

Systemy typów dla języków programowania

Fragmenty skryptu: Prof. Andrew M. Pitts, “Types”
Przetłumaczył: Paweł T. Wojciechowski

29 listopada 2006

Rysunek 1: Informacja kontaktowa

dr inż. Paweł T. Wojciechowski
Instytut Informatyki,
Politechnika Poznańska.

Tel: 61 665 2963

Biuro: Centrum Wykładowe PP, p.5

1 Wstęp

Przykładowe zastosowania systemów typów dla języków programowania przedstawiono na slajdzie.

Językowe bezpieczeństwo Systemy typów są ważnym narzędziem do projektowania bezpiecznych języków, choć w zasadzie, nietypowany język mógłby być także bezpieczny przez wykonywanie odpowiednich kontroli (sprawdzeń) w czasie wykonania programu.

Ponieważ jednak takie kontrole naruszają efektywność, w praktyce jest bardzo niewiele bezpiecznych nietypowanych języków.

Przykładem nietypowanego bezpiecznego języka jest LISP.

Natomiast język assemblera to kwintesencja nietypowanego języka, który nie jest bezpieczny.

Aby zapewnić bezpieczeństwo, typowane bezpieczne języki mogą używać kombinacji kontroli zarówno w czasie kompilacji, jak i kontroli w czasie wykonania programu, jakkolwiek nacisk jest na to pierwsze.

Ideałem byłby system typów implementujący algorytmicznie *rozwiązywalne* (ang. *decidable*) kontrole w czasie kompilacji do wyeliminowania wszystkich niewychwytywalnych błędów (a także pewnych rodzajów wychwytywalnych błędów).

Niektóre języki (np. C) zatrudniają typy, choć nie pretendują przy tym do grona bezpiecznych języków!

Niektóre języki są projektowane jako “bezpieczne” na mocy zastosowania systemu typów, choć może się okazać że wcale bezpieczne nie są z uwagi na nieprzewidywalną kombinację ich cech (języki obiektowe są szczególnie narażone na ten problem). Zobaczmy tego przykład na wykładzie, analizując kombinację polimorfizmu ML ze zmiennymi referencjami (ang. *mutable references*).

Te problemy znacznie przyczyniły się do rozwoju matematyki formalnej i logiki systemów typów: można jedynie wtedy *udowodnić* że język jest bezpieczny, po tym

Rysunek 2: Literatura

1. Wykłady: skrypt "Systemy typów dla języków programowania".
2. Laboratoria: dokumentacja systemów i artykuły wskazane przez prowadzącego.

Inne (niewymagane):

1. Benjamin C. Pierce, *Types and Programming Languages*, MIT Press, 2002.
 2. Benjamin C. Pierce, ed., *Advanced Topics in Types and Programming Languages*, MIT Press, 2005.
 3. Robert Harper, *Programming Languages: Theory and Practice*, 2006. Draft dostępny na stronie WWW autora.
 4. Glynn Winskel, *The Formal Semantics of Programming Languages: An Introduction*, MIT Press, 1996.
-
-

Rysunek 3: Forma zaliczenia przedmiotu

Wykłady:

- obecność na wykładach
- sprawdzian pisemny
- rozmowa zaliczeniowa przy wpisie do indeksu (jeśli potrzeba)

Laboratoria:

- obecność na laboratoriach
 - test i/lub projekt programistyczny
-

Rysunek 4:

Systemy komputerowe zawierające błędy programistyczne nie są godne zaufania.

Zmniejszenie liczby błędów programistycznych zwiększy niezawodność systemów.

Rysunek 5: Jak uniknąć części błędów programistycznych?

