachowywania typu)**.** *Jeśli* $\Gamma \triangleright M : \sigma$ *i* $M \overset{*}{\rightsquigarrow} N$ *to* $\Gamma \triangleright N : \sigma$ *(relacja redukcji zachowuje typ).*

Poniżej przedstawione są reguły wnioskowania dla rachunku $\lambda^{\to}$. Relację redukcji i konwersji wprowadza się analogicznie jak w rachunku λ (definicja 2.10). Dla relacji redukcji w rachunku $\lambda^{\to}$ prawdziwe jest także twierdzenie Churcha-Rossera(2.13).

Definicja 2.23. *Reguły wnioskowania w rachunku* $\lambda^{\to}$*.*

$$\frac{}{\Gamma, x : \sigma \triangleright x : \sigma}\ \mathrm{Ass}$$

$$\frac{\Gamma \triangleright M : \sigma}{\Gamma \triangleright M = M : \sigma}\ \mathrm{Refl}_1 \qquad \frac{\Gamma \triangleright M = M : \sigma}{\Gamma \triangleright M : \sigma}\ \mathrm{Refl}_2$$

$$\frac{\Gamma \triangleright M = N : \sigma}{\Gamma \triangleright N = M : \sigma}\ \mathrm{Sym} \qquad \frac{\Gamma \triangleright K = N : \sigma \qquad \Gamma \triangleright N = M : \sigma}{\Gamma \triangleright K = M : \sigma}\ \mathrm{Trans}$$

$$\frac{\Gamma \triangleright K = L : \sigma \to \tau \qquad \Gamma \triangleright N = M : \sigma}{\Gamma \triangleright KN = LM : \tau}\ \mathrm{MonApp}$$

$$\frac{\Gamma, x : \sigma \triangleright N = M : \tau}{\Gamma \triangleright \lambda x.N = \lambda x.M : \sigma \to \tau}\ \mathrm{MonAbs}$$

$$\frac{\Gamma \triangleright (\lambda x.M)N : \sigma}{\Gamma \triangleright (\lambda x.M)N = M[x := N] : \sigma}\ \beta \qquad \frac{\Gamma \triangleright \lambda x.Mx : \sigma}{\Gamma \triangleright \lambda x.Mx = M : \sigma}\ \eta \quad \text{gdzie } x \notin FV(M)$$

$$\frac{\Gamma \triangleright N = M : \sigma}{\Gamma[X := \tau] \triangleright N = M : \sigma[X := \tau]}\ \mathrm{Inst}$$

W rachunku $\lambda^{\to}$ prawdziwe jest następujące twierdzenie.

Twierdzenie 2.24 (Twierdzenie o mocnej normalizacji)**.** *Jeśli $\Gamma \triangleright M : \sigma$ to term M jest mocno normalizowalny.*

Z twierdzenia o mocnej normalizacji wynika, że każdy term w rachunku $\lambda^{\rightarrow}$ redukuje się do postaci normalnej w skończenie wielu krokach. Innymi słowy każda funkcja przedstawiona w postaci termu, redukuje się do wyniku w skończenie wielu krokach. Z programistycznego punktu widzenia, niemożliwe jest napisanie programu, który się nie kończy.

2.2.1. Rozstrzygalność typizowania

Zarówno rachunek λ, jak i rachunek $\lambda^{\rightarrow}$, umożliwia wyrażanie funkcji (algorytmów) w postaci termów. Relacja redukcji definiuje sposób wyliczania wartości funkcji (wykonywania algorytmów). Oba systemy mogą być użyte jako języki programowania. Język programowania, poza możliwością wyrażenia odpowiednio dużej klasy algorytmów, powinien ułatwiać kontrolę poprawności wprowadzanych programów.

W rachunku λ programista nie ma żadnej możliwości kontroli, czy wprowadzony przez niego term odpowiada jego intencjom. Każdy term poprawny składniowo reprezentuje pewną funkcję częściową. W szczególności nie ma żadnej kontroli nad ilością i rodzajem argumentów przekazywanych do funkcji. System nie generuje żadnych dodatkowych informacji, które mogłyby posłużyć do weryfikacji poprawności wprowadzonego programu.

W rachunku $\lambda^{\rightarrow}$ zbiór termów jest podzbiorem zbioru termów rachunku λ. Termami w rachunku $\lambda^{\rightarrow}$ są te termy z rachunku λ, dla których można wyprowadzić typ, co gwarantuje całkowitość funkcji. Typ jest informacją o rodzaju argumentów i rodzaju wyniku funkcji. Kontrola typów uniemożliwia zaaplikowanie funkcji do niewłaściwych argumentów. Programista definiując nową funkcję, wie jakiego typu powinny być jej argumenty i jakiego typu powinien być wynik. Dlatego warunkiem wykorzystania systemu typów jako języka programowania jest możliwość automatycznego sprawdzania typu programu, tzn. czy wprowadzony term ma zamierzony przez programistę typ.

Pożądaną cechą systemu jest możliwość automatycznego wyprowadzania typu, tzn. automatyczne znalezienie typu programu. Zwalnia to programistę z obowiązku jawnego podawania typu, pozostawiając możliwość porównania automatycznie wyprowadzonego typu z zamierzonym. Kolejną pożądaną cechą jest istnienie najogólniejszego typu. Polimorfizm powoduje, że term w rachunku $\lambda^{\rightarrow}$ może mieć więcej, niż jeden typ. Przykładem jest funkcja identyczności $\lambda x.x$, która może być typu np. $V \rightarrow V$, $\xi \rightarrow \xi$, $(\sigma \rightarrow \sigma) \rightarrow (\sigma \rightarrow \sigma)$. Wszystkie typy termu mogą być uzyskane z typu najogólniejszego (jeśli istnieje) za pomocą podstawienia. W rozważanym przykładzie najogólniejszym typem jest $V \rightarrow V$, a odpowiednimi podstawieniami kolejno $(V \rightarrow V)[V := \xi]$ i $(V \rightarrow V)[V := (\sigma \rightarrow \sigma)]$. Nieistnienie najogólniejszego typu może wymusić wielokrotne zdefiniowanie tej samej funkcji, za każdym razem z innym typem.

Możliwość użycia systemu jako języka programowania charakteryzują odpowiedzi na następujące problemy:

Problem sprawdzania typu. Czy dla danych M i σ istnieje Γ takie, że $\Gamma \triangleright M : \sigma$?

Problem wyprowadzania typu. Czy dla danego M istnieje Γ i σ takie, że $\Gamma \triangleright M : \sigma$?

Problem niepustości typu. Czy dla danego σ istnieje Γ i M takie, że $\Gamma \triangleright M : \sigma$?

Problem najogólniejszego typu. Czy dla każdego termu M istnieje Γ i σ takie, że $\Gamma \triangleright M : \sigma$ i σ jest najogólniejszym typem, tzn. dla każdego Δ, $\mathrm{dom}(\Delta) = \mathrm{dom}(\Gamma)$ i τ takich, że $\Delta \triangleright M : \tau$ istnieje takie podstawienie $[\vec{X} := \vec{\xi}]$, że $\Delta = \Gamma[\vec{X} := \vec{\xi}]$ i $\tau = \sigma[\vec{X} := \vec{\xi}]$?

Istnienie w powyższych problemach oznacza nie tylko istnienie danej cechy, ale także istnienie algorytmu który ją znajduje (tzn. znajduje odpowiednie środowisko typizujące, term lub typ).

Dla rachunku $\lambda^{\rightarrow}$ wszystkie te problemy są rozstrzygalne (odpowiedzi na powyższe pytania są twierdzące).

Dodatkowe informacje dotyczące rachunku $\lambda^{\rightarrow}$ znajdują się w [2] i [13].

2.3. System T Gödla

W rachunku $\lambda^{\rightarrow}$ można wyrazić znacznie mniejszą liczbę funkcji niż w rachunku λ. Rozszerzeniem klasy funkcji wyrażalnych z zachowaniem silnej normalizacji termów jest system T Gödla. W systemie T zdefiniowane są dodatkowe typy[7] — liczby naturalne NAT i wartości boolowskie BOOL.

Definicja rachunku $\lambda^{\rightarrow}$ zostaje rozszerzona w następujący sposób (w nawiasie podane są interpretacje typów i termów).

- Zbiór wyrażeń typowych $\mathcal{T}$ zostaje rozszerzony o stałe typowe NAT (liczby naturalne) i BOOL (typ boolowski);
- Zbiór quasitermów Λ zostaje powiększony o stałe O (zero), Suc (następnik), Rec (rekursor dla liczb naturalnych), True, False i If;
- Do reguł wnioskowania należy dołączyć reguły:
  dla typu NAT

$$\frac{}{\Gamma \rhd \mathsf{O} : \mathsf{NAT}}\ \mathsf{NATi}_1 \qquad \frac{\Gamma \rhd x : \mathsf{NAT}}{\Gamma \rhd \mathsf{Suc}\ x : \mathsf{NAT}}\ \mathsf{NATi}_2$$

$$\frac{\Gamma \rhd x : \mathsf{NAT} \qquad \Gamma \rhd y : \sigma \qquad \Gamma \rhd z : \sigma \to (\mathsf{NAT} \to \sigma)}{\Gamma \rhd \mathsf{Rec}\ x\ y\ z : \sigma}\ \mathsf{NATe}$$

$$\frac{\Gamma \rhd \mathsf{Rec}\ \mathsf{O}\ y\ z : \sigma}{\Gamma \rhd \mathsf{Rec}\ \mathsf{O}\ y\ z = y : \sigma}\ \mathsf{NAT}{=}_1 \qquad \frac{\Gamma \rhd \mathsf{Rec}\ (\mathsf{S}\ x)\ y\ z : \sigma}{\Gamma \rhd \mathsf{Rec}\ (\mathsf{S}\ x)\ y\ z = z\ (\mathsf{Rec}\ x\ y\ z)\ x : \sigma}\ \mathsf{NAT}{=}_2$$

  i dla typu BOOL

$$\frac{}{\Gamma \rhd \mathsf{True} : \mathsf{BOOL}}\ \mathsf{BOOLi}_1 \qquad \frac{}{\Gamma \rhd \mathsf{False} : \mathsf{BOOL}}\ \mathsf{BOOLi}_1$$

$$\frac{\Gamma \rhd x : \mathsf{BOOL} \qquad \Gamma \rhd y : \sigma \qquad \Gamma \rhd z : \sigma}{\Gamma \rhd \mathsf{If}\ x\ y\ z : \sigma}\ \mathsf{BOOLe}$$

$$\frac{\Gamma \rhd \mathsf{If}\ \mathsf{True}\ y\ z : \sigma}{\Gamma \rhd \mathsf{If}\ \mathsf{True}\ y\ z = y : \sigma}\ \mathsf{BOOL}{=}_1 \qquad \frac{\Gamma \rhd \mathsf{If}\ \mathsf{False}\ y\ z : \sigma}{\Gamma \rhd \mathsf{If}\ \mathsf{False}\ y\ z = z : \sigma}\ \mathsf{BOOL}{=}_2$$

- Relację redukcji rozszerza się w naturalny sposób (zgodnie z regułami $\mathsf{NAT}{=}_1$,$\mathsf{NAT}{=}_2$,$\mathsf{BOOL}{=}_1$ i $\mathsf{BOOL}{=}_2$).

Twierdzenie 2.25. *Termy w systemie T są mocno normalizowalne.*

W systemie T nie ma możliwości wprowadzania do systemu nowych typów. Bardziej szczegółowe omówienie systemu T znajduje się w [5].

[7] W części literatury system T jest przedstawiony jako rachunkiem $\lambda^{\rightarrow}$ rozszerzony tylko o liczby naturalne. Nie zmienia to własności systemu, ponieważ wartości boolowskie można wyrazić przy pomocy liczb naturalnych

2.4. System F Girarda

System F wprowadzony został niezależnie przez Girarda w 1972 r. i Reynoldsa w 1974 r. System F jest rozszerzeniem rachunku $\lambda^{\to}$ o abstrakcję dla typów. W rachunku $\lambda^{\to}$ funkcja identyczności $\lambda x.x$ jest typu $X \to X$, a w systemie F ta funkcja jest typu $\forall X.X \to X$. Znak $\forall$ pełni podobną rolę, jak λ dla termów, tzn. funkcja identyczności może być typu $Y \to Y$ dla każdego typu Y.

System F nazywany jest także rachunkiem λ drugiego rzędu lub polimorficznym rachunkiem λ i oznaczany $\lambda 2$.

Poniżej przedstawione są najważniejsze definicje. Pominięto fragmenty, które są identyczne w rachunku $\lambda^{\to}$.

Definicja 2.26. Zbiór wyrażeń typowych $\mathcal{T}$ *definiuje się za pomocą przeliczalnego zbioru zmiennych typowych* $\mathcal{V}_{\mathcal{T}} = \{V, V', V'', \ldots\}$ *,konstruktorów* $\to$ *i* $\forall$ *następująco*

$$\begin{aligned} \mathcal{V}_{\mathcal{T}} &::= V \mid \mathcal{V}_{\mathcal{T}}' \\ \mathcal{T} &::= \mathcal{V}_{\mathcal{T}} \mid \mathcal{T} \to \mathcal{T} \mid (\forall \mathcal{V}_{\mathcal{T}}.\mathcal{T}) \end{aligned}$$

Stosowana jest następująca notacja.

***Notacja* 2.27.**

1. $\forall X_1 \ldots X_n.\sigma$ oznacza $(\forall X_1(\ldots(\forall X_n.\sigma)\ldots))$.
2. $\forall$ wiąże mocniej niż $\to$.
3. Zbiór zmiennych wolnych w typie V oznacza się przez $FTV(V)$, a zbiór zmiennych związanych przez $BTV(V)$.

Definicja 2.28.

- Zbiór quasitermów Λ_{T} *definiuje się przy pomocy przeliczalnego zboru zmiennych V następująco:*

$$\begin{aligned} \mathcal{V} &::= v \mid \mathcal{V}' \\ \Lambda_{\mathrm{T}} &::= \mathcal{V} \mid (\Lambda_{\mathrm{T}}\Lambda_{\mathrm{T}}) \mid (\lambda \mathcal{V} : \mathcal{T}.\Lambda_{\mathrm{T}}) \mid (\Lambda \mathcal{V}_{\mathcal{T}}.\Lambda_{\mathrm{T}}) \mid (\Lambda_{\mathrm{T}}\mathcal{T}) \end{aligned}$$

- Termem w systemie F *jest quasiterm M, dla którego można wyprowadzić typ, tzn. istnieje takie* Γ *i* σ*, że przy pomocy reguł zdefiniowanych w (2.29) można wyprowadzić* $\Gamma \rhd M : \sigma$*.*

Definicja 2.29. *Reguły typizowania w systemie F.*

$$\frac{}{\Gamma, x : \sigma \rhd x : \sigma}\ \mathrm{Ass}$$

$$\frac{\Gamma \rhd M : \sigma \to \tau \qquad \Gamma \rhd N : \sigma}{\Gamma \rhd MN : \tau}\ {\to}\mathtt{e} \qquad\qquad \frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma.M : \sigma \to \tau}\ {\to}\mathtt{i}$$

$$\frac{\Gamma \rhd M : \forall X.\sigma}{\Gamma \rhd M\tau : \sigma[X := \tau]}\ \forall\mathtt{e} \qquad\qquad \frac{\Gamma \rhd M : \sigma}{\Gamma \rhd \Lambda X.M : (\forall X.\sigma)}\ \forall\mathtt{i} \quad (X \notin FTV(\Gamma))$$

Definicja 2.30. *Reguły wnioskowania w systemie F.*

$$\frac{}{\Gamma, x : \sigma \rhd x : \sigma}\ \text{Ass}$$

$$\frac{\Gamma \rhd M : \sigma}{\Gamma \rhd M = M : \sigma}\ \text{Refl}_1 \qquad \frac{\Gamma \rhd M = M : \sigma}{\Gamma \rhd M : \sigma}\ \text{Refl}_2$$

$$\frac{\Gamma \rhd M = N : \sigma}{\Gamma \rhd N = M : \sigma}\ \text{Sym} \qquad \frac{\Gamma \rhd K = N : \sigma \qquad \Gamma \rhd N = M : \sigma}{\Gamma \rhd K = M : \sigma}\ \text{Trans}$$

$$\frac{\Gamma \rhd K = L : \sigma \to \tau \qquad \Gamma \rhd N = M : \sigma}{\Gamma \rhd KN = LM : \tau}\ \text{MonApp}$$

$$\frac{\Gamma, x : \sigma \rhd N = M : \tau}{\Gamma \rhd \lambda x : \sigma.N = \lambda x : \sigma.M : \sigma \to \tau}\ \text{MonAbs}$$

$$\frac{\Gamma \rhd N = M : \forall X.\sigma}{\Gamma \rhd N\tau = M\tau : \sigma[X := \tau]}\ \text{MonApp}^2$$

$$\frac{\Gamma, x : \sigma \rhd N = M : \sigma}{\Gamma \rhd \Lambda X.N = \Lambda X.M : (\forall X.\sigma)}\ \text{MonAbs}^2 \quad (X \notin FTV(\Gamma))$$

$$\frac{\Gamma \rhd (\lambda x : \tau.M)N : \sigma}{\Gamma \rhd (\lambda x : \tau.M)N = M[x := N] : \sigma}\ \beta \qquad \frac{\Gamma \rhd \lambda x : \tau.Mx : \sigma}{\Gamma \rhd \lambda x : \tau.Mx = M : \sigma}\ \eta \quad \text{gdzie } x \notin FV(M)$$

$$\frac{\Gamma \rhd (\Lambda X.M)\tau : \sigma}{\Gamma \rhd (\Lambda X.M)\tau = M[X := \tau] : \sigma}\ \beta^2 \qquad \frac{\Gamma \rhd \Lambda X.MX : \sigma}{\Gamma \rhd \Lambda X.MX = M : \sigma}\ \eta^2 \quad \text{gdzie } x \notin FTV(M)$$

Twierdzenie 2.31. *W systemie F termy są mocno normalizowalne.*

System F jest wystarczająco bogaty by wyrazić nie tylko funkcje wyrażalne w rachunku $\lambda^{\to}$ (co jest oczywiste) i w systemie T (wystarczy wyrazić typy i termy podane w definicji systemu T). System F pozwala na definiowanie nowych typów (np. liczb naturalnych). Niech przykładem definicji typu danych w systemie F będzie BOOL.

***Przykład* 2.32.** Definicja typu BOOL w systemie F.

- Typ BOOL jest reprezentowany przez wyrażenie $\forall X.(X \to X \to X)$.
- Termy True, False i If definiowane są następująco:

$$\begin{aligned} \mathsf{True} &= \Lambda X.\lambda y : X.\lambda z : X.y \\ \mathsf{False} &= \Lambda X.\lambda y : X.\lambda z : X.z \\ \mathsf{If} &= \lambda x : \mathsf{BOOL}.\lambda y : V.\lambda z : V.x\ V\ y\ z \end{aligned}$$

Dla tak zdefiniowanego typu BOOL można wyprowadzić reguły zdefiniowane w podrozdziale

2.3, np. regułę $\texttt{BOOLi}_1$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, z : X, y : X \rhd y : X}\text{Ass}}{\Gamma, z : X, y : X \rhd y : X = y : X}\text{Refl}_1}{\Gamma, y : X \rhd \lambda z : X.y = \lambda z : X.y : X \to X}\text{MonAbs}}{\Gamma \rhd \lambda y : X.\lambda z : X.y = \lambda y : X.\lambda z : X.y : X \to X \to X}\text{MonAbs}}{\Gamma \rhd \Lambda X.\lambda y : X.\lambda z : X.y = \Lambda X.\lambda y : X.\lambda z : X.y : \forall X.(X \to X \to X)}\text{MonAbs}^2}{\Gamma \rhd \Lambda X.\lambda y : X.\lambda z : X.y : \forall X.(X \to X \to X)}\text{Refl}_2}{\Gamma \rhd \mathsf{True} : \mathsf{BOOL}}\textit{po podstawieniu definicji}$$

Siła wyrażania systemu F jest wystarczająca, by wyrazić w nim dużą klasę funkcji. W wielu przypadkach trudność polega na znalezieniu termu reprezentującego zadaną funkcję i spełniającego dodatkowe wymogi, np. dotyczące złożoności obliczeniowej lub możliwości dowiedzenia pewnych własności. W systemie F istnieje wiele sposobów reprezentowania liczb naturalnych. Możliwe jest udowodnienie równości $x = \mathsf{pred}(\mathsf{Suc}\ x)$[8] dla każdego ustalonego x, a więc można sprawdzić, że np. $4 = \mathsf{pred}(\mathsf{Suc}\ 5)$. Nie jest jednak znana taka reprezentacja liczb naturalnych i poprzednika, by można było udowodnić tę równość w systemie F dla x będącego zmienną (a więc dla wszystkich x).