1. Testowanie: trudne dla programów współbieżnych i rozproszonych, brak gwarancji,
 2. Mechaniczne dowodzenie poprawności: daje się przeprowadzić dla małych programów w językach dla których istnieją *udowadniacze twierdzeń* (ang. *theorem provers*),
 3. Sprawdzanie modelu (ang. model checking): utwórz model systemu i zastosuj go do symulacji i weryfikacji,
 4. Dobre języki: oferują prawidłowe abstrakcje, które uniemożliwiają pewne błędy (np. garbage collector uniemożliwia błędy przydziału pamięci),
 5. Systemy typów: potrafią udowodnić, że *każde* wykonanie *dowolnego* programu zachowuje dane własności, wyrażone jako typy,
 6. Połączyć różne metody, np. 3. i 5.
-

Rysunek 6: Systemy typów dla języków programowania

“Jedną z najbardziej użytecznych koncepcji w programowaniu jest pojęcie typu, używane do klasyfikowania rodzaju obiektów które są manipulowane [przez programy].

Znacząca część błędów programistycznych jest wykrywana przez implementację, która sprawdza zgodność typów zanim jeszcze program zostanie wykonany.

Typy dostarczają taksonomii która pomaga ludziom myśleć i komunikować się na temat programów komputerowych.”

R. Milner, “Computing Tomorrow” (CUP, 1996)

Rysunek 7: Systemy typów dla języków programowania

Co to są “systemy typów” i do czego są wykorzystywane ?

“System typów jest syntaktyczną metodą udowadniania nieobecności pewnych zachowań programu przez klasyfikowanie fraz zgodnie z rodzajem wartości przez nie obliczanych.”

B. Pierce, “Types and Programming Languages” (MIT, 2002)

Rysunek 8: Systemy typów dla języków programowania

“Praca nad systemami typów dla języków programowania dotyka obecnie wielu dziedzin informatyki, od projektowania i implementacji języków programowania do inżynierii oprogramowania, bezpieczeństwa sieci komputerowych, baz danych, i analizy systemów współbieżnych i rozproszonych.”

B. Pierce ed., “Advanced Topics in Types and Programming Languages” (MIT, 2005)

Rysunek 9: Zastosowania systemów typów

- Detekcja błędów przez *sprawdzanie typów*: statycznie (dające się rozstrzygnąć błędy wykryte zanim programy są wykonane) albo dynamicznie (błędy typowania wykryte podczas wykonania programu).
 - Abstrahowanie i wspomaganie strukturalizacji dużych systemów.
 - Dokumentacja.
 - Efektywność.
 - Bezpieczeństwo języka programowania.
-

Rysunek 10: Językowe bezpieczeństwo (ang. *safety*)

Nieformalna definicje bezpiecznego języka:

“Bezpieczny język to taki, który chroni swoje własne abstrakcje wysokiego poziomu [niezależnie od tego jaki legalny program byśmy w nim napisali].”

“Bezpieczny język jest kompletnie zdefiniowany przez swoją dokumentację programisty [raczej niż kompilator który używamy].”

“Bezpieczny język może pozwalać na *wychwytywalne błędy* (ang. *trapped errors*) [te które mogą być bezproblemowo obsłużone], ale nie może pozwalać na niewychwytywalne błędy [czyli te które powodują nieprzewidywalny krach programu].”

Rysunek 11: Formalne systemy typów

- Zapewniają precyzyjny, matematyczny opis nieformalnych systemów typów (takich jak te które pojawiają się w dokumentacji większości typowanych języków).
 - Stanowią podstawę twierdzeń o *niezawodności systemu typów* (ang. *type soundness*): “każdy dobrze typowany program nie jest w stanie wyprodukować w czasie wykonania żadnych błędów (określonego rodzaju).”
 - Umożliwiają rozdzielenie specyfikacji aspektów typowania danego języka od spraw algorytmicznych: formalny system typów może zdefiniować typowanie niezależnie od poszczególnych implementacji algorytmów sprawdzania typów.
-

Rysunek 12: Zaawansowane zastosowania

Przykładowo, zaprojektowano systemy typów do:

1. bezpiecznych operacji na wskaźnikach i zarządzania pamięcią,
 2. bezpiecznego wykonywania nieznanych programów (“sandboxes”),
 3. unikania warunków wyścigu, np. bezpieczne zamki (ang. safe locking),
 4. uzyskiwania atomowości w programach wielowątkowych,
 5. unikania zakleszczenia,
 6. kompozycji protokołów, np. typy sesyjne (ang. session types),
 7. generacja kodu z dowodem (ang. proof-carrying code).
-

Rysunek 13: Zaawansowane zastosowania

Dość niezwykle zastosowania systemów typów:

1. bezpieczne przetaczanie komunikatów (and. type-safe marshalling),
2. kontrola wersji oprogramowania,
3. bezpieczna dynamiczna aktualizacja kodu (ang. type-safe dynamic software updating),
4. weryfikacja reguł deklaratywnej synchronizacji.

Nowe zastosowania pojawiają się na bieżąco!

Rysunek 14: Przykładowe narzędzia i języki

1. Objective Caml: popularny wariant języka ML
 2. Java 1.5: "generics" (typy polimorficzne)
 3. Java Programming Toolkit: Escjava - Extended Static Checker
 4. Java Pathfinder: weryfikacja bytecode-u Java (NASA)
 5. CCured: bezpieczne zarządzanie pamięcią w języku C
 6. Cyclone: bezpieczny dialekt języka C
 7. XDuce: typowany język do przetwarzania XML
 8. Scala: przetwarzanie XML, kompilowany do bytekodu Java
 9. Pict i Nomadic Pict: statyczne typowanie komunikacji
 10. Acute: bezpieczna komunikacja między programami
-

Rysunek 15: Typowe “stwierdzenie” w systemie typów

(ang. *typing judgement*)

jest relacją między środowiskiem typowania (Γ), frazami programu (M) i wyrażeniami typu (t), którą zapisujemy

$$\Gamma \vdash M : t$$

i czytamy “mając dane przypisanie typów do wolnych identyfikatorów frazy M wyspecyfikowane przez środowisko Γ , fraza M ma typ t ”.

Na przykład

$$f : int\ list \rightarrow int, b : bool \vdash (if\ b\ then\ f\ nil\ else\ 3) : int$$

jest prawidłowym stwierdzeniem typowania w języku ML

jak jego syntaktyka i semantyka operacyjna (która opisuje jak programy w tym języku działają) zostanie formalnie wyspecyfikowana. Istotą tego wykładu jest wstępne zapoznanie się z taką formalizacją, oraz zilustrowanie jej użycia.

Standard ML (Milner, Tofte, Harper i MacQueen 1997) był pierwszym przykładem pełnowymiarowego języka programowania, posiadającego kompletną formalną specyfikację, i dla którego *niezawodność systemu typów* (ang. *type soundness*) była przedmiotem wielu formalnych dowodów.

Studiowanie formalnych systemów typów jest częścią *strukturalnej semantyki operacyjnej*: aby wyspecyfikować formalny system typów, należy podać szereg aksjomatów i reguł do indukcyjnego generowania takiego rodzaju asercji, lub “twierdzenia”, jak to na slajdzie 15. Idealnie, reguły te powinny podążać za strukturą frazy M , wyjaśniając jak ją typować (tj. “znajdować jej typ”) poprzez pokazanie jak podfrazy frazy M mogą być typowane. Tak więc możemy tu mówić o zestawie reguł *ukierunkowanych syntaktycznie*.

Kiedy sformalizowaliśmy dany system typów, jesteśmy w stanie *udowodnić* rezultaty dotyczące *niezawodności systemu typów* (ang. *type soundness*), *sprawdzania typów* (ang. *type checking*), *typowalności* (ang. *typeability*), oraz *dedukcji typów* (ang. *type inference*); trzy ostatnie problemy opisane zostały na slajdzie 17.