Typizowanie w systemie F jest nierozstrzygalne. Udowodniono, że nie istnieją algorytmy rozwiązujące problemy zdefiniowane w podrozdziale 2.2.1. Programista używający systemu F musiałby wprowadzać termy wraz z typami, które w przypadku bardziej złożonych programów są skomplikowane. Wystarczy spojrzeć na term If, który , mimo że reprezentuje bardzo prostą funkcję, jest już dość zawiły $\lambda x : (\forall X.(X \to X \to X)).\lambda y : V.\lambda z : V.x\ V\ y\ z$.

Szczegółowy opis systemu F znajduje się w [2] i [13].

2.5. Logika intuicjonistyczna

Logika intuicjonistyczna jest jedną z logik konstruktywnych. Łatwo intuicyjnie dostrzec różnicę w stosunku do logiki klasycznej rozważając następujące twierdzenie.

Twierdzenie 2.33. *Istnieją dwie niewymierne liczby a i b takie, że a^b jest wymierne.*

Dowód. Albo $(\sqrt{2})^{\sqrt{2}}$ jest wymierne i wtedy $a = b = \sqrt{2}$ albo $(\sqrt{2})^{\sqrt{2}}$ jest niewymierne i wtedy $a = (\sqrt{2})^{\sqrt{2}}, b = \sqrt{2}$. □

Dowód tego twierdzenie jest oczywiście poprawny w logice klasycznej. Jest to przykład dowodu niekonstruktywnego, ze względu na wykorzystanie reguły wykluczonego środka. Pomimo, że twierdzenie jest dowiedzione i wiadomo, że liczby a i b istnieją, to nie można (na podstawie dowodu) znaleźć pary liczb spełniających warunek określony w twierdzeniu. Konstruktywny dowód tego twierdzenia jest oparty na twierdzeniu Gelfonda–Schneidera[7]: jeśli $a \notin 0, 1$ jest liczbą algebraiczną, a b niewymierną liczbą algebraiczną, to liczba a^b jest przestępna. Z twierdzenia Gelfonda–Schneidera wynika, że liczby $a = b = \sqrt{2}$ spełniają twierdzenie 2.33.

Logika intuicjonistyczna nie dopuszcza dowodów niekonstruktywnych. Znaczenie formuły logicznej wyjaśnia się nie za pomocą prawdy i fałszu, ale za pomocą pojęcia dowodu. Prawdziwe są te formuły, które zostały udowodnione. Z tego powodu formuła $A \vee \neg A$ nie jest tautologią w logice intuicjonistycznej. W logice klasycznej prawdziwość tego zdania oparta jest na założeniu, że formuła A jest albo prawdziwa, albo nieprawdziwa. W logice intuicjonistycznej zdanie $A \vee \neg A$ jest prawdziwe tylko wtedy, gdy istnieje dowód A lub dowód $\neg A$.

[8] pred oznacza poprzednik.

Operatory logiczne interpretowane są według interpretacji Bouwera–Heytinga–Kołmogorowa (interpretacji BHK). Interpretacja BHK wyraża dowody formuł zbudowanych za pomocą operatorów logicznych przez dowody ich składników. A,B oznacza formuły, a X zbiór obiektów, który przebiegają zmienne.

1. Dowodem $A \wedge B$ jest przedstawienie dowodu A i dowodu B.
2. Dowodem $A \vee B$ jest przedstawienie dowodu A lub dowodu B i wskazanie, który dowód został przedstawiony.
3. Dowodem $A \to B$ jest konstrukcja pozwalająca przekształcić dowód A w dowód B.
4. Absurd $\perp$ nie posiada dowodu.
5. Dowodem $\neg A$ jest dowód $A \to \perp$.
6. Dowodem $\forall x.A(x)$ jest konstrukcja pozwalająca przekształcić dowolny $y \in X$ w dowód $A(y)$.
7. Dowodem $\exists x.A(x)$ jest przedstawienie pewnego $y \in X$ i dowodu $A(y)$.

Z punktu 7 wynika, że dowód twierdzenia(2.33) w logice intuicjonistycznej musi zawierać choć jedną parę liczb a i b, o których wiadomo, że są niewymierne i że liczba a^b jest wymierna.

Poniżej przedstawiono kilka przykładów formuł prawdziwych w logice klasycznej, których nie można udowodnić w logice intuicjonistycznej.

prawdziwe w logice klasycznej	prawdziwe w logice intuicjonistycznej	nieprawdziwe w logice intuicjonistycznej
$A \vee \neg A$	$\neg\neg(A \vee \neg A)$	$A \vee \neg A$
$\neg\neg A \Leftrightarrow A$	$A \Rightarrow \neg\neg A$	$\neg\neg A \Rightarrow A$
$\neg A \vee B \Leftrightarrow A \to B$	$\neg A \vee B \Rightarrow A \to B$	$A \to B \Rightarrow \neg A \vee B$
$\neg A \to \neg B \Leftrightarrow B \to A$	$B \to A \Rightarrow \neg A \to \neg B$	$\neg A \to \neg B \Rightarrow B \to A$

Logika intuicjonistyczna omówiona jest w [12].

2.6. System dedukcji naturalnej

System dedukcji naturalnej został wprowadzony przez Gentzena w 1932 r. i rozwinięty przez Prawitza w latach 60-tych. System składa się z reguł wnioskowania opisanych przez schemat

$$\frac{H_1 \quad H_2 \quad \dots \quad H_n}{C}\,\mathrm{R}$$

gdzie H_i oznaczają przesłanki (może ich być zero lub więcej) a C wniosek (zawsze jeden) który możemy wysnuć przy użyciu reguły R. Dla każdego operatora logicznego $\diamond$ zdefiniowane są reguły wprowadzania $\diamond$`i` (opisujące jakie założenia są potrzebne żeby wysnuć wniosek zawierający $\diamond$) i eliminacji $\diamond$`e` (opisujące wniosek jaki możemy wysnuć z założenia zawierającego $\diamond$). Dowody reprezentowane są przez drzewa wywodu. W liściach mogą znajdować się założenia lub udowodnione wcześniej formuły. Nowe formuły otrzymuje się przez zastosowanie jednej z reguł wnioskowania. Każda formuła zależy od otwartych założeń znajdujących się nad nią w drzewie wywodu. Założenia mogą być zamykane przez reguły $\vee$`e`, $\to$`i`, $\neg$`i` i $\exists$`e`. Zamykane założenia oznaczane są przez $[A]^n$, a (n) dopisywane jest do nazwy reguły, przy pomocy której założenia zamknięto. n jest znacznikiem, np. liczbą naturalną lub literą. Dozwolone jest zamykanie nieistniejących założeń. Formuła zostaje udowodniona jeśli występuje w korzeniu drzewa wywodu i wszystkie założenia są zamknięte. Zapis

$$\begin{array}{c} A \\ \vdots\,\mathcal{D} \\ B \end{array}$$

oznacza, że przy założeniu A można dowieść B przy pomocy drzewa wywodu oznaczonego przez $\mathcal{D}$.

Poniżej zostaną przedstawione reguły wnioskowania dla logiki intuicjonistycznej.

Definicja 2.34. Reguły wnioskowania dla rachunku zdań:

Reguły wprowadzania

$$\frac{A \qquad B}{A \wedge B}\,\wedge\mathtt{i}$$

$$\frac{A}{A \vee B}\,\vee\mathtt{i}_1 \qquad \frac{B}{A \vee B}\,\vee\mathtt{i}_2$$

$$\frac{\begin{array}{c} [A]^n \\ \vdots \\ B \end{array}}{A \to B}\,\to\mathtt{i}\ (n)$$

Reguły eliminacji

$$\frac{A \wedge B}{A}\,\wedge\mathtt{e}_1 \qquad \frac{A \wedge B}{B}\,\wedge\mathtt{e}_2$$

$$\frac{A \vee B \qquad \begin{array}{c} [A]^n \\ \vdots \\ C \end{array} \qquad \begin{array}{c} [B]^n \\ \vdots \\ C \end{array}}{C}\,\vee\mathtt{e}\ (n)$$

$$\frac{A \to B \qquad A}{B}\,\to\mathtt{e}$$

$$\frac{\perp}{A}\,\perp\mathtt{e}$$

Reguły wnioskowania dla rachunku predykatów:

$$\frac{A}{\forall x.A}\,\forall\mathtt{i}$$

o ile x nie jest zmienną wolną w otwartych założeniach od których zależy A

$$\frac{\forall x.A}{A[t := x]}\,\forall\mathtt{e}$$

$$\frac{A[t := x]}{\exists x.A}\,\exists\mathtt{i}$$

$$\frac{\exists x.A \qquad \begin{array}{c} [A] \\ \vdots \\ B \end{array}}{B}\,\exists\mathtt{e}$$

x może być zmienną wolną tylko w założeniu zamykanym przez tę regułę

Symbol $\perp$ oznacza „absurd” i nie istnieje dla niego reguła wprowadzania. Operator negacji $\neg A$ można zdefiniować jako $A\to\perp$ co jest równoważne następującym regułom (za B w $\to\mathtt{i}$ i $\to\mathtt{e}$ należy podstawić $\perp$).

$$\frac{\begin{array}{c} [A]^n \\ \vdots \\ \perp \end{array}}{\neg A}\,\neg\mathtt{i}\ (n) \qquad \frac{\neg A \qquad A}{\perp}\,\neg\mathtt{e}$$

Poniżej przedstawiony jest dowód prawdziwości zdania $(\sigma \to \tau \to \xi) \to (\sigma \to \tau) \to (\sigma \to \xi)$.

***Przykład* 2.35.**

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\sigma\to\tau\to\xi]^1 \qquad [\sigma]^2}{\tau\to\xi}\to\mathtt{e} \qquad [\tau]^4}{\xi}\to\mathtt{e}}{\tau\to\xi}\to\mathtt{i}\ (4) \qquad \dfrac{[\sigma\to\tau]^3 \qquad [\sigma]^2}{\tau}\to\mathtt{e}}{\xi}\to\mathtt{e}}{\sigma\to\xi}\to\mathtt{i}\ (2)}{(\sigma\to\tau)\to(\sigma\to\xi)}\to\mathtt{i}\ (3)}{(\sigma\to\tau\to\xi)\to(\sigma\to\tau)\to(\sigma\to\xi)}\to\mathtt{i}\ (1)$$

Aby uzyskać system równoważny logice klasycznej trzeba dołożyć regułę podwójnego zaprzeczenia

$$\dfrac{\neg\neg A}{A}\,\neg\neg$$

Należy podkreślić, że nie jest to ani reguła wprowadzania ani eliminacji. Jej dodanie powoduje możliwość dodatkowej konwersji formuł logicznych, co w tym systemie jest niezbędne do udowodnienia wielu naturalnych dla logiki klasycznej formuł (np. zasady wyłączonego środka, czy właśnie podwójnego zaprzeczenia). Niech za przykład posłuży dowód prawdziwości zdania $A \vee \neg A$.

$$\dfrac{\dfrac{\dfrac{[\neg(A\vee\neg A)]^2 \qquad \dfrac{\dfrac{\dfrac{[\neg(A\vee\neg A)]^2 \qquad \dfrac{[A]^1}{A\vee\neg A}\vee\mathtt{i}_1}{\bot}\neg\mathtt{e}}{\neg A}\neg\mathtt{i}\ (1)}{A\vee\neg A}\vee\mathtt{i}_2}{\bot}\neg\mathtt{e}}{\neg\neg(A\vee\neg A)}\neg\mathtt{i}\ (2)}{A\vee\neg A}\neg\neg$$

Wszystkie założenia zostały zamknięte (przy pomocy liczb oznaczono reguły zamykające odpowiednie założenia) więc twierdzenie zostało dowiedzione. W dowodzie istotną rolę spełnia reguła podwójnego zaprzeczenia. Bez niej udowodnienie tego twierdzenia nie jest możliwe. Zamiast reguły podwójnego zaprzeczenia można do systemu dołożyć regułę opowiadającą zasadzie wyłączonego środka

$$\dfrac{}{A\vee\neg A}\,\text{zasada wyłączonego środka}$$

Regułę podwójnego zaprzeczenia w systemie z zasadą wyłączonego środka można wyprowadzić (obie reguły są równoważne).

Dodatkowe informacje na temat systemu dedukcji naturalnej znajdują się w [9] i [12].

2.7. Normalizacja dowodów

System dedukcji naturalnej umożliwia operowanie na strukturze dowodu. Dowód zdania $A \to B \to A$ może wyglądać następująco

$$\dfrac{\dfrac{\dfrac{\dfrac{[A]^1 \qquad [B]^2}{A\wedge B}\wedge\mathtt{i}}{A}\wedge\mathtt{e}_1}{B\to A}\to\mathtt{i}\ (2)}{A\to B\to A}\to\mathtt{i}\ (1)$$

Zastosowanie reguły wprowadzenia i następującej bezpośrednio po niej reguły eliminacji operatora $\wedge$ nie jest konieczne. Prostszy dowód tego samego zdania nie zawiera $\wedge$.

$$\dfrac{\dfrac{[A]^1}{B \to A}\to\mathtt{i}}{A \to B \to A}\to\mathtt{i}\ (1)$$

Drugi dowód powstał z pierwszego przez redukcję reguł $\wedge\mathtt{i}$ i $\wedge\mathtt{e}_1$.

Podobne zasady redukcji zdefiniować można dla każdego operatora (dla reguły wprowadzania i występującej bezpośrednio pod nią reguły eliminacji).

Definicja 2.36. Relację redukcji w jednym kroku $\rightsquigarrow$ *dla systemu dedukcji naturalnej definiuje się następująco:*

$$\dfrac{\dfrac{\overset{\vdots\,\mathcal{D}_1}{A} \qquad \overset{\vdots\,\mathcal{D}_2}{B}}{A \wedge B}\wedge\mathtt{i}}{A}\wedge\mathtt{e}_1 \quad\rightsquigarrow\quad \overset{\vdots\,\mathcal{D}_1}{A} \qquad\qquad \dfrac{\dfrac{\overset{\vdots\,\mathcal{D}_1}{A} \qquad \overset{\vdots\,\mathcal{D}_2}{B}}{A \wedge B}\wedge\mathtt{i}}{B}\wedge\mathtt{e}_2 \quad\rightsquigarrow\quad \overset{\vdots\,\mathcal{D}_2}{B}$$

$$\dfrac{\dfrac{\overset{\vdots\,\mathcal{D}_1}{A}}{A \vee B}\vee\mathtt{i}_1 \qquad \overset{[A]^n}{\overset{\vdots\,\mathcal{D}_2}{C}} \qquad \overset{[B]^n}{\overset{\vdots\,\mathcal{D}_3}{C}}}{C}\vee\mathtt{e} \quad\rightsquigarrow\quad \overset{\vdots\,\mathcal{D}_1}{\overset{A}{\overset{\vdots\,\mathcal{D}_2}{C}}}$$

$$\dfrac{\dfrac{\overset{\vdots\,\mathcal{D}_1}{B}}{A \vee B}\vee\mathtt{i}_2 \qquad \overset{[A]^n}{\overset{\vdots\,\mathcal{D}_2}{C}} \qquad \overset{[B]^n}{\overset{\vdots\,\mathcal{D}_3}{C}}}{C}\vee\mathtt{e} \quad\rightsquigarrow\quad \overset{\vdots\,\mathcal{D}_1}{\overset{B}{\overset{\vdots\,\mathcal{D}_3}{C}}}$$

$$\dfrac{\dfrac{\overset{[A]^n}{\overset{\vdots\,\mathcal{D}_1}{B}}}{A \to B}\to\mathtt{i}\ (n) \qquad \overset{\vdots\,\mathcal{D}_2}{A}}{B}\to\mathtt{e} \quad\rightsquigarrow\quad \overset{\vdots\,\mathcal{D}_2}{\overset{A}{\overset{\vdots\,\mathcal{D}_1}{B}}}$$

$$\dfrac{\dfrac{\overset{\vdots\,\mathcal{D}_1}{A}}{\forall x.A}\forall\mathtt{i}}{A[t/x]}\forall\mathtt{e} \quad\rightsquigarrow\quad \overset{\vdots\,\mathcal{D}_1[t/x]}{A[t/x]}$$

$$\dfrac{\dfrac{\overset{\vdots\,\mathcal{D}_1}{A[t/x]}}{\exists x.A}\exists\mathtt{i} \qquad \overset{[A]^n}{\overset{\vdots\,\mathcal{D}_2}{C}}}{C}\exists\mathtt{e}\ (n) \quad\rightsquigarrow\quad \overset{\vdots\,\mathcal{D}_1}{\overset{A[t/x]}{\overset{\vdots\,\mathcal{D}_2}{C}}}$$

Relacja redukcji *jest domknięciem zwrotnym i przechodnim relacji redukcji w jednym kroku.*

Twierdzenie 2.37. *Relacja redukcji w systemie dedukcji naturalnej spełnia twierdzenie Churcha–Rossera(2.13). Dowody w systemie dedukcji naturalnej są mocno normalizowalne.*

Z powyższego twierdzenia wynika, że każdy dowód posiada jedyną postać normalną. Dowody, które nie są w postaci normalnej, zawierają konstrukcje, które są zbędne. Zastosowanie relacji redukcji pozwala na doprowadzenie dowodu do najprostszej postaci przez operacje na jego strukturze.

Dowód przedstawiony w przykładzie 2.35 nie jest w postaci normalnej (zawiera następujące po sobie reguły $\to$`i` i $\to$`e`). Zastosowanie relacji redukcji prowadzi do następującego dowodu.

***Przykład* 2.38.** Postać normalna dowodu prawdziwości zdania $(\sigma \to \tau \to \xi) \to (\sigma \to \tau) \to (\sigma \to \xi)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\sigma \to \tau \to \xi]^1 \qquad [\sigma]^2}{\tau \to \xi}\to\mathtt{e} \qquad \dfrac{[\sigma \to \tau]^3 \qquad [\sigma]^2}{\tau}\to\mathtt{e}}{\xi}\to\mathtt{e}}{\sigma \to \xi}\to\mathtt{i}\ (2)}{(\sigma \to \tau) \to (\sigma \to \xi)}\to\mathtt{i}\ (3)}{(\sigma \to \tau \to \xi) \to (\sigma \to \tau) \to (\sigma \to \xi)}\to\mathtt{i}\ (1)$$

Normalizacja dowodów w systemie dedukcji naturalnej jest omówiona w [5] i [12].

2.8. Izomorfizm Curry'ego-Howarda

W 1958r. Curry zauważył związek pomiędzy logiką i rachunkiem $\lambda^{\to}$. W 1969r. Howard rozszerzył jego wnioski na rachunek predykatów w systemie dedukcji naturalnej i rozszerzenie rachunku $\lambda^{\to}$.

Porównanie reguł wnioskowania w systemie dedukcji naturalnej (definicja 2.34) z regułami typizowania w rachunku $\lambda^{\to}$ (definicja 2.18) prowadzi do wniosku, że odpowiadające sobie reguły po usunięciu informacji o termach są identyczne[9]. W $\lambda^{\to}$ rolę znaczników założeń pełnią zmienne należące do środowiska typizującego.

rachunek $\lambda^{\to}$	system dedukcji naturalnej
$\dfrac{M : \sigma \to \tau \qquad N : \sigma}{MN : \tau}\to\mathtt{e}$	$\dfrac{\sigma\to\tau \qquad \sigma}{\tau}\to\mathtt{e}$
$\dfrac{\begin{matrix}[x : \sigma]\\ \vdots \\ M : \tau\end{matrix}}{\lambda x.M : \sigma \to \tau}\to\mathtt{i}$	$\dfrac{\begin{matrix}[\sigma]^n\\ \vdots \\ \tau\end{matrix}}{\sigma\to\tau}\to\mathtt{i}\ (n)$

Ilustracją izomorfizmu Curry'ego–Howarda jest następujący przykład. $A \to B \to A$ jest formułą w logice intuicjonistycznej, którą można potraktować jako typ w rachunku $\lambda^{\to}$. Zamiast szukania dowodu tej formuły można szukać termu, który będzie miał taki typ. Wystarczy zdefiniować funkcję, która bierze dwa argumenty i zwraca pierwszy z nich — $\lambda xy.x$. Wyprowadzenie typu dla tej funkcji wygląda następująco

$$\dfrac{\dfrac{[x : A]}{\lambda y.x : B \to A}\to\mathtt{i}}{\lambda xy.x : A \to B \to A}\to\mathtt{i}$$

[9] W tym rozdziale reguły typizowania w rachunku $\lambda^{\to}$ przedstawiane będą bez środowiska typizującego. Elementom znajdującym się w środowisku odpowiadają otwarte założenia. Obie postacie reguł są równoważne.