Rysunek 16: Różne notacje relacji typowania

“x ma typ integer”

styl ML (używany na wykładach):

`x : integer`

styl Haskell:

`x :: integer`

styl C/Java:

`integer x`

Rysunek 17: Sprawdzanie, typowalność, oraz dedukcja typu

Weźmy system typów dla języka programowania ze stwierdzeniami w formie $\Gamma \vdash M : t$.

Problem *sprawdzania typu*: mając dane Γ , M i t , czy stwierdzenie $\Gamma \vdash M : t$ dało by się wywieść (wyprowadzić) w tym systemie typów ?

Problem *typowalności*: mając dane Γ i M , czy jest jakikolwiek typ t dla którego stwierdzenie $\Gamma \vdash M : t$ dało by się wywieść w tym systemie typów ?

Drugi problem jest zwykle trudniejszy niż pierwszy. Jego rozwiązanie zwykle ciąga za sobą wymyślenie *algorytmu dedukcji* typu, który oblicza typ t dla każdego Γ i M (lub kończy się niepowodzeniem, jeśli takiego typu nie ma).

Rysunek 18: Polimorfizm = “ma wiele typów”

Przeciążenie (ang. *overloading*) (lub 'ad-hoc' polimorfizm): taki sam symbol oznacza operacje z nie mającymi nic wspólnego implementacjami. (Np. + może oznaczać zarówno dodanie liczb naturalnych, jak i konkatenację łańcuchów tekstowych.)

Włączenie (ang. *subsumption*) $t_1 <: t_2$: każda fraza $M_1 : t_1$ może być używana jako $M_1 : t_2$ bez narażania bezpieczeństwa.

Parametryczny polimorfizm (ang. *parametric polymorphism*) (“generics”): takie samo wyrażenie przynależy do rodziny strukturalnie powiązanych typów.

2 Polimorfizm w ML

Wczesne formy statycznego typowania, takie jak w języku Pascal, uniemożliwiały pisanie *generycznego kodu*. Na przykład, procedura do sortowania list danego typu nie mogła być wykorzystana do list innego typu danych.

Byłoby naturalne mieć *polimorficzną* procedurę sortowania – tj. taką która działa (jednociele) na listach szeregu różnych typów.

Na znaczenie *polimorfizmu* (ang. *polymorphism*) dla języków programowania zwrócił po raz pierwszy uwagę Strachey (1967), identyfikując szereg różnych jego znaczeń.

Możemy wywołać funkcje *ile*, np. jako `ile [1; 2; 3]` lub `ile ["Ala"; "kot"]`.

Na tym kursie skoncentrujemy się na parametrycznym polimorfizmie. Jednym ze sposobów jego uzyskania jest uczynienie parametryzacji typu *eksplikatywną* częścią syntaktyki języka. My jednak spojrzymy na *implikatywną* wersję parametrycznego polimorfizmu, po raz pierwszy zaimplementowaną w rodzinie języków ML (i następnie adoptowaną gdzie indziej, na przykład pod nazwą “generics” w najnowszych wersjach języków Java i C#).

Frazy w języku ML zawierać mogą bardzo niewiele informacji o typach: algorytm dedukcji typów znajduje “najbardziej ogólny” typ (schemat) dla każdej dobrze uformowanej frazy, z którego wszystkie inne typy frazy mogą być otrzymane przez specjalizowanie zmiennych typu.

W dalszej części kursu przedstawiona zostanie precyzyjna formalizacja systemu typów oraz związanego z nim algorytmu dedukcji typu dla małego fragmentu języka ML, który nazwijmy Mini-ML.

Rysunek 19: Przykład parametrycznego polimorfizmu

W Objective Caml (OCaml), popularnej implementacji języka ML, funkcja `ile` obliczająca ilość elementów listy

```
let rec ile x =
  match x with
  [] -> 0
  | h::t -> 1 + ile t
```

ma typ $t \text{ list} \rightarrow int$ dla wszystkich typów t .

Rysunek 20: Zmienne typu i schematy typów dla Mini-ML

Aby sformalizować wyrażenia takie jak

“funkcja `ile` ma typ $t \text{ list} \rightarrow int$, dla wszystkich typów t ”

jest naturalne, aby wprowadzić *zmienne typu* (ang. *type variables*) α (tj. zmienne które mogą być zastępowane przez typy) i pisać

$$ile : \forall \alpha (\alpha \text{ list} \rightarrow int) .$$

$\forall \alpha (\alpha \text{ list} \rightarrow int)$ jest przykładem *schematu typów* (ang. *type scheme*).

Rysunek 21: Polimorfizm let-związanych zmiennych w ML

Na przykład w wyrażeniu

$$\text{let } f = \lambda x(x) \text{ in } (f \text{ true}) :: (f \text{ nil})$$

funkcja $\lambda x(x)$ ma typ $t \rightarrow t$ dla dowolnego typu t , a zmienna f do której ta funkcja jest przywiązana jest używana polimorficznie:

- w $(f \text{ true})$, f ma typ $bool \rightarrow bool$

- w $(f \text{ nil})$, f ma typ $bool \text{ list} \rightarrow bool \text{ list}$.

Całościowo, wyrażenie to ma typ $bool \text{ list}$.

2.1 System typów języka ML

Aby sformalizować system typów musieliśmy wprowadzić *zmiennne typu*. Interaktywny ML pokaże typ $\alpha \text{ list} \rightarrow int$ jako typ funkcji *ile*. Jednakże w formalnym opisie systemu typów, konieczne jest aby kwantyfikacja po typach była w jakiś sposób widoczna. Przyczyna tego tkwi w typowaniu lokalnych deklaracji - zob. przykład na slajdzie 21.

Wyrażenie $(f \text{ true}) :: (f \text{ nil})$ ma typ $bool \text{ list}$, mając dane pewne założenie o typie zmiennej f . Możliwe dwa takie założenia pokazano na slajdzie 22. My najbardziej zainteresowani jesteśmy drugą możliwością, gdyż prowadzi ona do systemu typów który ma bardzo użyteczne własności.

Poniżej przedstawimy gramatykę typów i schematów typu, którą będziemy używać w kolejnych przykładach.

Należy zwrócić uwagę na następujące sprawy w odniesieniu do schematów typu $\forall A(t)$:

1. Przypadek kiedy A jest pusty, $A = \{\}$, jest dopuszczalny: $\forall\{\}(t)$ jest dobrze uformowanym schematem typu. **Często będziemy patrzeć na zbiory typów jako na podzbiór zbioru schematów typu, identyfikując typ t ze schematem typu $\forall\{\}(t)$.**
2. Każde pojawienie się w typie t , zmiennej typu $\alpha \in A$, staje się związane w schemacie $\forall A(t)$. Tak więc z definicji, *wolne zmienne typu* (ang. *free type variables*) schematu typów $\forall A(t)$ są to wszystkie te zmienne typu, które pojawiają się w

Rysunek 22: Ad-hoc vs. parametryczny polimorfizm

“Ad-hoc” polimorfizm:

jeśli $f : bool \rightarrow bool$
 oraz $f : bool\ list \rightarrow bool\ list$,
 to $(f\ true) :: (f\ nil) : bool\ list$

“Parametric” polimorfizm:

Jeśli $f : \forall\alpha(\alpha \rightarrow \alpha)$,
 to $(f\ true) :: (f\ nil) : bool\ list$

Rysunek 23: Mini-ML typy i schematy typów

Types

$t ::= \alpha$	zmienna typu
$bool$	typ wartości logicznych
$t \rightarrow t$	typ funkcji
$t\ list$	typ list

gdzie α przyjmuje wartości z ustalonego, przeliczalnego, nieskończonego zbioru TyVar.

Schematy typów

$$\sigma ::= \forall A(t)$$

gdzie A zmienia się po skończonych podzb. zbioru TyVar.