Wystarczy zauważyć, że to wyprowadzenie typu jest identycznie zbudowane jak dowód formuły $A \to B \to A$ znajdujący się na stronie 20.

W przykładzie 2.35 przedstawiony został dowód prawdziwości zdania $(\sigma \to \tau \to \xi) \to (\sigma \to \tau) \to (\sigma \to \xi)$. Na jego podstawie można zbudować term, którego drzewo wywodu typu będzie miało identyczną strukturę, jak dowód formuły. Taki term przedstawiony jest poniżej.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[x:\sigma\to\tau\to\xi] \qquad [z:\sigma]}{xz:\tau\to\xi}\,{\to}\mathtt{e} \qquad [v:\tau]}{xzv:\xi}\,{\to}\mathtt{e}}{\lambda v.xzv:\tau\to\xi}\,{\to}\mathtt{i} \qquad \cfrac{[y:\sigma\to\tau] \qquad [z:\sigma]}{yz:\tau}\,{\to}\mathtt{e}}{(\lambda v.xzv)(yz):\xi}\,{\to}\mathtt{e}}{\lambda z.(\lambda v.xzv)(yz):\sigma\to\xi}\,{\to}\mathtt{i}}{\lambda yz.(\lambda v.xzv)(yz):(\sigma\to\tau)\to(\sigma\to\xi)}\,{\to}\mathtt{i}}{\lambda xyz.(\lambda v.xzv)(yz):(\sigma\to\tau\to\xi)\to(\sigma\to\tau)\to(\sigma\to\xi)}\,{\to}\mathtt{i}$$

Jednocześnie postać termu $\lambda xyz.(\lambda v.xzv)(yz) : (\sigma \to \tau \to \xi) \to (\sigma \to \tau) \to (\sigma \to \xi)$, a właściwie postać drzewa wywodu typu termu, pozwala na zbudowanie dowodu formuły $(\sigma \to \tau \to \xi) \to (\sigma \to \tau) \to (\sigma \to \xi)$.

Poza związkiem między dowodami i termami, oraz typami i formułami istnieje także związek między relacjami redukcji w każdym z tych systemów. Dowód z przykładu 2.35 nie jest w postaci normalnej, podobnie jak odpowiadający mu term. Dowód w postaci normalnej przedstawiony jest w przykładzie 2.38. Term można doprowadzić do postaci normalnej, korzystając z relacji redukcji (podkreślono redukowaną część termu).

$$\lambda xyz.\underline{(\lambda v.xzv)(yz)} \rightsquigarrow \lambda xyz.xz(yz)$$

Term w postaci normalnej odpowiada dowodowi w postaci normalnej. Przekształcenie dowodu w postaci normalnej w term doprowadziłoby do tego samego wyniku (tzn. termu $\lambda xyz.xz(yz)$).

Oczywiście w systemie dedukcji naturalnej poza operatorem $\to$ zdefiniowane są także $\vee$,$\wedge$ i $\bot$. Dołożenie typów i termów odpowiadających tym operatorom do rachunku $\lambda^{\to}$ prowadzi do rachunku $\lambda^{\to,\vee,\wedge,\bot}$ (operatorowi $\vee$ odpowiada suma rozłączna, $\wedge$ — para). Operatory $\vee$ i $\wedge$ wprowadza się następująco.

Definicja 2.39. *Wprowadzenie $\vee$ wymaga zdefiniowania konstruktora typu $\vee$ i termów* Inl,Inr *i* when. *Termy* Inl *i* Inr *są konstruktorami sumy rozłącznej.*

$$\cfrac{M:\sigma}{\mathsf{Inl}\ M:\sigma\vee\tau}\,\vee\mathtt{i}_1 \qquad\qquad \cfrac{N:\tau}{\mathsf{Inr}\ N:\sigma\vee\tau}\,\vee\mathtt{i}_2$$

$$\cfrac{N:\sigma\vee\tau \quad P:\sigma\to\xi \quad Q:\tau\to\xi}{\mathsf{when}\ N\ P\ Q:\xi}\,\vee\mathtt{e}$$

$$\cfrac{\mathsf{Inl}\ N:\sigma\vee\tau \quad P:\sigma\to\xi \quad Q:\tau\to\xi}{\mathsf{when}\ (\mathsf{Inl}\ N)\ P\ Q\ =\ P\ N:\xi}\,\vee{=}_1 \qquad \cfrac{\mathsf{Inr}\ M:\sigma\vee\tau \quad P:\sigma\to\xi \quad Q:\tau\to\xi}{\mathsf{when}\ (\mathsf{Inr}\ M)\ P\ Q\ =\ Q\ M:\xi}\,\vee{=}_1$$

Definicja 2.40. *Wprowadzenie* $\wedge$ *wymaga zdefiniowania konstruktora typu* $\wedge$ *i termów* Pair,fst *i* snd. *Term* Pair *jest konstruktorem pary,* fst *zwraca pierwszy element z pary, a* snd *— drugi.*

$$\dfrac{M : \sigma \qquad N : \tau}{\mathsf{Pair}\ N\ M : \sigma \wedge \tau}\ \wedge\mathtt{i}$$

$$\dfrac{N : \sigma \wedge \tau}{\mathsf{fst}\ N : \sigma}\ \wedge\mathtt{e}_1 \qquad\qquad \dfrac{N : \sigma \wedge \tau}{\mathsf{snd}\ N : \tau}\ \wedge\mathtt{e}_2$$

$$\dfrac{\mathsf{Pair}\ N\ M : \sigma \wedge \tau}{\mathsf{fst}(\mathsf{Pair}\ N\ M) = N : \sigma}\ \wedge{=}_1 \qquad\qquad \dfrac{\mathsf{Pair}\ N\ M : \sigma \wedge \tau}{\mathsf{snd}(\mathsf{Pair}\ N\ M) = N : \tau}\ \wedge{=}_2$$

Podobnie wprowadza się kolejne operatory.

Rachunek $\lambda^{\to,\vee,\wedge,\bot}$ jest izomorficzny z systemem dedukcji naturalnej dla rachunku zdań. W szczególności izomorfizm Curry'ego–Howarda definiuje następujące równoważności.

rachunek $\lambda^{\to,\vee,\wedge,\bot}$	—	system dedukcji naturalnej
σ jest typem	—	σ jest formułą
M jest termem (programem)	—	M jest dowodem
term M jest typu σ	—	M jest dowodem formuły σ
term N redukuje się do M	—	dowód N redukuje się do dowodu M

Kolejnym przykładem obrazującym izomorfizm Curry'ego-Howarda jest dowód formuły $((a \wedge b) \vee (c \wedge d) \to ((a \vee c) \wedge (b \vee d))$. Poniżej przedstawione jest wyprowadzenie typu dla termu, który odpowiada dowodowi formuły w systemie dedukcji naturalnej.

***Przykład* 2.41.**

$$\dfrac{\dfrac{[z : (a \wedge b) \vee (c \wedge d)] \qquad \mathcal{D}_1 \qquad \mathcal{D}_2}{\begin{array}{ll}\mathsf{when}\ z & (\lambda x.\mathsf{Pair}\ (\mathsf{Inl}\ (\mathsf{fst}\ x))\ (\mathsf{Inl}\ (\mathsf{snd}\ x))) \\ & (\lambda y.\mathsf{Pair}\ (\mathsf{Inr}\ (\mathsf{fst}\ y))\ (\mathsf{Inr}\ (\mathsf{snd}\ y))) : (a \vee c) \wedge (b \vee d)\end{array}}\ \vee\mathtt{e}}{\begin{array}{ll}\lambda z.\mathsf{when}\ z & (\lambda x.\mathsf{Pair}\ (\mathsf{Inl}\ (\mathsf{fst}\ x))\ (\mathsf{Inl}\ (\mathsf{snd}\ x))) \\ & (\lambda y.\mathsf{Pair}\ (\mathsf{Inr}\ (\mathsf{fst}\ y))\ (\mathsf{Inr}\ (\mathsf{snd}\ y))) : ((a \wedge b) \vee (c \wedge d)) \to ((a \vee c) \wedge (b \vee d))\end{array}}\ \to\mathtt{i}$$

gdzie $\mathcal{D}_1$ i $\mathcal{D}_2$ oznaczają odpowiednio

$$\dfrac{\dfrac{\dfrac{\dfrac{[x : a \wedge b]}{\mathsf{fst}\ x : a}\ \wedge\mathtt{e}_1}{\mathsf{Inl}\ (\mathsf{fst}\ x) : (a \vee c)}\ \vee\mathtt{i}_1 \qquad \dfrac{\dfrac{[x : a \wedge b]}{\mathsf{snd}\ x : b}\ \wedge\mathtt{e}_2}{\mathsf{Inl}\ (\mathsf{snd}\ x) : (b \vee d)}\ \vee\mathtt{i}_1}{\mathsf{Pair}\ (\mathsf{Inl}\ (\mathsf{fst}\ x))\ (\mathsf{Inl}\ (\mathsf{snd}\ x)) : (a \vee c) \wedge (b \vee d)}\ \wedge\mathtt{i}}{\lambda x.\mathsf{Pair}\ (\mathsf{Inl}\ (\mathsf{fst}\ x))\ (\mathsf{Inl}\ (\mathsf{snd}\ x)) : (a \wedge b) \to ((a \vee c) \wedge (b \vee d))}\ \to\mathtt{i}$$

i

$$\dfrac{\dfrac{\dfrac{\dfrac{[y : c \wedge d]}{\mathsf{fst}\ y : c}\ \wedge\mathtt{e}_1}{\mathsf{Inr}\ (\mathsf{fst}\ y) : (a \vee c)}\ \vee\mathtt{i}_2 \qquad \dfrac{\dfrac{[y : c \wedge d]}{\mathsf{snd}\ y : d}\ \wedge\mathtt{e}_2}{\mathsf{Inr}\ (\mathsf{snd}\ y) : (b \vee d)}\ \vee\mathtt{i}_2}{\mathsf{Pair}\ (\mathsf{Inr}\ (\mathsf{fst}\ y))\ (\mathsf{Inr}\ (\mathsf{snd}\ y)) : (a \vee c) \wedge (b \vee d)}\ \wedge\mathtt{i}}{\lambda y.\mathsf{Pair}\ (\mathsf{Inr}\ (\mathsf{fst}\ y))\ (\mathsf{Inr}\ (\mathsf{snd}\ y)) : (c \wedge d) \to ((a \vee c) \wedge (b \vee d))}\ \to\mathtt{i}$$

3. System ET

System ET powstał jako część pracy doktorskiej dr Zdzisława Spławskiego[10][1]. Po raz pierwszy został publicznie przedstawiony na Seminarium Warszawsko–Wrocławskim w listopadzie 1992 r. Nazwa ET jest skrótem Extended T i nawiązuje do systemu T Gödla. Jak opisano w podrozdziale 2.3 system T jest rozszerzeniem rachunku $\lambda^{\to}$ o liczby naturalne: typ NAT, rekursor Rec i konstruktory O i Suc. System ET jest rozszerzeniem rachunku $\lambda^{\to}$ o możliwość definiowania nowych typów danych. Nowe typy danych definiuje się przez indukcję, podając listę konstruktorów nowego typu, bądź przez koindukcję podając listę destruktorów. Ilustrują to następujące przykłady. Przykłady będą podane w postaci zbliżonej do programów w implementacji ET. Znak „•” poprzedza część generowaną przez system automatycznie na podstawie definicji typu lub kotypu.

***Przykład* 3.1** (Przykład definicji typu)**.** Definicja liczb naturalnych w systemie ET.

$$
\begin{array}{rl}
\texttt{datatype} & \mathsf{NAT} = \mathsf{Suc}\ \texttt{from}\ \mathsf{NAT} \mid \mathsf{O}\ ; \\
\bullet\ \texttt{con} & \mathsf{Suc} : \mathsf{NAT} \to \mathsf{NAT} \\
 & \mathsf{O} : \mathsf{NAT} \\
\bullet\ \texttt{iter} & \mathsf{NatIt} : \mathsf{Nat} \to (V \to V) \to V \to V \\
\bullet\ \texttt{comp} & \mathsf{NatIt}\ (\mathsf{Suc}\ u) = \lambda v_1\ v_2.v_1\ (\mathsf{NatIt}\ u\ v_1\ v_2) \\
 & \mathsf{NatIt}\ \mathsf{O} = \lambda v_1\ v_2.v_2
\end{array}
$$

Liczby naturalne są definiowane przez dwa konstruktory: O interpretowany jako liczba 0 i Suc interpretowany jako następnik. Liczbie 3 odpowiada term Suc(Suc(Suc(0))). Pierwsza linia przykładu zawiera definicję typu: słowo kluczowe `datatype`, nazwę typu (NAT) i definicje kolejnych konstruktorów rozdzielone znakiem „|”. Każdy z konstruktorów jest definiowany przez nazwę i listę typów argumentów, poprzedzoną słowem kluczowym `from`. Typem wyniku konstruktora jest zawsze definiowany typ danych (w tym przykładzie NAT). Suc `from` NAT oznacza, że konstruktor Suc musi być zaaplikowany do argumentu typu NAT. Konstruktor O jest zeroargumentowy. Wprowadzoną definicję system potwierdza wyświetlając, po „• `con`”, nazwę i typ każdego z konstruktorów.

System ET automatycznie generuje dla wprowadzonego typu iterator NatIt. Jego typ przedstawiony jest po „• `iter`” a reguły obliczania (redukcji) po „• `comp`”.

Z logicznego punktu widzenia (nawiązując do izomorfizmu Curry’ego–Howarda) nowy typ jest nowym operatorem logicznym. Konstruktory odpowiadają regułom wprowadzania, a iterator regule eliminacji. Reguły obliczania odpowiadają regułom definiującym relację równości. W podrozdziale 3.2 przedstawione są reguły wnioskowania formalizujące te związki.

[1] Autor w pracy doktorskiej i późniejszych raportach używał nazwy IPL - Inferential Programming Language and Logic. W 1998r. nazwa została zmieniona na ET.

***Przykład* 3.2.** Definicja listy elementów typu U w systemie ET.

$$\begin{array}{ll}
\texttt{datatype} & \mathsf{LIST}\ U = \mathsf{Cons}\ \texttt{from}\ U\ (\mathsf{LIST}\ U) \mid \mathsf{Nil}\ ; \\
\bullet\ \texttt{con} & \mathsf{Cons} : U \to (\mathsf{List}\ U) \to (\mathsf{LIST}\ U) \\
 & \mathsf{Nil} : \mathsf{LIST}\ U \\
\bullet\ \texttt{iter} & \mathsf{ListIt} : \mathsf{List}\ U \to (U \to V \to V) \to V \to V \\
\bullet\ \texttt{comp} & \mathsf{ListIt}\ (\mathsf{Cons}\ u_1\ u_2) = \lambda v_1\ v_2.v_1\ u_1\ (\mathsf{ListIt}\ u_2\ v_1\ v_2) \\
 & \mathsf{ListIt}\ \mathsf{Nil} = \lambda v_1\ v_2.v_2
\end{array}$$

Listę definiuje się za pomocą dwóch konstruktorów: Cons, który dodaje element do listy i Nil, który jest interpretowany jako lista pusta. Ciekawą funkcją, którą można zdefiniować w ET jest map. map f l aplikuje funkcję f do wszystkich elementów listy l. Specyfikacją map jest układ równań.

$$\begin{array}{ll}
\mathit{map}\ f\ \mathit{Nil} & = \mathit{Nil} \\
\mathit{map}\ f\ (\mathit{Cons}\ h\ t) & = \mathit{Cons}\ (f\ h)\ (\mathit{map}\ f\ t)
\end{array}$$

Funkcję map można w ET zdefiniować następująco.

$$\mathsf{map}\ = \lambda f\ l.\ \mathsf{ListIt}\ l\ (\lambda h\ t.\ \mathsf{Cons}\ (f\ h)\ t)\ \mathsf{Nil} : (U \to V) \to \mathsf{LIST}\ U \to \mathsf{LIST}\ V$$

Dowód, że definicja map spełnia specyfikację znajduje się w podrozdziale 4.2.4. Należy zwrócić uwagę, że dzięki polimorfizmowi funkcja map jest zdefiniowana dla list dowolnego typu.

***Przykład* 3.3.** Definicja strumienia (nieskończonej listy, nieskończonego ciągu) elementów typu U w systemie ET.

$$\begin{array}{ll}
\texttt{codatatype} & \mathsf{STREAM}\ U = \mathsf{Shd}\ \texttt{to}\ U\ \&\ \mathsf{Stl}\ \texttt{to}\ \mathsf{STREAM}\ U\ ; \\
\bullet\ \texttt{des} & \mathsf{Shd} : (\mathsf{STREAM}\ U) \to U \\
 & \mathsf{Stl} : (\mathsf{STREAM}\ U) \to (\mathsf{STREAM}\ U) \\
\bullet\ \texttt{coiter} & \mathsf{SCoit} : (V \to U) \to (V \to V) \to V \to (\mathsf{STREAM}\ U) \\
\bullet\ \texttt{comp} & \mathsf{Shd}\ (\mathsf{SCoit}\ v_1\ v_2\ u) = v_1\ u \\
 & \mathsf{Stl}\ (\mathsf{SCoit}\ v_1\ v_2\ u) = \mathsf{Stl}\ (\mathsf{SCoit}\ v_1\ v_2\ (v_2\ u))
\end{array}$$

Strumień definiuje się przez koindukcję podając listę destruktorów przedzielonych znakiem „&". Po nazwie destruktora następuje deklaracja typu poprzedzona słowem `to`. Shd interpretuje się jako pierwszy element strumienia, a Stl jako ogon, czyli strumień zaczynający się od następnego elementu. System automatycznie generuje koiterator i reguły obliczania dla zdefiniowanego typu. Reguły obliczania potwierdzają poprawność interpretacji destruktorów, tzn. z postaci reguł wynika co zwracają destruktory.

W przeciwieństwie do typów, które definiuje się podając sposoby konstrukcji, kotypy definiuje się określając sposoby destrukcji. W tym przykładzie strumień jest typem, z którego można zawsze wyliczyć pierwszy element i ogon (w odróżnieniu od listy skończonej).

Kolejnym przykładem wyjaśniającym istotę kotypów niech będzie definicja strumienia (ciągu) kolejnych liczb naturalnych.