Kiedy $A = \alpha_1, \dots, \alpha_n$, piszemy $\forall A(t)$ jako $\forall\alpha_1, \dots, \alpha_n(t)$.

t , ale które nie są w skończonym zbiorze A . (Na przykład zbiorem wolnych zmiennych typu $\forall\alpha(\alpha \rightarrow \alpha')$ jest α' .)

Tak jak zwykle w przypadku konstrukcji wiążących zmienne, nie interesują nas poszczególne nazwy związanych zmiennych typu (gdyż tak czy inaczej możemy musieć je zmienić, aby uniknąć uchwycenia podczas podstawiania typów pod zmienne typów, innych zmiennych o tych samych nazwach, które mogą wystąpić w podstawianych typach).

Dlatego też **będziemy identyfikować schematy typów uwzględniając możliwą alfa-konwersję zmiennych typu wiązanych przez \forall** . Na przykład, $\forall\alpha(\alpha \rightarrow \alpha')$ i $\forall\alpha''(\alpha'' \rightarrow \alpha')$ wpadają do tej samej alfa-równoznacznej klasy i mogą być stosowane zamiennie. Oczywiście skończony zbiór

$$wzt(\forall A(t))$$

wolnych zmiennych typu w schemacie typu jest dobrze zdefiniowany uwzględniając możliwą alfa-konwersję związanych zmiennych typu (tj. spójne, równoczesne przemianowanie nazw tych zmiennych). Będziemy także pisać $wzt(t)$ mając na myśli skończony zbiór (wolnych) zmiennych typu występujących w t .

3. Schematy typów w ML nie są typami w ML. Tak więc, na przykład $\alpha \rightarrow \forall\alpha'(\alpha')$ nie jest ani dobrze uformowanym typem w Mini-ML, ani dobrze uformowanym schematem w Mini-ML. Możemy otrzymać typy ze schematów typu przez zastąpienie zmiennych typu przez typy.

Proste przykłady generalizacji:

$$\begin{aligned} \forall\alpha(\alpha \rightarrow \alpha) &\succ bool \rightarrow bool \\ \forall\alpha(\alpha \rightarrow \alpha) &\succ \alpha' list \rightarrow \alpha' list \\ \forall\alpha(\alpha \rightarrow \alpha) &\succ (\alpha' \rightarrow \alpha') \rightarrow (\alpha' \rightarrow \alpha') . \end{aligned}$$

Jednakże

$$\forall\alpha(\alpha \rightarrow \alpha) \not\succeq (\alpha' \rightarrow \alpha') \rightarrow \alpha' .$$

A to dlatego, że w *zastąpieniu* (ang. *substitution*) $t[t'/\alpha]$, z definicji musimy zastąpić *wszystkie* wystąpienia w t zmiennej typu α przez t' . Tak więc, kiedy $t = \alpha \rightarrow \alpha$, to nie ma takiego typu t' dla którego $t[t'/\alpha]$ byłoby typem $(\alpha \rightarrow \alpha) \rightarrow \alpha$. (Po prostu dlatego, że w drzewie syntaktycznym $t[t'/\alpha] = t' \rightarrow t'$, dwa poddrzewa poniżej najbardziej skrajnego konstruktora \rightarrow są równe (konkretnie równe t'), podczas gdy jest to nieprawdą w $(\alpha \rightarrow \alpha) \rightarrow \alpha$.)

Inny przykład

$$\forall\alpha_1, \alpha_2(\alpha_1 \rightarrow \alpha_2) \succ \alpha list \rightarrow bool .$$

Rysunek 24: Relacja “generalizuj”

między schematami typów a typami

Mówimy że schemat typu $\sigma = \forall \alpha_1, \dots, \alpha_n (t')$ *generalizuje* typ t , co zapisujemy $\boxed{\sigma \succ t}$ jeżeli t może być otrzymane z typu t' przez równoczesne podstawienie pewnych typów t_i pod zmienne typów α_i ($i = 1, \dots, n$):

$$t = t'[t_1/\alpha_1, \dots, t_n/\alpha_n] .$$

(Uwaga: Specyficzny wybór nazw zmiennych typu w σ nie wpływa na tę relację.)

Odwrotna relacja jest zwana specjalizacją: typ t jest *specjalizacją* schematu typu σ , jeśli $\sigma \succ t$.

Jakkolwiek

$$\forall \alpha_1 (\alpha_1 \rightarrow \alpha_2) \neq \alpha \text{ list} \rightarrow \text{bool}$$

ponieważ α_2 jest wolną zmienną typu w schemacie typów $\forall \alpha_1 (\alpha_1 \rightarrow \alpha_2)$, a więc nie może być zastępowane podczas specjalizacji.

Slajd 25 prezentuje formę stwierdzeń typowania, którą będziemy używali aby zilustrować polimorfizm w ML oraz wnioskowanie typu. Ponieważ będziemy brali pod uwagę jedynie mały podzbiór typów języka ML, ograniczymy naszą uwagę jedynie do typowania dla małego podzbioru M wyrażeń języka ML, generowanego przez gramatykę na slajdzie 26. Aksjomaty i reguły generujące indukcyjnie relację typowania w Mini-ML dla tych wyrażeń podano na slajdach 27–30.

Zwróćmy uwagę na następujące rzeczy w systemie typów na slajdach 27–30.

1. Mając dane środowisko Γ piszemy $\Gamma, x : \sigma$ mając na myśli środowisko typowania z dziedziną $\text{dom}(\Gamma) \cup \{x\}$, które odwzorowuje x do σ a w pozostałych przypadkach tak jak Γ . Kiedy używamy tej notacji, to niemal w każdym przypadku będzie $x \notin \text{dom}(\Gamma)$ (zob. reguły (fn), (let) i (match)).
2. W regule (fn) używamy $\Gamma, x : t_1$ jako skrót dla $\Gamma, x : \forall \{ \} (t_1)$. Podobnie, w regule (match), $\Gamma, x_1 : t_1, x_2 : t_1 \text{ list}$ tak na prawdę oznacza $\Gamma, x_1 : \forall \{ \} (t_1), x_2 : \forall \{ \} (t_1 \text{ list})$. (Pamiętajmy, że z definicji, środowisko typowania ma odwzorowywać zmienne do schematów typu, raczej niż do typów.)

Rysunek 25: Stwierdzenie typowania w Mini-ML

ma formę $\boxed{\Gamma \vdash M : t}$ gdzie

- *Środowisko typowania* Γ jest skończoną funkcją ze zmiennych do *schematów typu*.
(Zapisujemy $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ aby wskazać że Γ ma dziedzinę zdefiniowaną jako $dom(\Gamma) = \{x_1, \dots, x_n\}$ i odwzorowuje każde x_i do schematu typów σ_i dla $i = 1..n$.)
 - M jest wyrażeniem w języku Mini-ML.
 - t jest typem w Mini-ML.
-

Rysunek 26: Wyrażenia w Mini-ML

$M ::= x$	zmienna
true	wartości logiczne
false	
if M then M else M	warunek
$\lambda x(M)$	abstrakcja funkcji
$M M$	aplikacja funkcji
let $x = M$ in M	lokalna deklaracja
nil	pusta lista
$M :: M$	konstruktor listy (list cons)
match M with	
nil $\Rightarrow M$	
$x :: x \Rightarrow M$	wybór po wzorcu

Rysunek 27: System typów Mini-ML (1/4)

(var \succ) $\Gamma \vdash x : t$ jeśli $(x : \sigma) \in \Gamma$ oraz $\sigma \succ t$

(bool) $\Gamma \vdash B : bool$ jeśli $B \in \{\text{true}, \text{false}\}$

(if)
$$\frac{\Gamma \vdash M_1 : bool \quad \Gamma \vdash M_2 : t \quad \Gamma \vdash M_3 : t}{\Gamma \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : t}$$