***Przykład* 3.4** (Strumień kolejnych liczb naturalnych)**.** Używając definicji liczb naturalnych i strumienia można zdefiniować funkcję natstr, która zaaplikowana do liczby n zwraca strumień kolejnych liczb naturalnych, począwszy od n. Specyfikacją natstr jest układ równań.

$$\begin{array}{ll}
\mathit{Shd}\ (\mathit{natstr}\ n) & = n \\
\mathit{Stl}\ (\mathit{natstr}\ n) & = \mathit{natstr}\ (\mathit{Suc}\ N)
\end{array}$$

Funkcję natstr można zdefiniować następująco.

$$\mathsf{natstr} \quad = \lambda n.\mathsf{SCoit}\ (\lambda x.x)\ \mathsf{Suc}\ n : \mathsf{NAT} \to \mathsf{STREAM\ NAT}$$

Własności zdefiniowanej funkcji wynikają z reguł obliczania. Można sprawdzić, że funkcja spełnia swoją specyfikację (równości wynikają bądź z definicji, bądź z zastosowania reguły obliczania). Trzeba podkreślić, że w implementacji ET takie równości sprawdzane są automatycznie (podrozdział 4.2.5).

$$\begin{aligned}
\mathsf{Shd}\ (\mathsf{natstr}\ N) &= \mathsf{Shd}\ (\mathsf{SCoit}\ (\lambda x.x)\ \mathsf{Suc}\ N) \\
&= N \\
\mathsf{Stl}\ (\mathsf{natstr}\ N) &= \mathsf{Stl}\ (\mathsf{SCoit}\ (\lambda x.x)\ \mathsf{Suc}\ N) \\
&= \mathsf{SCoit}\ (\lambda x.x)\ \mathsf{Suc}\ (\mathsf{Suc}\ n) \\
&= \mathsf{natstr}\ (\mathsf{Suc}\ N)
\end{aligned}$$

Inną ciekawą funkcją działającą na strumieniach jest smap. smap aplikuje funkcję, która jest pierwszym argumentem, do wszystkich (nieskończenie wielu) elementów strumienia, który jest drugim argumentem. Specyfikacją smap jest para równań.

$$\begin{aligned}
\mathit{Shd}\ (\mathit{smap}\ f\ s) &= f\ (\mathit{Shd}\ s) \\
\mathit{Stl}\ (\mathit{smap}\ f\ s) &= \mathit{smap}\ f\ (\mathit{Stl}\ s)
\end{aligned}$$

Funkcję smap można w systemie ET zdefiniować następująco.

$$\mathsf{smap} \quad = \lambda f\ s.\ \mathsf{SCoit}\ (\lambda x.f\ (\mathsf{Shd}\ x))\ \mathsf{Stl}\ s : (V \to U) \to \mathsf{STREAM}\ V \to \mathsf{STREAM}\ U$$

Dowód, że smap spełnia specyfikację przedstawiony jest w podrozdziale 4.2.5.

Jeśli sqr:NAT→NAT jest funkcją podnoszącą swój argument do kwadratu (definicja tej funkcji zostanie pominięta), to zdefiniowana poniżej funkcja sqrstr jest strumieniem kwadratów kolejnych liczb naturalnych, począwszy od zera.

$$\mathsf{sqrstr} \quad = \mathsf{smap\ sqrt}\ (\mathsf{natstr\ O})$$

Należy zwrócić uwagę, że we wszystkich przykładach kotyp reprezentował strumień nieskończonej długości. Co ciekawe smap aplikuje dowolną funkcję do wszystkich, a więc do nieskończenie wielu, elementów tego strumienia. Pomimo tej własności termy (programy) w systemie ET mają własność mocnej normalizacji, tzn. prowadzą do wyniku w skończonej ilości kroków. Wyjaśnienie tej pozornej sprzeczności leży w cechach kotypów. Strumień jest „przepisem” na wyliczenie kolejnych elementów i jako taki, pomimo że reprezentuje obiekt potencjalnie nieskończony, nie składa się z nieskończenie wielu elementów. Funkcja smap w odpowiedni sposób modyfikuje sposób wyliczania kolejnych elementów.

System ET zostanie przedstawiony w trzech częściach: system bez reguł pozwalających definiować nowe typy, reguły pozwalające definiować nowe typy przez indukcję i reguły pozwalające definiować nowe typy przez koindukcję.

Należy zwrócić uwagę na niejednoznaczne użycie terminu „typ” w opisie systemu ET. „Typ” w zależności od kontekstu określa wszystkie typy w systemie, typy danych lub, kiedy podkreśla się różnicę między indukcją i koindukcją, tylko typy danych zdefiniowane przez indukcję. Typy danych zdefiniowane przez koindukcję nazywa się w tym ostatnim przypadku „kotypami”.

3.1. Podstawowe definicje

Definicja fragmentu systemu ET bez reguł pozwalających definiować nowe typy, przypomina definicję $\lambda^{\to}$.

Definicja 3.5. Zbiór wyrażeń typowych $\mathcal{T}$ *definiuje się przy pomocy przeliczalnego zbioru zmiennych typowych* $\mathcal{V}_\mathcal{T} = \{V, V', V'', \ldots\}$ *następująco:*

$$\begin{aligned} \mathcal{V}_\mathcal{T} &::= V \mid \mathcal{V}_\mathcal{T}' \\ \mathcal{T} &::= \mathcal{V}_\mathcal{T} \mid \mathcal{T} \to \mathcal{T} \mid \Phi\mathcal{T}_1 \ldots \mathcal{T}_n \mid \Psi\mathcal{T}_1 \ldots \mathcal{T}_m \end{aligned}$$

gdzie Φ *jest używane dla oznaczenia konstruktorów typów zdefiniowanych indukcyjnie, a* Ψ *— koindukcyjnie.*

Definicja 3.6. Zbiór quasitermów Λ *definiuje się przy użyciu nieskończonego, przeliczalnego zbioru zmiennych* $\mathcal{V} = \{v, v', v'', \ldots\}$.

$$\begin{aligned} &\mathcal{V} ::= v \mid \mathcal{V}' \\ &\Lambda ::= \mathcal{V} \mid (\Lambda\Lambda) \mid (\lambda\mathcal{V}.\Lambda) \mid (\texttt{let}\ \mathcal{V}_1 \ = \ \Lambda_1\ \texttt{in}\ \Lambda) \qquad \textit{gdzie } \mathcal{V}_1 \notin FV(\Lambda_1) \end{aligned}$$

Konstrukcja `let` $x \ = \ M$ `in` N umożliwia lokalne deklarowanie stałych (funkcji). Każde wolne wystąpienie x w N jest zamieniane na M, z tym że każde z tych wystąpień może mieć inny typ. Z tego powodu konstrukcja z `let` nie jest równoważna termowi $(\lambda x.N)M$, w którym każde wystąpienie x w N ma ten sam typ.

Poniżej przedstawione są łącznie reguły typizacji i reguły równościowe w ET.

Definicja 3.7. *Reguły wnioskowania dla* ET.

$$\frac{}{\Gamma, x : \sigma \rhd x : \sigma}\ \text{Ass}$$

$$\frac{\Gamma \rhd M : \sigma}{\Gamma \rhd M = M : \sigma}\ \text{Refl}_1 \qquad\qquad \frac{\Gamma \rhd M = M : \sigma}{\Gamma \rhd M : \sigma}\ \text{Refl}_2$$

$$\frac{\Gamma \rhd M = N : \sigma}{\Gamma \rhd N = M : \sigma}\ \text{Sym} \qquad\qquad \frac{\Gamma \rhd K = N : \sigma \qquad \Gamma \rhd N = M : \sigma}{\Gamma \rhd K = M : \sigma}\ \text{Trans}$$

$$\frac{\Gamma \rhd K = L : \sigma \to \tau \qquad \Gamma \rhd N = M : \sigma}{\Gamma \rhd KN = LM : \tau}\ \text{MonApp}$$

$$\frac{\Gamma, x : \sigma \rhd N = M : \tau}{\Gamma \rhd \lambda x.N = \lambda x.M : \sigma \to \tau}\ \text{MonAbs}$$

$$\frac{\Gamma \rhd (\lambda x.M)N : \sigma}{\Gamma \rhd (\lambda x.M)N = M[x := N] : \sigma}\ \beta \qquad\qquad \frac{\Gamma \rhd \lambda x.Mx : \sigma}{\Gamma \rhd \lambda x.Mx = M : \sigma}\ \eta \quad \text{gdzie } x \notin FV(M)$$

$$\frac{\Gamma \rhd N = M : \sigma}{\Gamma[U := \tau] \rhd N = M : \sigma[U := \tau]}\ \text{Inst}$$

$$\frac{\Gamma \rhd N : \sigma \qquad \Gamma \rhd M[x := N] : \tau}{\Gamma \rhd (\texttt{let}\ x \ = \ N\ \texttt{in}\ M) = M[x := N] : \tau}\ \text{Let} \quad \text{gdzie } x \notin FV(N)$$

Termami w ET są quasitermy dla których można przy pomocy reguł wnioskowania wyprowadzić typ. Relacja redukcji, definiowana analogicznie jak w przypadku poprzednio omawianych systemów, spełnia twierdzenie Churcha–Rossera (twierdzenie 2.13). System ET zachowuje własność mocnej normalizacji.

Twierdzenie 3.8. *Termy w systemie ET są mocno normalizowalne.*

Dowód tego twierdzenia polega na zdefiniowaniu tłumaczenia systemu ET w system F, gdzie termy są mocno normalizowalne.

3.2. Definiowanie typów indukcyjnych

System ET umożliwia wprowadzania nowych typów danych przez indukcję, tzn. w argumentach konstruktorów mogą występować elementy definiowanego typu. Przykładem typu indukcyjnego są liczby naturalne. Argumentem konstruktora Suc (następnika) jest właśnie liczba naturalna.

Wprowadzenie typów indukcyjnych do ET wymaga następujących definicji.

Definicja 3.9. *Zbiory wyrażeń typowych $\tau[X]^p$, w których X występuje tylko pozytywnie i $\tau[X]^n$, w których X występuje tylko negatywnie definiuje się następująco:*

$$\begin{aligned} \tau[X]^p &::= \sigma \mid X \mid \tau_1[X]^n \to \tau_2[X]^p \mid \Phi\tau_1[X]^p \ldots \tau_n[X]^p \\ \tau[X]^n &::= \sigma \mid \tau_1[X]^p \to \tau_2[X]^n \mid \Phi\tau_1[X]^n \ldots \tau_n[X]^n \\ &\quad \textit{gdzie} \quad X \notin FTV(\sigma) \end{aligned}$$

Twierdzenie 3.10. *Dla każdego wyrażenia typowego $\tau[X]^p$, w którym X występuje tylko pozytywnie istnieje* funkcja monotoniczności

$$Mon^{\alpha\to\beta}_{\tau[X]^p} : (\alpha \to \beta) \to \tau[X := \alpha]^p \to \tau[X := \beta]^p,$$

która zaaplikowana do funkcji f typu $\alpha \to \beta$ i wyrażenia typu $\tau[X := \alpha]$ zwraca wyrażenie typu $\tau[X := \beta]$, w którym wszystkie wystąpienie wyrażeń typu α zamieniono na wyrażenia typu β przy pomocy funkcji f.

Definicja funkcji monotoniczności, a więc dowód powyższego twierdzenia, znajdują się w [10].

Definicja nowego typu danych $\Phi\vec{U}$ składa się z listy nazw konstruktorów wraz z typami argumentów dla każdego z konstruktorów. Typ $\Phi\vec{U}$ może występować w typach argumentów konstruktorów tylko pozytywnie. Na tej postawie generowany jest iterator dla definiowanego typu. Deklarowany typ jest polimorficzny tzn. wszystkie zmienne typowe występujące w argumentach konstruktorów stają się argumentami typu danych. W przykładzie 3.3 zmienna typowa U, oznaczająca typ elementów strumienia, jest argumentem typu danych. Za zmienne będące argumentami typu danych mogą być podstawiane dowolne wyrażenia typowe. Strumień został więc zdefiniowany dla dowolnego typu elementów. W przykładzie 3.4 użyto strumienia liczb naturalnych, więc za zmienną U został podstawiony typ NAT. Polimorficzne typy danych są analogonem funkcji polimorficznych. W obu przypadkach polimorfizm zwalnia programistę z oddzielnego definiowania funkcji lub typu danych dla każdego konkretnego typu oddzielnie.

Podczas wprowadzania następnych definicji używane będą następujące oznaczenia:

***Notacja* 3.11.**

1. Φ oznacza nowy typ danych;
2. r oznacza ilość konstruktorów definiowanego typu, $r \geqslant 0$;
3. $\boldsymbol{Con^i_\Phi}$ oznacza i-ty konstruktor, $i = 1 \ldots r$;
4. $p(i)$ oznacza ilość argumentów i-tego konstruktora, $p(i) \geqslant 0$, $i = 1 \ldots r$;
5. ξ^i_j jest typem j-tego argumentu i-tego konstruktora, $i = 1 \ldots r$, $j = 1 \ldots p(i)$;

6. $\boldsymbol{Elim_\Phi}$ oznacza iterator typu Φ;
7. $\vec{U}$ oznacza zbiór zmiennych typowych zawartych w typach argumentów konstruktorów, $\vec{U} = \bigcup_{i=1}^{r} \bigcup_{j=1}^{p(i)} FTV(\xi_j^i)$

Poniżej przedstawiony jest schemat definiowania typu indukcyjnego $\Phi\vec{U}$. Na podstawie listy konstruktorów (w pierwszej linii) system automatycznie generuje: typy konstruktorów oznaczone przez • `con`, typ iteratora oznaczony przez • `iter` i reguły obliczania dla iteratora oznaczone przez • `comp`. Reguł obliczania jest tyle ile konstruktorów (po jednej regule dla każdego konstruktora). Trzeba podkreślić, że typ $\Phi\vec{U}$ może występować w typach argumentów konstruktorów ξ tylko pozytywnie.

Definicja 3.12. *Schemat definiowania typu indukcyjnego:*

$$
\begin{array}{ll}
\texttt{datatype} & \Phi\vec{U} = \boldsymbol{Con_\Phi^1}\ \texttt{from}\ \xi_1^1 \ldots \xi_{p(1)}^1 \mid \ldots \mid \boldsymbol{Con_\Phi^r}\ \texttt{from}\ \xi_1^r \ldots \xi_{p(r)}^r \\
\bullet\ \texttt{con} & \boldsymbol{Con_\Phi^1} : \xi_1^1 \to \ldots \to \xi_{p(1)}^1 \to \Phi\vec{U} \\
 & \vdots \\
 & \boldsymbol{Con_\Phi^r} : \xi_1^r \to \ldots \to \xi_{p(r)}^r \to \Phi\vec{U} \\
\bullet\ \texttt{iter} & \boldsymbol{Elim_\Phi} : \Phi\vec{U} \to (\xi_1^1[\Phi\vec{U} := V] \to \ldots \to \xi_{p(1)}^1[\Phi\vec{U} := V] \to V) \to \ldots \\
 & \qquad \ldots \to (\xi_1^r[\Phi\vec{U} := V] \to \ldots \to \xi_{p(r)}^r[\Phi\vec{U} := V] \to V) \to V \\
\bullet\ \texttt{comp} & \boldsymbol{Elim_\Phi}(\boldsymbol{Con_\Phi^1} u_1 \ldots u_{p(1)}) = \lambda v_1 \ldots v_r . v_1 \widehat{u}_1^1 \ldots \widehat{u}_{p(1)}^1 \\
 & \vdots \\
 & \boldsymbol{Elim_\Phi}(\boldsymbol{Con_\Phi^r} u_1 \ldots u_{p(r)}) = \lambda v_1 \ldots v_r . v_r \widehat{u}_1^r \ldots \widehat{u}_{p(r)}^r \\
 & \textit{gdzie } \widehat{u}_j^i := Mon_{\xi_j^i[\Phi\vec{U}:=V]^p}^{\Phi\vec{U}\to V}(\lambda z. \boldsymbol{Elim_\Phi}\ z\ v_1 \ldots v_r)\ u_j
\end{array}
$$

Jeśli i-ty konstruktor jest zeroargumentowy, tzn. $p(i) = 0$, *to typ* $\xi_1^i \to \ldots \to \xi_{p(i)}^i \to X$ *oznacza* X.

W przykładzie 3.1 zastosowano powyższy schemat do zdefiniowania liczb naturalnych. Należy zauważyć, że:

- typ nazywa się NAT, Φ=NAT;
- pierwszy konstruktor nazywa się Suc i ma jeden argument typu NAT, $\boldsymbol{Con_\Phi^1} = \mathsf{Suc}$, $p(1) = 1$, $\xi_1^1 = \mathsf{NAT}$;
- drugi konstruktor nazywa się O i jest zeroargumentowy, $\boldsymbol{Con_\Phi^2} = \mathsf{O}$, $p(1) = 0$;
- typy argumentów konstruktorów nie zawierają żadnych zmiennych typowych, więc $\vec{U} = \emptyset$.

3.2.1. Reguły wnioskowania dla typów indukcyjnych

Schemat pozwalający definiować nowe typy danych można zapisać także w formie reguł wnioskowania. Jak wspomniano wcześniej konstruktory odpowiadają regułom wprowadzania, a iterator regule eliminacji.

$$
\frac{\Gamma \rhd M_1 : \xi_1^i[\vec{U} := \vec{\sigma}] \quad \ldots \quad \Gamma \rhd M_{p(i)} : \xi_{p(i)}^i[\vec{U} := \vec{\sigma}]}{\Gamma \rhd \boldsymbol{Con_\Phi^i} M_1 \ldots M_{p(i)} : \Phi\vec{\sigma}}\ \Phi\mathtt{i}^i
$$

$$
\frac{\Gamma \rhd K : \Phi\vec{\sigma} \quad \Gamma, x_1 : \zeta_1^1, \ldots, x_{p(1)} : \zeta_{p(i)}^1 \rhd N_1 : \tau \quad \ldots \quad \Gamma, x_1 : \zeta_1^r, \ldots, x_{p(r)} : \zeta_{p(r)}^r \rhd N_r : \tau}{\Gamma \rhd \boldsymbol{Elim_\Phi}\ K\ (\lambda x_1 \ldots x_{p(1)}.N_1) \ldots (\lambda x_1 \ldots x_{p(r)}.N_r) : \tau}\ \Phi\mathtt{e}
$$

gdzie $\zeta^i_j := \xi^i_j[\vec{U} := \vec{\sigma}][\Phi\vec{U} := \tau]$. Reguły obliczania iteratora odpowiadają regułom definiującym relację równości.

$$\frac{\Gamma \rhd \boldsymbol{Elim_\Phi}(\boldsymbol{Con^i_\Phi}\ M_1 \ldots M_{p(i)})\ L_1 \ldots L_r : \tau}{\Gamma \rhd \boldsymbol{Elim_\Phi}(\boldsymbol{Con^i_\Phi}\ M_1 \ldots M_{p(i)})\ L_1 \ldots L_r = L_i\ \widehat{M}^i_1 \ldots \widehat{M}^i_{p(i)} : \tau}\ \Phi{=}^i$$

gdzie $\widehat{M}^i_j := Mon^{\Phi\vec{\sigma}\to\tau}_{\xi^i_j[\vec{U}:=\vec{\sigma}]}(\lambda z.\boldsymbol{Elim_\Phi}\ z\ L_1 \ldots L_r)\ M_j$

3.3. Definiowanie kotypów

Nowe typy danych wprowadza się przez koindukcję definiując listę destruktorów typu, wraz z typami wyniku każdego z destruktorów. Definiowany typ może występować w typach wyników destruktorów tylko pozytywnie.

Definicja wyrażeń typowych $\tau[X]^p$ i $\tau[X]^n$ (definicja 3.9) zostaje rozszerzona na kotypy.

Definicja 3.13. *Zbiory wyrażeń typowych $\tau[X]^p$, w których X występuje tylko pozytywnie i $\tau[X]^n$, w których X występuje tylko negatywnie definiuje się następująco:*

$$\begin{aligned}
\tau[X]^p &::= \sigma \mid X \mid \tau_1[X]^n \to \tau_2[X]^p \mid \Psi\tau_1[X]^p \ldots \tau_n[X]^p \\
\tau[X]^n &::= \sigma \mid \tau_1[X]^p \to \tau_2[X]^n \mid \Psi\tau_1[X]^n \ldots \tau_n[X]^n \\
&\quad \textit{gdzie} \quad X \notin FTV(\sigma)
\end{aligned}$$

We wprowadzanych definicja używane są następujące oznaczenia:

***Notacja* 3.14.**

1. Ψ oznacza nowy typ danych;
2. r oznacza ilość destruktorów definiowanego typu, $r \geqslant 0$;
3. $\boldsymbol{Des^i_\Phi}$ oznacza i-ty destruktor, $i = 1 \ldots r$;
4. $p(i)$ oznacza ilość składników wyniku i-tego destruktora, $p(i) \geqslant 0$, $i = 1 \ldots r$;
5. $\boldsymbol{Intro_\Phi}$ oznacza iterator typu Φ;
6. $\vec{U}$ oznacza zbiór zmiennych typowych zawartych w typach wyników destruktorów, $\vec{U} = \bigcup_{i=1}^{r} \bigcup_{j=1}^{p(i)} FTV(\xi^i_j)$;
7. $+$ oznacza sumę rozłączną.

Poniżej przedstawiony jest schemat definiowania kotypu. System na podstawie listy destruktorów generuje ich typy, koiterator i reguły obliczania dla każdego z destruktorów.

Definicja 3.15. *Schemat definiowania kotypu:*

$$\begin{array}{ll}
\texttt{codatatype} & \Psi\vec{U} = \boldsymbol{Des^1_\Phi}\ \texttt{to}\ \xi^1_1 \dots \xi^1_{p(1)} \mid \dots \mid \boldsymbol{Des^r_\Phi}\ \texttt{to}\ \xi^r_1 \dots \xi^r_{p(r)} \\
\bullet\ \texttt{des} & \boldsymbol{Des^1_\Phi} : \Psi\vec{U} \to \xi^1_1 + \dots + \xi^1_{p(1)} \\
& \vdots \\
& \boldsymbol{Des^r_\Phi} : \Psi\vec{U} \to \xi^r_1 + \dots + \xi^r_{p(r)} \\
\bullet\ \texttt{coiter} & \boldsymbol{Intro_\Phi} : (V \to \xi^1_1[\Psi\vec{U} := V] + \dots + \xi^1_{p(1)}[\Psi\vec{U} := V]) \to \dots \\
& \qquad \dots \to (V \to \xi^r_1[\Psi\vec{U} := V] + \dots + \xi^r_{p(r)}[\Psi\vec{U} := V]) \to V \to \Psi\vec{U} \\
\bullet\ \texttt{comp} & \boldsymbol{Des^1_\Phi}(\boldsymbol{Intro_\Phi} v_1 \dots v_r u) = Mon^{V \to \Psi\vec{U}}_{(\xi^1_1 + \dots + \xi^1_{p(1)})[\Psi\vec{U} := V]^p}(\boldsymbol{Intro_\Phi} v_1 \dots v_r)(v_1 u) \\
& \vdots \\
& \boldsymbol{Des^r_\Phi}(\boldsymbol{Intro_\Phi} v_1 \dots v_r u) = Mon^{V \to \Psi\vec{U}}_{(\xi^r_1 + \dots + \xi^r_{p(r)})[\Psi\vec{U} := V]^p}(\boldsymbol{Intro_\Phi} v_1 \dots v_r)(v_r u)
\end{array}$$

3.3.1. Reguły wnioskowania dla typów koindukcyjnych

Podobnie jak w przypadku typów definiowanych przez indukcję, schemat definiowania typów definiowanych przez koindukcję można zapisać w formie reguł wnioskowania. Destruktory odpowiadają regułom eliminacji a koiterator regule wprowadzania. Reguły obliczania odpowiadają regułom definiującym relację równości.

$$\frac{\Gamma, x : \sigma \rhd N_1 : \zeta^1_1 + \dots + \zeta^1_{p(1)} \qquad \dots \qquad \Gamma, x : \sigma \rhd N_r : \zeta^r_1 + \dots + \zeta^r_{p(r)} \qquad \Gamma \rhd M : \sigma}{\Gamma \rhd \boldsymbol{Intro_\Phi}\ (\lambda x.N_1) \dots (\lambda x.N_r)\ M : \Psi\vec{U}}\ \Psi\texttt{i}$$

$$\frac{\sigma \rhd K : \Psi\vec{U}}{\Gamma \rhd \boldsymbol{Des^i_\Phi} K : (\xi^i_1 + \dots + \xi^i_{p(i)})[\vec{U} := \vec{\sigma}]}\ \Psi\texttt{e}^i$$

$$\frac{\Gamma \rhd \boldsymbol{Des^i_\Phi}(\boldsymbol{Intro_\Phi}\ M_1 \dots M_r\ L)\ : (\xi^i_1 + \dots + \xi^i_{p(i)})[\vec{U} := \vec{\sigma}]}{\begin{array}{l}\Gamma \rhd \boldsymbol{Des^i_\Phi}(\boldsymbol{Intro_\Phi}\ M_1 \dots M_r\ L) = \\ \quad = Mon^{V \to \Psi\vec{U}}_{(\xi^i_1 + \dots + \xi^i_{p(i)})[\Psi\vec{U} := V]^p}(\boldsymbol{Intro_\Phi} M_1 \dots M_r)(M_i L) : (\xi^i_1 + \dots + \xi^i_{p(i)})[\vec{U} := \vec{\sigma}]\end{array}}\ \Psi{=}^i$$

gdzie $\zeta^i_j := \xi^i_j[\vec{U} := \vec{\sigma}][\Phi\vec{U} := \tau]$.

W pracy dla czytelności przedstawiono system ET z iteratorami i koiteratorami. W [10] zawarte są reguły pozwalające generować również rekursory i korekursory, które są rozszerzeniem iteratorów i koiteratorów. Tam też znajdują się dowody wszystkich przytoczonych twierdzeń. Implementacja systemu ET, która jest częścią pracy, generuje dla definiowanych typów danych iteratory i rekursory, a dla kotypów koiteratory i korekursory.

4. Język programowania ET

Język programowania ET jest językiem funkcyjnym. Słowa kluczowe i gramatyka języka zostały tak dobrane, by w maksymalnym stopniu konstrukcje w ET przypominały analogiczne konstrukcje w SML-u – popularnym języku funkcyjnym [8]. Z tego powodu choćby pobieżna znajomość SML-a, bądź innego, podobnego języka funkcyjnego, jest pomocna w trakcie czytania gramatyki języka i przykładów programów w ET.

4.1. Gramatyka ET

Poniżej przedstawiona jest gramatyka języka ET. Symbole terminalne oznaczone są `czcionką maszynową`, symbole nieterminalne *czcionką pochyloną*. $[M]$ oznacza, że M powtarza się co najwyżej raz a $\{M\}$, że M powtarza się co najmniej raz. Zapis $\left| \begin{smallmatrix} M \\ N \end{smallmatrix} \right|$ oznacza M lub N. Gramatyka w sposób jednoznaczny wyznacza zbiór poprawnych składniowo programów, a także kolejność i kierunek wiązania poszczególnych konstrukcji np. infiksowych konstruktorów typów.

$$\begin{array}{rcl} \textit{program} & ::= & [\ \{\ \textit{declaration}\ \}\ ] \\ \textit{declaration} & ::= & \left[\left| \begin{array}{l} \textit{value\_binding} \\ \textit{term} \\ \textit{datatype\_declaration} \\ \textit{codatatype\_declaration} \end{array} \right| \right] \ \texttt{;} \end{array}$$

Program w ET jest ciągiem deklaracji. Deklaracje wprowadzają nowe definicje do środowiska. Środowisko jest funkcją przypisującą identyfikatorom termy (w praktyce jest po prostu ciągiem wprowadzonych definicji). Deklaracja jest przypisaniem (przypisuje term określonemu identyfikatorowi), termem (przypisaniem termu za domyślny identyfikator „`it`”), definicją typu lub definicją kotypu.

$$
\begin{array}{rcl}
\mathit{atomic\_term} & ::= & \left|\begin{array}{l} \mathit{value\_name}\\ \mathit{parameter}\\ \mathit{constructor}\\ \mathit{dectructor}\\ \mathit{iterator}\\ \mathit{recursor}\\ \mathit{coiterator}\\ \mathit{corecursor}\\ \texttt{()}\\ \texttt{(}\ \mathit{term}\ \texttt{)}\\ \texttt{let}\ \{\ \mathit{value\_binding}\ ;\ \}\ \texttt{in}\ \mathit{term}\ \texttt{end}\\ \texttt{fn}\ \{\ \mathit{parameter}\ \}\ \texttt{=>}\ \mathit{term}\\ \texttt{if}\ \mathit{term}\ \texttt{then}\ \mathit{term}\ \texttt{else}\ \mathit{term} \end{array}\right| \\
\mathit{application} & ::= & [\ \mathit{application}\ ]\ \mathit{atomic\_term}\\
\mathit{pair} & ::= & [\ \mathit{pair}\ \texttt{,}\ ]\ \mathit{application}\\
\mathit{term} & ::= & [\ \mathit{term}\ \texttt{=}\ ]\ \mathit{pair}
\end{array}
$$

Aplikacja, konstruktor pary i operator równości wiążą w lewo. Aplikacja wiąże najmocniej, a równość najsłabiej. Konstrukcja `fn` a b `=>` M jest równoważna `fn` a `=>` `fn` b `=>` M.

$$
\begin{array}{rcl}
\mathit{atomic\_type} & ::= & \left|\begin{array}{l} \mathit{type\_variable}\\ \mathit{datatype\_constructor}\\ \mathit{codatatype\_constructor}\\ \texttt{\{\}} \end{array}\right| \\
\mathit{type\_application} & ::= & \left|\begin{array}{l} \mathit{type\_application}\ \ \mathit{atomic\_type}\\ \mathit{datatype\_constructor}\\ \mathit{codatatype\_constructor} \end{array}\right| \\
\mathit{pair\_type} & ::= & [\ \mathit{pair\_type}\ \texttt{*}\ ]\ \left|\begin{array}{l} \mathit{type\_application}\ \ \mathit{atomic\_type}\\ \mathit{atomic\_type} \end{array}\right| \\
\mathit{union\_type} & ::= & [\ \mathit{union\_type}\ \texttt{+}\ ]\ \mathit{pair\_type}\\
\mathit{type} & ::= & \mathit{union\_type}\ [\ \texttt{->}\ \mathit{term}\ ]
\end{array}
$$

Konstruktory typów: funkcyjnego, sumy rozłącznej i pary są infiksowe. Konstruktor `->` wiąże w prawo, a pozostałe w lewo. Aplikacja typów wiąże w lewo. Konstruktor `->` wiąże najsłabiej, `*` wiąże mocniej od `+`. Najmocniej wiąże aplikacja typów[1].

$$
\mathit{value\_binding} \quad ::= \quad \texttt{val}\ \mathit{value\_name}\ \texttt{=}\ \mathit{term}
$$

Przypisanie wprowadza definicję do środowiska (przypisuje identyfikatorowi *value_name* wartość *term*).

[1] Należy podkreślić różnicę w stosunku do istniejącej implementacji IPL-a, w której kolejność aplikacji typów jest wyznaczana na podstawie liczby argumentów konstruktorów. Rozbiór gramatyczny w IPL-u zależy od kontekstu. Gramatyka ET jest bezkontekstowa.

$$\begin{array}{rcl}
\mathit{datatype_definition} & ::= & \texttt{datatype}\ \mathit{datatype_constructor}\ [\ \{\ \mathit{type_variable}\ \}\ ]\ \texttt{=} \\
 & & \quad [\ \mathit{constructor_list}\ ] \\
\mathit{codatatype_definition} & ::= & \texttt{codatatype}\ \mathit{codatatype_constructor}\ [\ \{\ \mathit{type_variable}\ \}\ ]\ \texttt{=} \\
 & & \quad [\ \mathit{destructor_list}\ ] \\
\mathit{constructor_list} & ::= & \mathit{constructor}\ [\ \texttt{from}\ \{\ \mathit{type}\ \}\ ]\ [\ \texttt{|}\ \mathit{constructor_list}\ ] \\
\mathit{destructor_list} & ::= & \mathit{destructor}\ [\ \texttt{to}\ \{\ \mathit{type}\ \}\ ]\ [\ \texttt{\&}\ \mathit{destructor_list}\ ]
\end{array}$$

Pominięcie konstrukcji `from` { *type* } powoduje zdefiniowanie konstruktora, który nie ma parametrów. Pominięcie `to` { *type* } w definicji destruktora jest równoważne zdefiniowaniu destruktora do typu `{}`, czyli absurdu[2].

$$\begin{array}{rcl}
\mathit{value_name} & ::= & \mathit{lower_case}\ \ \mathit{alphanum} \\
\mathit{parameter} & ::= & \mathit{lower_case}\ \ \mathit{alphanum} \\
\mathit{iterator} & ::= & \texttt{_}\ \mathit{datatype_constructor}\ \texttt{it} \\
\mathit{recursor} & ::= & \texttt{_}\ \mathit{datatype_constructor}\ \texttt{rec} \\
\mathit{coiterator} & ::= & \texttt{_}\ \mathit{codatatype_constructor}\ \texttt{ci} \\
\mathit{corecursor} & ::= & \texttt{_}\ \mathit{codatatype_constructor}\ \texttt{ct}
\end{array}$$

$$\begin{array}{rcl}
\mathit{constructor} & ::= & \mathit{upper_case}\ \ \mathit{alphanum} \\
\mathit{destructor} & ::= & \mathit{upper_case}\ \ \mathit{alphanum} \\
\mathit{type_variable} & ::= & \texttt{'}\ \mathit{alphanum} \\
\mathit{datatype_constructor} & ::= & \mathit{letter}\ \mathit{alphanum} \\
\mathit{codatatype_constructor} & ::= & \mathit{letter}\ \mathit{alphanum}
\end{array}$$

Nazwy konstruktorów i destruktorów rozpoczynają się wielką literą, a nazwy zmiennych związanych i definiowanych w środowisku – małą. Nazwy dla iteratorów system tworzy dodając znak podkreślenia na początku i `it` na końcu nazwy typu. Dla rekursorów jest to odpowiednio `rec`, dla koiteratorów – `ci`, a dla korekursorów – `cr`.

$$\begin{array}{rcl}
\mathit{letter} & ::= & \left|\begin{array}{l} \mathit{upper_case} \\ \mathit{lower_case} \end{array}\right| \\
\mathit{alphanum} & ::= & \left|\begin{array}{l} \mathit{letter} \\ \mathit{digit} \\ \texttt{_} \\ \texttt{'} \end{array}\right| \\
\mathit{uppercase} & ::= & \text{jeden ze znaków}\ \ \texttt{ABCDEFGHIJKLMNOPQRSTUVWXYZ} \\
\mathit{lowercase} & ::= & \text{jeden ze znaków}\ \ \texttt{abcdefghijklmnopqrstuvwxyz} \\
\mathit{digit} & ::= & \text{jeden ze znaków}\ \ \texttt{0123456789}
\end{array}$$

W tekście programu mogą znajdować się komentarze. Komentarz jest ciągiem znaków ograniczonych przez „`(*`” i „`*)`”.

[2] Definicja typu `{}` zostanie przytoczona w dalszej części pracy.

Poszczególne słowa języka muszą być rozdzielone spacją, znakiem tabulacji, znakiem końca linii bądź komentarzem jeśli granice słów nie są jednoznacznie wyznaczone. Można napisać np. `val b=fn a=>(a,a);`, ale usunięcie odstępu między `val` i `b` spowoduje zinterpretowanie `valb` jako identyfikatora. Podobnie między `fn` i `a`. Między pozostałymi słowami odstęp nie jest konieczny, ponieważ ich granice są rozróżnialne.

Obliczanie wartości funkcji polega na redukcji redeksów w termach. W ET są cztery rodzaje redeksów.

1. `(fn x => M) N` — (β-redeks), redukcja polega na podstawieniu za wolne wystąpienia `x` w `M` termu `N`;
2. `fn x => M x` — o ile nie ma wolnych wystąpień `x` w `M` (η-redeks); term redukuje się do `M`;
3. `IR (C M ...N)` — gdzie `IR` jest iteratorem bądź rekursorem, a `C` konstruktorem typu; redukcja następuje zgodnie z regułami obliczania dla typu;
4. `D (CIR M ...N)` — gdzie `CIR` jest koiteratorem bądź korekursorem, a `D` destruktorem kotypu; redukcja następuje zgodnie z regułami obliczania dla kotypu.

W reprezentacji wewnętrznej używana jest notacja de Bruijna, jednakże ze względu na wygodę użytkownika pamiętane są nazwy zmiennych związanych. Dzięki temu wyświetlane przez system nazwy zmiennych są zgodne z nazwami wprowadzonymi przez użytkownika. Podczas redukcji termu może dojść do sytuacji, w której nazwa zmiennej będzie niejednoznaczna. Do nazwy zmiennej dodany zostanie numer ujęty w nawiasy kwadratowe określający, która abstrakcja wiąże to wystąpienie zmiennej. Potrzeba zmiany nazw zmiennych została omówiona w uwagach 2.9 pkt. 5. Przedstawiony został tam przykład redukcji termu $(\lambda yz.yz)(\lambda yz.yz)$. W języku ET system wyświetla postać normalną termu, jeśli term poprzedzony jest poleceniem `norm`.

```
+ norm (fn y z => y z) (fn y z => y z);
fn z z => z[1] z : ('a -> 'b) -> 'a -> 'b
```

Postać normalna `fn z z => z[1] z` odpowiada termowi $\lambda wz.wz$ z przykładu. Zapis `z[1]` oznacza, że to wystąpienie zmiennej `z` jest wiązane przez abstrakcję znajdującą się o poziom wyżej.

Między deklaracjami mogą znajdować się następujące polecenia:
`use "`*nazwa_pliku*`";` – wczytuje i interpretuje deklaracje zawarte w pliku „*nazwa_pliku*”, tak jak gdyby użytkownik wprowadził je z klawiatury;
`show ;` – wyświetla wszystkie nazwy typów danych zdefiniowane w środowisku;
`show` *ident* `;` – wyświetla informację o typie *ident*;
`norm` *term* `;` – wyświetla *term* w postaci normalnej;
`exit ;` – kończy działanie programu.

4.1.1. Typy wbudowane

Jedynym typem, który musi być wbudowany w ET jest typ funkcyjny „`->`”. Względy praktyczne powodują, że w systemie zdefiniowane są dodatkowe typy wraz z odpowiednimi termami. Jednym z takich względów jest większa czytelność infiksowych konstruktorów, np. konstruktora pary „`,`”. Wystarczy porównać zapis `((a,b),c)` z `(Pair (Pair a b) c)`. Oba definiują parę złożoną z pary elementów `a` i `b`, i elementu `c`, ale pierwszy jest czytelniejszy. Inny powodem wbudowania typów jest chęć używania identyfikatorów, których użytkownik ET nie może zdefiniować, np. „`()`” jest konstruktorem typu `UNIT`, a konstruktory definiowane przez użytkownika w ET muszą rozpoczynać się wielką literą. Kolejną przyczyną jest konieczność posiadania liczb naturalnych zaimplementowanych w efektywny sposób.

Poniżej przedstawione są wszystkie typy wbudowane w ET. Każdej definicji typu towarzyszą typy konstruktorów wygenerowane przez system na podstawie definicji (oznaczone przez `con`), nazwa i typ iteratora (oznaczone przez `iter`) oraz reguły obliczania (oznaczone przez `comp`). Dodatkowo znak „+” poprzedza część zawierającą definicję (iterator i reguły obliczania są generowane przez system). Należy zwrócić uwagę, że przedstawione poniżej definicje nie są poprawne w ET jedynie z powodu użycia niedozwolonych identyfikatorów. Użytkownik może zdefiniować w systemie własne typy odpowiadające wbudowanym (np. `datatype Absurd = ;` odpowiada `datatype {} = ;`).

```
+ datatype {} = ;
iter case0 : {} -> 'a
```

```
+ datatype UNIT = () ;
con () : UNIT
iter case1 : UNIT -> 'a -> 'a
comp case1 () = fn v1 => v1
```

```
datatype BOOL = True | False ;
con True : BOOL
con False : BOOL
iter IF : BOOL -> 'a -> 'a -> 'a
comp IF True = fn v1 v2 => v1
comp IF False = fn v1 v2 => v2
```

Konstrukcja `if` *term1* `then` *term2* `else` *term3* jest tłumaczona w `IF` *term1 term2 term3*. Dodatkowo zdefiniowany jest operator infiksowy `= :  'a -> 'a -> BOOL`, który sprawdza czy argumenty są w relacji równości, która została zdefiniowana przy pomocy reguł wnioskowania w definicji 3.7 i w podrozdziałach 3.2.1 i 3.3.1. Termy są w relacji równości, jeśli ich postacie normalne są identyczne z dokładnością do nazw zmiennych związanych. Dzięki własności mocnej normalizacji i własności Churcha–Rossera równość jest rozstrzygalna w przypadku termów zamkniętych (wtedy operator zwraca `True` lub `False`). W przypadku porównywania termów otwartych, czyli termów, w których występują nie związane zmienne, operator równości może się nie zredukować do wartości `True` albo `False`, np. `fn a b=> a=b` redukuje się do `fn a b=>(a=b)` (bo to, czy `a` jest równe `b` zależy od wartości, które będą za te zmienne podstawione), ale `fn a b=>(a=a)` zredukuje się do `fn a b => True`.

Należy zwrócić uwagę, że konstruktory typów „+”, „*” i konstruktor pary „`,`” są infiksowe.

```
+ datatype + 'a 'b = Inl from 'a | Inr from 'b ;
con Inl : 'a -> ('a + 'b)
con Inr : 'b -> ('a + 'b)
iter when : ('a + 'b) -> ('a -> 'c) -> ('b -> 'c) -> 'c
comp when (Inl u1) = fn v1 v2 => v1 u1
comp when (Inr u1) = fn v1 v2 => v2 u1
```

```
+ datatype * 'a 'b = , from 'a 'b ;
con , : 'a -> 'b -> ('a * 'b)
iter split : ('a * 'b)  -> ('a -> 'b -> 'c) -> 'c
comp split (u1 , u2) = fn v1 => v1 u1 u2
```

Typ „+” odpowiada sumie rozłącznej, a typ „*” – parze. Dodatkowo zdefiniowane są funkcje:
`val fst = fn p => split p fn a b => a :  ('a * 'b) -> 'a`
`val snd = fn p => split p fn a b => b :  ('a * 'b) -> 'b`.
Łatwo sprawdzić (spoglądając na reguły obliczania), że funkcje zwracają odpowiednio pierwszy i drugi element pary, `fst (a,b)` = `a` i `snd (a,b)` = `b`.

Liczby naturalne

Ze względu na efektywność obliczeń i łatwość użycia wbudowano także liczby naturalne. Należy zwrócić uwagę na obecność rekursora. Poprzednio omawiane typy danych nie były indukcyjne, co powoduje, że rekursor nie różni się od iteratora. Liczby naturalne są typem indukcyjnym (argumentem konstruktora `Suc` jest liczna naturalna), więc rekursor jest inną funkcją niż iterator. Rekursory są ogólniejsze od iteratorów, tzn. pozwalają na wyrażenie większej ilości algorytmów (np. przy pomocy rekursora można zdefiniować poprzednik o stałej złożoności, a poprzednik zdefiniowany przy pomocy iteratora ma złożoność liniową), ale nie pozwalają na wyrażenie większej ilości funkcji.

```
+ datatype NAT = Suc from NAT | 0;
con Suc : NAT -> NAT
con 0 : NAT
iter _NATit : NAT -> ('a -> 'a) -> 'a -> 'a
comp _NATit (Suc u1) = fn v1 v2 => v1 (_NATit u1 v1 v2)
comp _NATit 0 = fn v1 v2 => v2
rec  _NATrec : NAT -> ((NAT * 'a) -> 'a) -> 'a -> 'a
comp _NATrec (Suc u1) = fn v1 v2 => v1 (u1 , (_NATrec u1 v1 v2))
comp _NATrec 0 = fn v1 v2 => v2
```

Ponieważ wprowadzanie liczb w postaci ciągu następników jest niewygodne, to system pozwala na wprowadzanie liczb w systemie dziesiętnym np. `3` jest skrótem `Suc (Suc (Suc 0))`. Jeśli liczba naturalna zostanie wprowadzona w postaci ciągu następników, to system wyświetli ją w systemie dziesiętnym.

W celu zwiększenia efektywności zostały wbudowane operacje dodawania, mnożenia, obliczania poprzednika i odejmowania. Ich definicje teoretyczne w ET wyglądają następująco.

```
val add = fn n m => _NATit n Suc m : NAT -> NAT -> NAT
val mult = fn n m => _NATit n (add m) 0 : NAT -> NAT -> NAT
val pred = fn n => _NATrec n fst 0 : NAT -> NAT
val sub = fn n m => _NATit m pred n : NAT -> NAT -> NAT
```

Funkcja `pred` wymaga użycia rekursora. Definicja poprzednika przy pomocy iteratora jest możliwa, ale tak zdefiniowana funkcja ma złożoność liniową i traci swoje własności równościowe[3]. Funkcja `add` zdefiniowana przez użytkownika wymaga wykonania n iteracji przy dodawaniu n do m. Wbudowana funkcja `add` wykorzystuje operację dodawanie w SML-u i wymaga tylko jednej iteracji. Należy podkreślić, że pomimo zastosowania efektywnych metod obliczania sumy i iloczynu liczb, zdefiniowane funkcje nie tracą swoich własności teoretycznych, np. można sprawdzić, że `add 0 n = n`. Dodatkowo funkcje wbudowane w wielu przypadkach muszą wykonywać się według definicji, np. wyrażenie `add 30 20` zostanie policzone w sposób efektywny, ale wyrażenie `fn n => add (Suc n) 5` zredukuje się do `fn n => Suc (add n 5)` zgodnie z definicją teoretyczną. Oczywiście po podstawieniu za `n` liczby naturalnej, `add n 5` zostanie wyliczone w sposób efektywny.

Konieczność zachowania własności liczb naturalnych (a co za tym idzie funkcji zdefiniowanych przy pomocy typu `NAT`) powoduje, że ilość pamięci wymagana do operacji na liczbach może liniowo zależeć od wielkości liczb, podobnie jak dla innych typów w ET. Z tego powodu używanie liczb większych od 10000 jest na średniej klasy komputerach kłopotliwe, tzn. może powodować intensywne stronicowanie pamięci. Ponieważ ET nie jest językiem używanym do praktycznej realizacji programów, to ograniczenie takie nie jest dokuczliwe.

[3]Przykłady odpowiednich równości znajdują się w następnym podrozdziale.

4.2. Przykłady użycia ET

W niniejszym podrozdziale przedstawione zostaną przykłady prostych typów i funkcji, które można zdefiniować w ET.

4.2.1. Kombinatory S,K,I

Poniżej przedstawiona jest definicja trzech standardowych kombinatorów `s`,`k` oraz `i`.

```
+ val s = fn f g x => f x (g x);
val s : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
+ val k = fn x y => x;
val k : 'a -> 'b -> 'a
+ val i = fn z => z;
val i : 'a -> 'a
```

Po każdej definicji system odpowiada wyświetlając nazwę identyfikatora i jego typ. Jak wiadomo `s k k` odpowiada kombinatorowi `i`.

```
+ norm s k k;
fn x => x : 'a -> 'a
```

Co więcej, ponieważ w ET równość termów jest rozstrzygalna, to można sprawdzić, że `s k k=i`.

```
+ s k k = i;
val it = True : BOOL
```

4.2.2. Intuicjonistyczny rachunek zdań

W systemie ET można wyrazić intuicjonistyczny rachunek zdań przedstawiony w podrozdziale 2.6.

Pamiętając, że konstruktory typów danych odpowiadają regułom wprowadzania, łatwo znaleźć definicje typów odpowiadających odpowiednim spójnikom. Dla operatora alternatywy istnieją dwie reguły wprowadzania, każda z jedną przesłanką, więc typ będzie posiadał dwa konstruktory jednoargumentowe. Dla wprowadzonego typu system automatycznie generuje iterator, rekursor i odpowiednie reguły obliczania.

```
+ datatype OR 'a 'b = Inl_ from 'a | Inr_ from 'b;
con Inl_ : 'a -> (OR 'a 'b)
con Inr_ : 'b -> (OR 'a 'b)
iter _ORit : (OR 'a 'b) -> ('a -> 'c) -> ('b -> 'c) -> 'c
comp _ORit (Inl_ u1) = fn v1 v2 => v1 u1
comp _ORit (Inr_ u1) = fn v1 v2 => v2 u1
_ORrec = _ORit
```

Należy zwrócić uwagę na podobieństwo konstruktorów i iteratora do reguł wprowadzania i eliminacji w systemie dedukcji naturalnej i do definicji operatora $\vee$ w rachunku $\lambda^{\to,\vee,\wedge,\bot}$ (definicja 2.39).

Można sprawdzić, że tak zdefiniowany operator zachowuje relację redukcji w systemie dedukcji naturalnej zdefiniowaną w definicji 2.36, a także jest zgodny z regułami wprowadzającymi relację równości dla rachunku $\lambda^{\to,\vee,\wedge,\bot}$ (definicja 2.39).

```
+ fn a b c => _ORit (Inl_ a) b c = b a;
val it = fn a b c => True : 'a -> ('a -> 'b) -> ('c -> 'b) -> BOOL
+ fn a b c => _ORit (Inr_ a) b c = c a;
val it = fn a b c => True : 'a -> ('b -> 'c) -> ('a -> 'c) -> BOOL
```

Trzeba podkreślić, że rozstrzyganie podobnych równości nie jest możliwe w żadnym powszechnie używanym języku programowania. Powyższe równości są spełnione dla dowolnych wartości zmiennych a, b i c. Rozstrzygalność równości w systemie ET wynika z mocnej normalizacji termów. Obie strony równości można zredukować do postaci normalnych i porównać. Każdy term posiada jedyną postać normalną.

Typ odpowiadający operatorowi $\vee$ – sumie rozłącznej został w ET wbudowany.

```
+ datatype + 'a 'b = Inl from 'a | Inr from 'b ;
con Inl : 'a -> ('a + 'b)
con Inr : 'b -> ('a + 'b)
iter when : ('a + 'b) -> ('a -> 'c) -> ('b -> 'c) -> 'c
comp when (Inl u1) = fn v1 v2 => v1 u1
comp when (Inr u1) = fn v1 v2 => v2 u1
```

Wbudowany jest w ET także typ odpowiadający operatorowi $\wedge$, czyli para (definicja znajduje się w podrozdziale 4.1.1).

Definicja funkcji przekształcająca sumę rozłączną par w parę sum rozłącznych przedstawiona jest poniżej.

```
+ val proof = fn z => when z (fn x => Inl (fst x),Inl (snd x))
=                            (fn y => Inr (fst y),Inr (snd y));
val proof : (('a * 'b) + ('c * 'd)) -> (('a + 'c) * ('b + 'd))
```

Typ funkcji `proof (('a * 'b) + ('c * 'd)) -> (('a + 'c) * ('b + 'd))` odpowiada wyrażeniu $((a \wedge b) \vee (c \wedge d)) \to ((a \vee c) \wedge (b \vee d))$. Patrząc przez izomorfizm Curry'ego-Howarda (podrozdział 2.8) `proof` jest dowodem formuły $((a \wedge b) \vee (c \wedge d)) \to ((a \vee c) \wedge (b \vee d))$ w systemie dedukcji naturalnej, co łatwo spostrzec porównując go z termem znajdującym się w korzeniu drzewa w przykładzie 2.41.

4.2.3. Liczby naturalne

Liczby naturalne zostały wbudowane.

```
+ datatype NAT = Suc from NAT | 0;
con Suc : NAT -> NAT
con 0 : NAT
iter _NATit : NAT -> ('a -> 'a) -> 'a -> 'a
comp _NATit (Suc u1) = fn v1 v2 => v1 (_NATit u1 v1 v2)
comp _NATit 0 = fn v1 v2 => v2
rec  _NATrec : NAT -> ((NAT * 'a) -> 'a) -> 'a -> 'a
comp _NATrec (Suc u1) = fn v1 v2 => v1 (u1 , (_NATrec u1 v1 v2))
comp _NATrec 0 = fn v1 v2 => v2
```

Rozstrzyganie równości w systemie ET można wykorzystać do weryfikacji specyfikacji funkcji. Dodawanie można zdefiniować przez indukcję po pierwszym argumencie, przy pomocy układu równości:

```
add  0      m = m
add (Suc n) m = Suc (add n m)
```

Funkcja `add` jest wbudowana.

```
val add = fn n => _NATit n Suc : NAT -> NAT -> NAT
```

System ET pozwala sprawdzić, że funkcja spełnia powyższe równania.

```
+ fn m => add  0 m = m;
val it = fn m => True : NAT -> BOOL
+ fn n m => add (Suc n) m = Suc (add n m);
val it = fn n m => True : NAT -> NAT -> BOOL
```

Dzięki temu, w systemie ET można dowieść, że funkcja spełnia równościową specyfikację, a więc jest zdefiniowana zgodnie z zamierzeniami użytkownika.

Trzeba w tym miejscu podkreślić wyjątkowe możliwości systemu ET. Powyższe równości są spełnione dla wszystkich liczb naturalnych. W językach nie posiadających własności teoretycznych ET, takie równości można rozstrzygać tylko dla konkretnych liczb, a więc automatyczne sprawdzenie czy funkcja spełnia specyfikację wymagałoby sprawdzenia nieskończonej ilości równości. Alternatywą jest ręczne dowodzenie własności programu.

Mnożenie można zdefiniować przez parę równań.

```
mult  0      m = 0
mult (Suc n) m = add n (mult n m)
```

Wbudowana funkcja `mult` spełnia powyższą specyfikację.

```
+ fn m => mult 0 m = 0;
val it = fn m => True : NAT -> BOOL
+ fn n m => mult (Suc n) m = add m (mult n m);
val it = fn n m => True : NAT -> NAT -> BOOL
```

Funkcję `le`, która zwraca `True` jeśli pierwszy argument nie jest większy od drugiego, definiują równania.

```
le  0       v       = True
le (Suc u)  0       = False
le (Suc u) (Suc v)  = le u v
```

W ET funkcję `le` można zdefiniować następująco (`ifZero` zwraca drugi argument jeśli pierwszy był zerem, lub trzeci w przeciwnym przypadku).

```
+ val ifZero = fn n a b => _NATit n (fn x => b) a;
val ifZero : NAT -> 'a -> 'a -> 'a
+ val le = fn u => _NATit u (fn y v => ifZero v False (y (pred v)))
=                           (fn v => True);
val le : NAT -> NAT -> BOOL
```

`le` spełnia swoją specyfikację.

```
+ fn v => le 0 v = True;
val it = fn v => True : NAT -> BOOL
+ fn u => le (Suc u) 0 = False;
val it = fn u => True : NAT -> BOOL
+ fn u v => le (Suc u) (Suc v) = le u v;
val it = fn u v => True : NAT -> NAT -> BOOL
```

4.2.4. Listy

W przykładzie 3.2 została zdefiniowana lista polimorficzna. W składni języka ET definicja listy wygląda następująco.

```
+ datatype LIST 'a = Cons from 'a (LIST 'a) | Nil;
con Cons : 'a -> (LIST 'a) -> (LIST 'a)
con Nil : LIST 'a
iter _LISTit : (LIST 'a) -> ('a -> 'b -> 'b) -> 'b -> 'b
comp _LISTit (Cons u1 u2) = fn v1 v2 => v1 u1 (_LISTit u2 v1 v2)
comp _LISTit Nil = fn v1 v2 => v2
rec  _LISTrec : (LIST 'a) -> ('a -> ((LIST 'a) * 'b) -> 'b) -> 'b -> 'b
comp _LISTrec (Cons u1 u2) = fn v1 v2 => v1 u1 (u2 , (_LISTrec u2 v1 v2))
comp _LISTrec Nil = fn v1 v2 => v2
```

Funkcja `map` aplikuje pierwszy argument do wszystkich elementów listy, która jest drugim argumentem. Specyfikacja została podana w przykładzie 3.2. Definicja `map` wygląda następująco.

```
+ val map = fn fun list => _LISTit list (fn head tail => Cons (fun head) tail) Nil;
val map : ('a -> 'b) -> (LIST 'a) -> (LIST 'b)
```

ET pozwala automatycznie sprawdzić, że funkcja spełnia specyfikację.

```
+ fn f => map f Nil = Nil;
val it = fn f => True : ('a -> 'b) -> BOOL
+ fn f head tail => map f (Cons head tail) = Cons (f head) (map f tail);
val it = fn f head tail => True : ('a -> 'b) -> 'a -> (LIST 'a) -> BOOL
```

W następującym przykładzie 6 zostaje dodane do każdego elementu listy złożonej z 3,5 i 0.

```
+ norm map (add 6) (Cons 3 (Cons 5 (Cons 0 Nil)));
Cons (9) (Cons (11) (Cons (6) Nil)) : LIST NAT
```

Funkcja append łączy dwie listy. Jej specyfikacją jest para równań.

```
append  Nil              list = list
append (Cons head tail) list = Cons head (append tail list)
```

Definicja append wygląda następująco.

```
+ val append = fn list1 list2 => _LISTit list1 Cons list2;
val append : (LIST 'a) -> (LIST 'a) -> (LIST 'a)
```

Funkcja append spełnia swoją specyfikację.

```
+ fn list => append Nil list = list;
val it = fn list => True : (LIST 'b) -> BOOL
+ fn list head tail => append (Cons head tail) list = Cons head (append tail list);
val it = fn list head tail => True : (LIST 'b) -> 'b -> (LIST 'b) -> BOOL
```

4.2.5. Strumienie

Rzadko spotykaną w innych językach programowania konstrukcją są kotypy. Kotypy mogą być użyte do reprezentowania potencjalnie nieskończonych typów danych. Przykładem będą strumienie, czyli nieskończone listy. Kotypy definiuje się podając listę destruktorów. System generuje koiterator, korekursor i reguły obliczania. Destruktory odpowiadają regułom eliminacji, a koiterator regule wprowadzania. Kotypy mają jedną regułę wprowadzania i dowolną ilość reguł eliminacji (typy mają dowolną ilość reguł wprowadzania i jedną eliminacji).

W przykładzie 3.3 został zdefiniowany polimorficzny strumień. W składni języka ET ta definicja przedstawiona jest poniżej.

```
+ codatatype STREAM 'a = Shd to 'a & Stl to STREAM 'a;
des Shd : (STREAM 'a) -> 'a
des Stl : (STREAM 'a) -> (STREAM 'a)
coiter _STREAMci : ('b -> 'a) -> ('b -> 'b) -> 'b -> (STREAM 'a)
comp Shd (_STREAMci v1 v2 u) = v1 u
comp Stl (_STREAMci v1 v2 u) = _STREAMci v1 v2 (v2 u)
corec _STREAMcr : ('b -> 'a) -> ('b -> ((STREAM 'a) + 'b)) -> 'b -> (STREAM 'a)
comp Shd (_STREAMcr v1 v2 u) = v1 u
comp Stl (_STREAMcr v1 v2 u) = when (v2 u) (fn x => x) (_STREAMcr v1 v2)
```

Funkcja `getNth` zwraca n-ty element strumienia.

```
+ val getNth = fn stream number => Shd (_NATit number Stl stream);
val getNth : (STREAM 'a) -> NAT -> 'a
```

`natstr` (zdefiniowany w przykładzie 3.4) jest strumieniem kolejnych liczb naturalnych. Poniżej przedstawiona jest specyfikacja i definicja `natstr`.

`Shd (natstr n) = n;`
`Stl (natstr n) = natstr (Suc n);`

```
+ val natstr = _STREAMci (fn x => x) Suc;
val natstr : NAT -> (STREAM NAT)
```

`natstr` spełnia specyfikację.

```
+ fn n => Shd (natstr n) = n;
val it = fn n => True : NAT -> BOOL
+ fn n => Stl (natstr n) = natstr (Suc n);
val it = fn n => True : NAT -> BOOL
```

Można sprawdzić, że

```
+ norm getNth (natstr 0) 40;
40 : NAT
+ 50 = getNth (natstr 0) 50;
True : BOOL
```

ale niestety w ET nie można tej równości dowieść automatycznie dla wszystkich n.

```
+ fn n => n = getNth (natstr 0) n;
val it =
  fn n => n = (Shd (_NATit n Stl (_STREAMci (fn x => x) Suc 0))) :
  NAT -> BOOL
```

Ciekawą funkcją jest `smap` — funkcja aplikująca swój argument do wszystkich (nieskończenie wielu) elementów strumienia. Jej definicja i specyfikacja została podana w przykładzie 3.4.

```
Shd (smap f s) = f (Shd s);
Stl (smap f s) = smap f (Stl s);
```

```
+ val smap = fn fun str => _STREAMci (fn x=> fun (Shd x)) Stl str;
val smap : ('a -> 'b) -> (STREAM 'a) -> (STREAM 'b)
```

System ET pozwala na automatyczne sprawdzenie, że funkcja `smap` spełnia swoją specyfikację.

```
+ fn f s => Shd (smap f s) = f (Shd s);
val it = fn f s => True : ('a -> 'b) -> (STREAM 'a) -> BOOL
+ fn f s => Stl (smap f s) = smap f (Stl s);
val it = fn f s => True : ('a -> 'b) -> (STREAM 'a) -> BOOL
```

`sqrstr` jest strumieniem kolejnych kwadratów liczb naturalnych.

```
+ val sqrstr = smap (fn x => mult x x) (natstr 0);
val sqrstr : STREAM (NAT)
+ getNth sqrstr 40;
val it = 1600 : NAT
```

4.2.6. Wieże Hanoi

Brak możliwości użycia ogólnej rekursji (użycia definiowanej funkcji w jej definicji) jest największym ograniczeniem ET w porównaniu do SML-a. Mimo to ET jest wystarczająco bogaty, by wyrażać wiele algorytmów, które zazwyczaj używają ogólnej rekursji. Przykładem będzie algorytm rozwiązujący problem wież Hanoi.

Problem wież Hanoi polega na przełożeniu n krążków z wieży `Source` na wieżę `Dest` przy pomocy wieży `Aux`. Początkowo krążki są ułożone tak, że największy jest na dole wieży `Source`, a najmniejszy na górze. W każdym ruchu można przełożyć jeden krążek z jednej wieży na drugą, pamiętając o zasadzie, że większy krążek nie może leżeć na mniejszym.

Elementy typu `TOWER` reprezentują wieże.

```
+ datatype TOWER = Source | Dest | Aux;
con Source : TOWER
con Dest : TOWER
con Aux : TOWER
iter _TOWERit : TOWER -> 'a -> 'a -> 'a -> 'a
comp _TOWERit Source = fn v1 v2 v3 => v1
comp _TOWERit Dest = fn v1 v2 v3 => v2
comp _TOWERit Aux = fn v1 v2 v3 => v3
_TOWERrec = _TOWERit
```

Funkcja `mkMove` tworzy parę reprezentującą ruch z `from_` do `to_`.

```
+ val mkMove = fn from_ to_ => (from_,to_);
val mkMove : 'a -> 'b -> ('a * 'b)
```

Funkcja `subTower` tworzy podstawienie (funkcję zamieniającą wieże), np. `subTower s d a` jest funkcją, która zaaplikowana do wieży `Source` zwróci `s`, dla `Dest` zwróci `d`, a dla `Aux` – `a`. Funkcja `appSubToList` aplikuje podstawienie w liście ruchów.

```
+ val subTower = fn s d a tower => _TOWERit tower s d a;
val subTower : 'a -> 'a -> 'a -> TOWER -> 'a
+ val appSubToList = fn sub list => _LISTit list
=                    (fn m => Cons (mkMove (sub (fst m)) (sub (snd m)))) Nil;
val appSubToList : ('a -> 'b) -> (LIST ('a * 'a)) -> (LIST ('b * 'b))
```

Rozwiązanie dla problemu n+1 wież jest konstruowane z rozwiązania dla n wież następująco:

1. należy przełożyć n krążków z wieży `Source` na wieżę `Aux` przy pomocy wieży `Dest` (do rozwiązania problemu dla n wież trzeba zaaplikować podstawienie `subTower Source Aux Dest`);
2. należy przełożyć ostatni krążek z wieży `Source` na wieżę `Dest`;
3. należy przełożyć n krążków z wieży `Aux` na wieżę `Dest` przy pomocy wieży `Aux` (do rozwiązania problemu dla n wież trzeba zaaplikować podstawienie `subTower Aux Dest Source`).

Funkcja `hanoi n` zwraca listę ruchów, będących rozwiązaniem problemu dla `n` wież.

```
val hanoi = fn number => _NATit number
=  (fn x => append (appSubToList (subTower Source Aux Dest) x)
=             (Cons (mkMove Source Dest)
=                   (appSubToList (subTower Aux Dest Source) x))) Nil;
val hanoi : NAT -> (LIST (TOWER * TOWER))

+ norm hanoi 2;
Cons (Source , Aux) (Cons (Source , Dest) (Cons (Aux , Dest) Nil)) :
LIST (TOWER * TOWER)
+ norm hanoi 3;
Cons (Source , Dest)
  (Cons (Source , Aux)
     (Cons (Dest , Aux)
        (Cons (Source , Dest)
           (Cons (Aux , Source)
              (Cons (Aux , Dest) (Cons (Source , Dest) Nil)))))) :
LIST (TOWER * TOWER)
```

Co ciekawe, można udowodnić, że funkcja `hanoi` buduje rozwiązanie w sposób opisany powyżej.

```
+ fn n => hanoi (Suc n) =
=            append (appSubToList (subTower Source Aux Dest) (hanoi n))
=            (Cons (mkMove Source Dest)
=                  (appSubToList (subTower Aux Dest Source) (hanoi n)));
val it = fn n => True : NAT -> BOOL
```

Zdefiniowanie funkcji rozwiązującej problem wież Hanoi w ET sugeruje, że także inne funkcje, które zazwyczaj są przedstawiane w formie używającej ogólnej rekursji, można w ET zdefiniować. Brak ogólnej rekursji nie jest więc istotnym ograniczeniem dla ET. Jednym z celów implementacji ET jest stworzenie narzędzia, które pozwoliłoby eksperymentować z nowym stylem programowania.

4.3. Dowodzenie równości za pomocą reguły jednoznaczności

Relacja równości w systemie ET została zdefiniowana przy pomocy reguł wnioskowania w definicji 3.7 i w podrozdziałach 3.2.1 i 3.3.1. Własność mocnej normalizacji powoduje, że równość termów jest rozstrzygalna. Odpowiedni operator = został zaimplementowany w języku, co umożliwia automatyczne sprawdzanie własności programów.

Relacja równoważności nie pozwala na udowodnienie wielu ciekawych własności termów w ET. W szczególności funkcje, które zostały oparte na różnych algorytmach są w ET różne, mimo że zwracają te same wyniki. W [11] zostały zdefiniowane reguły jednoznaczności rozszerzające relację równości. Rozszerzona relacja równości jest nierozstrzygalna, tzn. nie może być użyta do automatycznego sprawdzania równości termów.

Dla liczb naturalnych reguła jednoznaczności wygląda następująco.

$$\frac{\begin{array}{c}\Gamma, x : \mathtt{NAT} \rhd F(\mathtt{Suc}\ x) \ = \ N_1(F\ x) : \beta \\ \Gamma, x : \mathtt{NAT} \rhd G(\mathtt{Suc}\ x) \ = \ N_1(G\ x) : \beta \\ \Gamma \rhd F\ \mathtt{0} \ = \ N_2 : \beta \\ \Gamma \rhd G\ \mathtt{0} \ = \ N_2 : \beta\end{array}}{\Gamma \rhd F \ = \ G \ : \ \mathtt{NAT} \to \beta}\ U\ \mathtt{NAT}$$

Poniżej przedstawione są dwie różne definicje tej samej funkcji. Obie zwracają takie same wyniki, ale zbudowane są przy użyciu innych algorytmów. Wbudowana operacja dodawania zdefiniowana jest następująco.

```
+ val add = fn n => _NATit n Suc;
val add : NAT -> NAT -> NAT
```

Dodawanie można także zdefiniować inaczej.

```
+ val add1 = fn m => _NATit m (fn x y => Suc (x y)) (fn n => n);
val add1 : NAT -> NAT -> NAT
```

Obie funkcje spełniają tę samą specyfikację.

```
add  0       m = m
add (Suc n) m = Suc (add n m)
```

```
+ fn m => add 0 m = m;
val it = fn m => True : NAT -> BOOL
+ fn n m => add (Suc n) m = Suc (add n m);
val it = fn n m => True : NAT -> NAT -> BOOL

+ fn m => add1 0 m = m;
val it = fn m => True : NAT -> BOOL
+ fn n m => add1 (Suc n) m = Suc (add1 n m);
val it = fn n m => True : NAT -> NAT -> BOOL
```

Jednak w ET bez zastosowania reguły jednoznaczności nie można udowodnić równości funkcji.

```
+ add = add1;
False : BOOL
```

Poniżej przedstawiony jest dowód równości funkcji `add` i `add1` (pominięto typy). Przesłanki łatwo udowodnić (nawet automatycznie) używając reguł obliczania.

$$\frac{\begin{array}{rcl} \texttt{add}(\texttt{Suc}\ m) & = & (\lambda x\ n.\texttt{Suc}\ (x\ n))\ (\texttt{add}\ m) \\ \texttt{add1}(\texttt{Suc}\ m) & = & (\lambda x\ n.\texttt{Suc}\ (x\ n))\ (\texttt{add1}\ m) \\ & \texttt{add}\ 0 = \lambda n.n & \\ & \texttt{add1}\ 0 = \lambda n.n & \end{array}}{\texttt{add} = \texttt{add1}}\ U\ \texttt{NAT}$$

W [11] przedstawione są schematy pozwalające generować reguły jednoznaczności dla wszystkich typów i kotypów.

5. Implementacja

Częścią pracy jest implementacja języka ET[1]. Implementacja nie jest użyteczna jako język programowania do celów praktycznych, ale umożliwia łatwe i efektywne korzystanie z systemu ET w celu poznania jego własności i możliwości.

Zaimplementowanie języka programowania niesie ze sobą wiele problemów natury technicznej związanych z efektywną i jednocześnie czytelną realizacją kolejnych etapów kompilacji ET. Niektóre z nich, zwłaszcza te mające solidne podstawy teoretyczne, zostały opisane w niniejszym rozdziale. Kod źródłowy powstał w języku Standard ML z uwagi na łatwość programowania, czytelność kodu, charakter języka zbliżony do ET i przenośność na wiele platform bez potrzeby zmiany kodu (win32 i wiele systemów typu UNIX). Implementacja została wykonana w sposób pozwalający na łatwe rozszerzanie systemu o nowe własności, co jest bardzo pożądane ze względu na eksperymentalny charakter ET i możliwości jego rozwoju.

Opis poszczególnych etapów kompilacji jest dość pobieżny z uwagi na charakter pracy. Zwrócono uwagę na najważniejsze i najciekawsze problemy związane z implementacją ET. Przedstawione algorytmy są uproszczonymi wersjami algorytmów zastosowanych w kodzie programu. Uproszczenia polegają na usunięciu szczegółów technicznych, które zmniejszają czytelność i utrudniają zrozumienie zasady działania algorytmów.

Informacje na temat konstrukcji kompilatorów języków funkcjonalnych znajdują się w [1], [3] i [4].

5.1. Kod źródłowy

Implementacja została wykonana przy użyciu kompilatora języka SML. Standard ML jest polimorficznym językiem funkcyjnym wyposażonym w moduły, kontrolę i wyprowadzanie typów (opis SML-a znajduje się w [8]). Wykorzystano narzędzia i biblioteki zawarte w SML of New Jersey (Standard ML of New Jersey, Version 110.0.3, January 30, 1998), co może być powodem niewielkich trudności w skompilowaniu kodu przy użyciu innego kompilatora. Kod został napisany z troską o maksymalną zgodność z definicją języka SML'97 i biblioteką SML Basis Library. Jedynym źródłem ewentualnego braku kompatybilności z innymi kompilatorami SML-a może być wykorzystanie narzędzi ML-Lex i ML-Yacc, jak również modułu PrettyPrint.

Kod źródłowy ET został podzielony na moduły zawierające pojedyncze składniki implementacji. Każdy ze składników posiada interfejs modułu, która określa jakie typy i funkcje muszą być w module zdefiniowane i jednocześnie ukrywa, w jaki sposób zostały one zaimplementowane. Za przykład może posłużyć moduł definiujący środowisko, czyli zbiór wprowadzonych deklaracji. Żaden z modułów nie posiada informacji, czy środowisko zostało zaimplementowane przy pomocy list czy drzew binarnych. Z punktu widzenia pozostałych modułów jest to typ abstrakcyjny. Umożliwia to dokonanie nawet poważnych zmian w pojedynczym module, bez ko-

[1]Poprzednia implementacja powstała w 1992 r. Z uwagi na jej nieefektywność i kod źródłowy napisany w starej wersji języka SML przy użyciu niedostępnego już kompilatora, powstała potrzeba ponownej implementacji.

nieczności ingerencji w pozostałe. W SML-u odpowiednikiem modułu jest struktura, a sygnatura jest odpowiednikiem interfejsu modułu.

5.2. Interpreter ET

Program w języku ET jest ciągiem deklaracji. Środowisko jest zbiorem zdefiniowanych wcześniej typów i termów. Przetwarzanie kolejnych deklaracji powoduje zmianę środowiska i wyświetlenie informacji dla użytkownika.

Kolejne fazy przetwarzania prezentuje następujący rysunek.

↓ tekst programu — deklaracji

| **lexer** *analiza leksykalna* |

↓ słowa języka

| **parser** *analiza składniowa* |

↓ drzewo rozbioru

| **elaborator** *kontrola typu* |

↓ postać wewnętrzna

| **evaluator** *wykonanie kodu deklaracji, modyfikacja środowiska* |

| **prettyprinter** *wyświetlenie informacji dla użytkownika* |

Interpreter ET wywołuje na przemian **evaluator** (który wywołuje kolejne moduły przetwarzające program) i **pretty printer**.

5.3. Analiza leksykalna

Program jest wprowadzany do komputera w postaci tekstu. Analiza leksykalna polega na rozpoznaniu w tekście programu pojedynczych słów i przypisaniu ich do odpowiedniej kategorii syntaktycznej np. słowo kluczowe, komentarz, identyfikator. Strumień znaków zostaje zamieniony na strumień słów języka. Analizator leksykalny — lexer został stworzony przy użyciu programu ML-Lex, który jest odpowiednikiem lex-a dla SML-a.

Każdy język jest zbiorem napisów, napis jest skończonym ciągiem słów zbudowanych z symboli ustalonego alfabetu. Zadaniem analizatora leksykalnego jest rozpoznawanie pojedynczych słów języka. Słowa są skończonym ciągiem symboli, ale zbiór słów w języku jest często nieskończony, np. nazwą konstruktora w ET jest dowolny ciąg liter rozpoczynający się wielką literą. Do opisania zbiorów słów można posłużyć się wyrażeniami regularnymi. Wyrażenia regularne zbudowane są przy użyciu pięciu podstawowych operacji:

symbol: jeśli **a** jest symbolem z alfabetu, to wyrażenie regularne **a** reprezentuje zbiór słów złożony z jednego słowa **a**;

alternatywa: jeśli **N** i **M** są wyrażeniami regularnymi, to **N|M** reprezentuje zbiór słów będący sumą zbiorów **N** i **M**;

konkatenacja: jeśli **N** i **M** są wyrażeniami regularnymi, to **NM** reprezentuje zbiór słów, z których każde jest złożone z podsłowa należącego do zbioru **N** i podsłowa należącego do zbioru **M**;

epsilon: ϵ reprezentuje zbiór złożony z pustego słowa;

powtórzenie: jeśli **M** jest wyrażeniem regularnym, to **M*** (dopełnienie Kleenego) reprezentuje zbiór słów złożonych z zera lub więcej podsłów, z których każde należy do zbioru **M**;

Np. wyrażenie **((a|b)a)*)** reprezentuje nieskończony zbiór słów złożonych z symboli „a" i „b",takich że: są złożone z parzystej ilości symboli i na parzystych pozycjach występuje symbol „a" tzn. {ϵ, aa, ba, aaaa, baaa, baba, bababa, baaaaaba,...}. Wyrażenia regularne mogą być rozpoznawane przez deterministyczny automat skończony. Lex przekształca listę wyrażeń regularnych definiujących zbiory słów dopuszczalnych w języku w program, będący implementacją automatu rozpoznającego podane wyrażenia. Odpowiedni algorytm konstrukcji został opracowany w 1975 r. W 1995 r. znaleziono bardziej efektywny sposób kodowania automatu (Flex), co pozwoliło na znaczne ograniczenie wymagań pamięciowych i osiągnięcie szybkości analizatorów pisanych ręcznie. Zaletą stosowania Lex-a jest łatwość zadawania zbiorów słów w postaci wyrażeń i łatwość wprowadzania zmian. Ręcznie zaimplementowany lexer może być szybszy i mieć mniejsze wymagania pamięciowe, ale zmiana jednego ze zbiorów rozpoznawanych wyrazów może powodować konieczność zmiany całego kodu.

Definicja analizatora leksykalnego znajduje się w pliku `parser/et.lex`. ML-Lex tworzy na podstawie tego pliku kod źródłowy analizatora leksykalnego (plik `parser/et.lex.sml`). Definicja składa się z reguł postaci wyrażenie regularne => (kod);. Fragment przedstawiony jest poniżej.

```
"to" => (makeToken Tokens.To yytext currPos duringParsing);
\(   => (makeToken Tokens.LeftParen yytext currPos duringParsing);
\)   => (makeToken Tokens.RightParen yytext currPos duringParsing);
```

Wśród listy wyrażeń regularnych (w podanym fragmencie wyrażenia są proste – odpowiadają słowom „to", „(„ i „)") ML-Lex dopasowuje to, które odpowiada dłuższemu słowu (np. do słowa „tom" zostanie dopasowane wyrażenie definiujące identyfikatory).

Wszystkie reguły w implementacji ET tworzą odpowiedni element typu `Tokens` (np. `Tokens.To`, `Tokens.LeftParen`), który reprezentuje rozpoznane słowo i zawiera informację o jego położeniu w tekście programu, co może być wykorzystane do wygenerowania informacji o błędach.

5.4. Analiza składniowa

Język składa się często z nieskończenie wielu napisów. Wyrażenia regularne nie zapewniają odpowiedniej siły wyrażania dla większości języków (np. nie można wyrazić zbioru słów złożonych z dowolnej ilości nawiasów otwierających i takiej samej ilości nawiasów zamykających tzn. {(), (()), ((())),...}). Formalizmem pozwalającym na zdefiniowanie zbioru dopuszczalnych napisów w wielu językach jest gramatyka bezkontekstowa.

Gramatyka bezkontekstowa składa się ze skończonego zbioru reguł produkcji

$$\text{symbol\_nieterminalny} \rightarrow \text{symbol symbol} \ldots \text{symbol},$$

które oznaczają, że za symbol z lewej strony można podstawić symbole z prawej. Po prawej stronie znajduje się zero lub więcej symboli. Symbole dzielą się na symbole terminalne, które są słowami z języka i nie mogą znajdować się z lewej strony reguł, i symbole nieterminalne, dla których musi istnieć choć jedna reguła produkcji, w której są po lewej stronie. Wyróżniony symbol nieterminalny oznacza początek, tzn. symbol od którego rozpoczyna się stosowanie reguł produkcji, czyli konstrukcja drzewa rozbioru. Drzewo rozbioru reprezentuje składnię programu. Gramatyka jest jednoznaczna jeśli nie istnieją dwa różne drzewa rozbioru dla żadnego z napisów.

Konstrukcja parsera, programu zamieniającego napis, czyli ciąg słów na drzewo rozbioru, jest ogólnie zadaniem trudnym, ale dla gramatyk LR(k) da się zautomatyzować. Gramatyki typu LR(k) (*Left-to-right parse, Rightmost derivation, k token lookahead*) są gramatykami, dla których można skonstruować drzewo analizując słowa od lewej do prawej strony (od pierwszego do ostatniego), rozwijając zawsze symbol nieterminalny z prawej strony i spoglądając na k słów w przód. Gramatyki LR(k) są jednoznaczne. Deterministyczny automat skończony ze stosem pozwala konstruować drzewo rozbioru dla języków definiowanych przez takie gramatyki. Ponieważ liczba stanów w automacie, czyli wymagania pamięciowe parsera, rosną wykładniczo względem (k+1) a większość języków programowania posiada gramatykę LR(1), to yacc (program tworzący parser) działa dla gramatyk LR(1)[2].

Parser został wygenerowany przy użyciu programu ML-Yacc, który jest odpowiednikiem programu yacc dla SML-a.

W pliku `parser/et.grm` znajduje się definicja gramatyki ET w formacie wymaganym przez program ML-Yacc. Zdefiniowane są symbole terminalne, np.

```
%term Fn
    | Val
    | Let | In | End
    | Datatype | From
    | Codatatype | To ...
```

nieterminalne, np.

```
%nonterm declaration          of FromParser.etDeclaration
       | atomicTerm           of FromParser.etTerm
       | parameters           of FromParser.String list
       | term                 of FromParser.etTerm ...
```

deklaracje ustalające kolejność i kierunek wiązania operatorów, np.

```
%left Plus
%left Star
%left Equals ...
```

i reguły produkcji w postaci
symbol_nieterminalny: symbol1 ... symbol2 (kod1) | symbol3 ... symbol4 (kod2) | ...,
co oznacza, że symbol_nieterminalny może zostać przekształcony w symbol1 ... symbol2 i zostanie wykonany kod1, albo w symbol3 ... symbol4 i zostanie wykonany kod2. Przykład reguły produkcji znajduje się poniżej.

```
declaration:
          (FromParser.mkTermWithPos FromParser.Empty () defaultPos defaultPos)
 | valueBinding      (FromParser.mkTermWithPos FromParser.ValBind valueBinding
                       valueBindingleft valueBindingright)
 | term      (FromParser.mkTermWithPos FromParser.Term term termleft termright)
 | datatypeDefinition   (FromParser.mkTermWithPos FromParser.DatatypeDef
           datatypeDefinition datatypeDefinitionleft datatypeDefinitionright)
 | codatatypeDefinition (FromParser.mkTermWithPos FromParser.CodatatypeDef
      codatatypeDefinition codatatypeDefinitionleft codatatypeDefinitionright)...
```

[2]Właściwie yacc wymaga gramatyki typy LALR(1), ale różnica między tymi gramatykami jest trudna do wyjaśnienia bez prezentacji szczegółów technicznych, które można znaleźć w [1].

Kod źródłowy parsera, generowany przez ML-Yacc, znajduje się w plikach `parser/et.grm.sig`, `parser/et.grm.sml`. Sygnatura i kod modułu łączącego lexer i parser znajduje się w pliku `parser/etParser.sml`.

```
signature etPARSER =
sig
structure FromParser : etFROMPARSER

type LexerType

val makeLexer : (int -> string) -> (bool -> unit) -> LexerType
val parse : LexerType -> FromParser.Position.Position ->
   (FromParser.etDeclaration * LexerType * FromParser.Position.Position)
end
```

Funkcja `makeLexer` zaaplikowana do funkcji zwracającej kolejne znaki programu (najczęściej jest to funkcja wczytująca kolejną linię) i funkcji wyświetlającej znak zachęty (argumentem jest `true` dla pierwszej linii i `false` dla pozostałych) zwraca lexer. Funkcja `parse` zaaplikowana do lexera i pozycji w tekście pierwszego słowa znajdującego się w lexerze zwraca krotkę zawierającą rozpoznaną deklarację, lexer i pozycję pierwszego nie analizowanego słowa.

Sygnatura modułu zawierającego definicję typu zwracanego przez parser, czyli drzewo rozbioru programu, znajduje się w pliku `parser/etFromParser.sml`.

```
signature etFROMPARSER=
sig
...
  datatype etTerm =
      Let of (etValBinding list * etTerm) * LRPos
    | Fn of (String list * etTerm )  * LRPos
    | Ident of string  * LRPos
    | App of (etTerm * etTerm) * LRPos
  and etValBinding =
      Val of (String * etTerm) * LRPos
...
  datatype etDeclaration =
      LexerError of string * LRPos
    | ParserError of string * LRPos
    | Eof of unit * LRPos
    | Empty of unit * LRPos
    | Term of etTerm * LRPos
    | ValBind of etValBinding * LRPos
    | DatatypeDef
          of (String * (String list) * (String * etType list) list) * LRPos
    | CodatatypeDef
          of (String * (String list) * (String * etType list) list) * LRPos
...
 end
```

5.5. Kontrola typu

Drzewo rozbioru reprezentuje deklarację zbudowaną poprawnie składniowo. Elaborator wyprowadza i sprawdza typy termów zawartych w deklaracji, po czym generuje reprezentację wewnętrzną i dodaje nową deklarację do środowiska. Środowisko jest zbiorem zdefiniowanych termów wraz z ich typami i jest wykorzystywane do odwoływania się do poprzednio wprowadzonych definicji. W trakcie wyprowadzania i kontroli typu sprawdzane jest istnienie w środowisku wszystkich definicji, do których odwołuje się przetwarzana deklaracja.

5.5.1. Algorytm unifikacji

Podczas wyprowadzanie typu, często zachodzi potrzeba przyrównania dwóch typów. Jeśli typy zawierają zmienne typowe, to szukane jest takie podstawienie (przypisanie zmiennym typowym typów), które spowoduje, że typy będą równe, np. dla typów $a \to a$ i b takie podstawienie istnieje $b := a \to a$, dla typów $c \to c$ i c takiego podstawienia nie ma. Podstawienie jest rozwiązaniem układu równań typowych. Problem równości typów jest w systemie ET rozstrzygalny. Algorytm unifikacji dla zadanego układu równań zwraca najogólniejsze podstawienie, dla którego typy są równe, lub informację, że takie podstawienie nie istnieje. Podstawienie S jest najogólniejsze, jeśli dla każdego podstawienia S_1, które jest rozwiązaniem zadanego układu równości typów, istnieje podstawienie S_2, takie, że S_1 jest równoważne złożeniu podstawień S i S_2. Dla układu równań $a \to b = c, b = a \to d$ rozwiązaniem jest podstawienie $\{b := a \to (a \to a), c := a \to (a \to (a \to a)), d := a \to a\}$. Podstawieniem najogólniejszym jest $\{b := a \to d, c := a \to (a \to d)\}$. Poprzednie podstawienie może być uzyskane z najogólniejszego po zastosowaniu podstawienia $\{d := a \to a\}$.

Poniżej przedstawiony jest algorytm unifikacji dla systemu $\lambda^{\to}$ (dla ET algorytm wymaga uwzględnienia większej ilości konstruktorów typu). Funkcja D zwraca parę różniących się części typów lub π (parę pustą), jeśli typy się nie różnią.

```
D (a,b)          => if a=b then π else (a,b)
D (a→b,c→d)      => if D(a,c)=π then D(b,d) else D(a,c)
D (a,b→c)        => (a,b→c)
D (a→b,c)        => (a→b,c)
```

Funkcja Unify zwraca najogólniejsze podstawienie, doprowadzające do równości typów, albo Fail jeśli takie podstawienie nie istnieje. Argumentem funkcji jest lista par typów, odpowiadająca liście równań między typami. appSub oznacza funkcję aplikującą podstawienie do typu , IdSub podstawienie identycznościowe, a sub w kodzie jest aktualnym podstawieniem (algorytm stopniowo rozszerza podstawienie doprowadzając do rozwiązania). sub[a:=b] oznacza zamianę wszystkich wystąpień zmiennej a na wyrażenie b w podstawieniu sub. Podstawienie jest reprezentowane przez zbiór przypisań postaci zmienna:=typ. Funkcja U zaaplikowana do podstawienia sub i listy par typów zwraca najogólniejsze podstawienie unifikujące pary typów z listy zadanej jako drugi argument i stanowiące rozszerzenie podstawienia sub. Funkcja Unify wywołuje U z początkowym podstawieniem IdSub, które jest stopniowo rozszerzane.

```
U sub {(T1,T2),(T3,T4),...} => if (appS sub T1) = (appS sub T2)
                               then U sub {(T3,T4),...}
                               else let
                                 (Ta,Tb) := D (appSub sub T1,appSub sub T2)
                               in
                                 if Ta jest zmienną, która nie występuje w Tb
                                 then U ({Ta:=Tb} ∪ (sub[Ta:=Tb])) {(T1,T2),(T3,T4),...}
                                 elseif Tb jest zmienną, która nie występuje w Ta
                                 then U ({Tb:=Ta} ∪ (sub[Tb:=Ta])) {(T1,T2),(T3,T4),...}
                                 else Fail
                               end
U sub {}                    => sub
Unify {(T1,T2),(T3,T4),...} => U IdSub {(T1,T2),(T3,T4),...}
```

Algorytm unifikacji znajduje się w pliku `compiler/etUnifier.sml`.

5.5.2. Algorytm rekonstrukcji typu

Zarówno dla systemu $\lambda^{\rightarrow}$ jak i dla system ET problem wyprowadzania typu jest rozstrzygalny. W obu przypadkach można użyć algorytmu W zaproponowanego przez Milnera w latach 70-tych. Algorytm W znajduje najogólniejszy typ termu wyprowadzalny w zadanym środowisku, bądź informuje, że taki typ nie istnieje. Algorytm W przedstawiony jest poniżej. Funkcja W zaaplikowana do środowiska i termu, zwraca parę złożoną z podstawienia i najogólniejszego w zadanym środowisku typu termu. Zmienna env w kodzie algorytmu jest reprezentuje środowisko (zbiór przypisań postaci term:typ).

```
W env x        => (IdSub,σ)   gdzie  x:σ ∈ env
W env (N M)    => let
                    (sub1,typ1) = W env N
                    and (sub2,typ2) = W (appS sub1 env) N
                    and a =nowa zmienna typowa
                    and sub3 = Unify (appS sub2 typ1,typ2→a)
                  in
                    (sub3,appS sub3 a)
                  end
W env (λx.M)   => let
                    a =nowa zmienna typowa
                    and (sub,typ) = W {env,x:a} M
                  in
                    (sub,appS sub (a→typ)
                  end
```

Algorytm wyprowadzania typu znajduje się w pliku compiler/etElab.sml.

Informacje na temat unifikacji i rekonstrukcji typu znajdują się w [4] i [6].

5.6. Wykonanie kodu

Kod programu zapisany jest postaci wewnętrznej. Postać wewnętrzna po pominięciu specjalnych konstrukcji dla typów danych jest termem beztypowego rachunku λ zapisanym przy użyciu notacji de Bruijna, która została omówiona w punkcie 5 w uwagach 2.9. Zmienne związane pamiętane są w postaci numeru, określającego która abstrakcja je wiąże (0 – najbliższa, 1

– poprzednia itd.). Termowi $\lambda y.(\lambda z.\lambda y.yz)y$ odpowiada zapis $\lambda(\lambda\lambda 01)0$. Dla ułatwienia zostanie zaprezentowana ta część implementacji.

```
signature etABSTRACTSYNTAX =
sig
...
  datatype etTerm =
    Var of TermNumberId
  | BoundVar of int
  | App of etTerm * etTerm
  | Abs of string * etTerm
...
end
```

Z powyższej definicji wynika, że term może być zmienną zdefiniowaną w środowisku, zmienną związaną, aplikacją lub abstrakcją. Wykonywanie programu polega na redukowaniu β-redeksów znajdujących się w termie. Za zmienne zdefiniowane w środowisku podstawiana jest ich definicja. Ze względu na własność mocnej normalizacji kolejność redukcji w przypadku ET nie ma żadnego wpływu na wynik działania programu. Termy są redukowane do postaci normalnej. Używana jest leniwa strategia redukcji, tzn. redukowany jest najbardziej zewnętrzny, lewy redeks znajdujący się w termie. Dzięki zastosowaniu notacji de Bruijna β-redukcja nie wymaga zmiany nazw zmiennych. Poniżej przedstawiony jest algorytm redukcji termów. Funkcja Sub realizuje podstawienie termu N za zmienne związane na poziomie m. Funkcja Lift odpowiednio modyfikuje poziom wiązań w podstawianym termie.

```
Lift i m (j)      => if j<i then j else j+m
Lift i m (N M)    => (Lift i m (N))(Lift i m (M))
Lift i m (λ.M)    => λ.(Lift (i+1) m (M))

Sub N m n         => if n<m then n
                     elseif n=m then Lift 0 n (N)
                     else n-1
Sub N m (M P) => (Sub N m M) (Sub N m P)
Sub N m (λ.M) => λ.(Sub N (m+1) M)
```

gdzie β-redukcji odpowiada (λ.M) N => Sub N 0 M

Za przykład działania powyższego algorytmu posłuży normalizacja termu $\lambda y.\underline{(\lambda z.\lambda y.yz)y}$. β-redukcja podkreślonej części prowadzi do postaci $\lambda y.\lambda y'.y'y$. Należy zwrócić uwagę na konieczną zmianę nazw zmiennych (z y na y'). Ta sama redukcja termu w postaci de Bruijna $\lambda\underline{(\lambda\lambda 01)0}$ prowadzi do kolejnych kroków (podkreślono redukowaną część) $\lambda(\underline{\mathsf{Sub}\ 0\ 0\ (\lambda 01)}) \Rightarrow \lambda\lambda(\underline{\mathsf{Sub}\ 0\ 1\ (01)})$ $\Rightarrow \lambda\lambda(\underline{(\mathsf{Sub}\ 0\ 1\ 0)}(\mathsf{Sub}\ 0\ 1\ 1)) \Rightarrow \lambda\lambda(0\ \underline{(\mathsf{Sub}\ 0\ 1\ 1)}) \Rightarrow \lambda\lambda(0\ \underline{(\mathsf{Lift}\ 0\ 1\ 0)}) \Rightarrow \lambda\lambda(0\ 1)$, co odpowiada termowi $\lambda y.\lambda y'.y'y$. Redukcja przy użyciu notacji de Bruijna nie wymaga zmiany nazw zmiennych i dzięki temu jest użyteczna w implementacji.

Algorytm redukcji termów jest zaimplementowany w pliku `syntax/etEval.sml`.

5.7. Odśmiecanie pamięci

W językach, w których deklaracje powodują dokładanie nowych definicji do środowiska, istnieje potrzeba usuwania definicji, które są już niepotrzebne (*garbage collection*). Rozważmy przykład programu.

```
val a = fn x => x;
val a = fn z y => y;
```

Pierwsza deklaracja wprowadzi do środowiska zmienną `a` i przypisze jej wartość `fn x => x`. Druga wprowadzi zmienną `a` i przypisze jej wartość `fn z y => y`. Powoduje to, że funkcja `fn x => x` jest już niewidoczna i można ją usunąć ze środowiska, o ile nie jest wykorzystywana w termach zdefiniowanych w środowisku. Kolejny przykład pokazuje, w jakiej sytuacji powtórne zdefiniowanie zmiennej o tej samej nazwie nie wystarcza, by usunąć poprzednią definicję.

```
val a = fn x => x;
val b = fn x => a;
val a = fn x y => y;
```

Tym razem odwołanie do funkcji `fn x => x` występuje w funkcji `fn x => a`, więc nie może być usunięta, mimo iż jest niedostępna z zewnątrz (nie można się do niej odwołać w nowych definicjach). Usuwaniem zbędnych definicji ze środowiska zajmuje się program czyszczący pamięć (*garbage collector*). W ogólnym przypadku środowisko może być przedstawione w postaci grafu skierowanego, w którego węzłach znajdują się definicje. Z węzła A do węzła B prowadzi krawędź, jeśli term w A odwołuje się do zmiennej zdefiniowanej w B. Zadaniem programu czyszczącego pamięć jest znalezienie spójnego podgrafu, do którego należy ostatnio wprowadzona definicja. W ET nie ma funkcji wzajemnie rekurencyjnych, co powoduje, że w grafie nie ma cykli. Dzięki temu złożoność programu czyszczącego pamięć zależy liniowo od wielkości środowiska i wymaga tylko jednego przejścia po definicjach zawartych w środowisku.

Algorytm odśmiecania pamięci przedstawiony jest poniżej. Parametr defVar jest zbiorem zawierającym zdefiniowane zmienne. Środowisko, przekazywane jako parametr env, jest zbiorem definicji. Algorytm sprawdza, czy zmienne w środowisku są przykryte przez wcześniejsze definicje (zmienne zdefiniowane wcześniej są w zmiennej defVar) i jeśli tak, to ją usuwa. Funkcja GarbageCollector zaaplikowana do środowiska zwraca środowisko po usunięciu zbędnych definicji. Funkcja Gc zaaplikowana do zbioru zmiennych przykrywających definicje w środowisku i środowiska zwraca środowisko.

Gc defVar {x:=N,y:=M,...} =>
  if x∉defVar
  then {x:=N} ∪ (Gc (defVar \ (zmienne termu N zdefiniowane w środowisku)) {y:=M,...})
  else Gc defVars {y:=M,...}
GarbageCollector env => Gc {} env

Program czyszczący pamięć znajduje się w pliku `syntax/etEnvironment.sml`.

6. Podsumowanie

Praca przedstawia system ET, jego własności i możliwość użycia jako języka programowania.

Opisane są podstawowe formalizmy z teorii funkcji i typów, na których oparty jest ET: beztypowy rachunku λ, rachunek λ z typami, system T Gödla i system F Girarda, a także logika intuicjonistyczna i system dedukcji naturalnej.

System ET pozwala na wyrażanie dużej klasy funkcji zachowując własności teoretyczne systemów uboższych, np. własność mocnej normalizacji czy rozstrzygalność typizowania. Związek z logiką intuicjonistyczną i rozstrzygalność równości pozwalają na automatyczne sprawdzanie wielu własności wprowadzanych funkcji.

Wykonana implementacja pozwala na eksperymentowanie z systemem w celu poznania możliwości wykorzystania jego własności teoretycznych i sprawdzania, jakie funkcje można w tym systemie wyrazić.

Otwartym problemem pozostaje charakteryzacja zbioru funkcji wyrażalnych w systemie ET. Górnym ograniczeniem tego zbioru jest zbiór funkcji wyrażalnych w systemie F (wynika to z faktu, że istnieje tłumaczenie systemu ET w system F), a dolnym – zbiór funkcji wyrażalnych w systemie T (ponieważ w ET można zdefiniować liczby naturalne i wartości boolowskie).

Dzięki efektywnemu zaimplementowaniu liczb naturalnych zwiększyła się szybkość wykonywania wielu algorytmów. Umożliwia to praktyczne zaimplementowanie i używanie większej klasy funkcji w ET. Kolejnym krokiem powinno być wbudowanie list. Implementacja została wykonana ze szczególną dbałością o możliwość jej późniejszego rozwoju. Dzięki zastosowaniu narzędzi takich jak ML-Lex i ML-Yacc i podzieleniu kodu na moduły z dobrze określonymi zależnościami, rozszerzanie systemu o nowe własności jest stosunkowo łatwe , co jest bardzo pożądane ze względu na eksperymentalny charakter ET i duże możliwości rozwoju.

Jedną z możliwości rozwinięcia istniejącej implementacji jest dołączenie reguł jednoznaczności, które w szczególnym przypadku zostały opisane w podrozdziale 4.3. W znaczny sposób zwiększyłoby to klasę dowodliwych równości, choć oczywiście wymagałoby udziału użytkownika systemu w procesie dowodzenia.

Bibliografia

[1] A. W. Appel. *Modern Compiler Implementation in ML: Basic Techniques.* Cambridge University Press, 1997.

[2] H. P. Barendregt. Lambda calculi with types. *Handbook of Logic in Computer Science*, wolumen 2, Background: Computational structures. Oxford University Press, 1992.

[3] A. Diller. *Compiling Functional Languages.* John Wiley & Sons LTD, 1988.

[4] A. J. Field, P. G. Harrison. *Functional Programming.* Addison-Wesley Publishing Company, 1988.

[5] J.-Y. Girard, Y. Lafont, P. Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambrige University Press, 1989.

[6] J. R. Hindley. *Basic Simple Type Theory.* Cambridge Tracts in Theoretical Computer Science. Cambrige University Press, 1996.

[7] W. Narkiewicz. *Teoria liczb.* PWN, 1990.

[8] L. Paulson. *ML for the Working Programmer.* Cambrige University Press, 1996.

[9] M. Ryan, M. Sadler. Valuation systems and consequence relations. *Handbook of Logic in Computer Science*, wolumen 1, Background: Mathematical structures. Oxford University Press, 1992.

[10] Z. Spławski. *Proof-Theoretic Approach to Inductive Definitions in ML-like Programming Language versus Second-Order Lambda Calculus.* Praca doktorska, Instytut Matematyki Uniwersytetu Wrocławskiego, 1993.

[11] Z. Spławski. Proving equalities in $\lambda^{\to}$ with positive (co-)inductive data types. Raport instytutowy, Politechnika Wrocławska, 1995.

[12] A. S. Troelstra, D. van Dalen. *Constructivism in Mathematics. An Introduction. Vol I.* Studies in Logic and the Foundations of Mathematics. Elsevier Science Publishers b.v., 1988.

[13] A. S. Troelstra, H. Schwichtenberg. *Basic Proof Theory.* Cambridge Tracts in Theoretical Computer Science. Cambrige University Press, 1996.