3. W regule (let) notacja $wzt(\Gamma)$ oznacza zbiór wszystkich zmiennych typu pojawiających się jako wolne (nie związane) w pewnych schematach typów przypisanych w Γ . (Na przykład, jeśli $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$, to wtedy $wzt(\Gamma) = wzt(\sigma_1) \cup \dots \cup wzt(\sigma_n)$.) Tak więc zbiór $A = wzt(t_1) - wzt(\Gamma)$ użyty w tej regule składa się ze wszystkich zmiennych typu w t_1 , które nie występują wolne w żadnym schemacie typów przypisanym w Γ .

Tak jak zwykle, aksjomaty i reguły na slajdach 27–30 są schematyczne: Γ , M , t , i σ oznaczają *dowolne* dobrze uformowane środowisko, wyrażenie, typ, i schemat typu. Aksjomaty i reguły są używane do wygenerowania indukcyjnie *relacji typowania*—podzbioru wszystkich możliwych trójek $\Gamma \vdash M : \sigma$. Mówimy że poszczególna trójka $\Gamma \vdash M : \sigma$ jest *wywnioskowalna* (lub *udowodnialna*, lub *ważna*) w systemie typów, jeśli istnieje jej dowód używając aksjomatów i reguł systemu typów. Tak więc relacja typowania składa się z dokładnie tych trójek dla których istnieje taki dowód. Kiedy $\Gamma = \{\}$, piszemy jedynie

$$\vdash M : \sigma$$

jeśli daje się wywnioskować że $\{\} \vdash M : \sigma$, i mówimy wtedy, że wyrażenie M (koniecznie domknięte) jest *typowalne* w Mini-ML ze schematem typu σ .

Przykład Zweryfikujmy, że podany wcześniej na slajdzie 21 przykład polimorfizmu zmiennych let-związanych ma rzeczywiście podany tam typ, tj. że:

$$\vdash \text{let } f = \lambda x(x) \text{ in } (f \text{ true}) :: (f \text{ nil}) : bool \text{ list} .$$

Dowód. Po pierwsze zauważmy że $\vdash \lambda x(x) : \forall \alpha(\alpha \rightarrow \alpha)$, jak pokazuje poniższy dowód:

$$\frac{\frac{}{x : \alpha \vdash x : \alpha} \text{ (var } \succ)}{\{\} \vdash \lambda x(x) : \alpha \rightarrow \alpha} \text{ (fn)}}{\text{używając } \forall \{\}(\alpha) \succ \alpha} \quad (1)$$

Rysunek 28: System typów Mini-ML (2/4)

(nil) $\Gamma \vdash \text{nil} : t \text{ list}$

(cons)
$$\frac{\Gamma \vdash M_1 : t \quad \Gamma \vdash M_2 : t \text{ list}}{\Gamma \vdash M_1 :: M_2 : t \text{ list}}$$

(match)
$$\frac{\Gamma \vdash M_1 : t_1 \text{ list} \quad \Gamma \vdash M_2 : t_2 \quad \Gamma, x_1 : t_1, x_2 : t_1 \text{ list} \vdash M_3 : t_2}{\Gamma \vdash \text{match } M_1 \text{ with nil} \Rightarrow M_2 \mid x_1 :: x_2 \Rightarrow M_3 : t_2}$$
 jeśli $x_1, x_2 \notin \text{dom}(\Gamma)$
oraz $x_1 \neq x_2$

Rysunek 29: System typów Mini-ML (3/4)

(fn)
$$\frac{\Gamma, x : t_1 \vdash M : t_2}{\Gamma \vdash \lambda x(M) : t_1 \rightarrow t_2} \quad \text{jeśli } x \notin \text{dom}(\Gamma)$$

(ap)
$$\frac{\Gamma \vdash M_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash M_2 : t_1}{\Gamma \vdash M_1 M_2 : t_2}$$

Rysunek 30: System typów Mini-ML (4/4)

$$\text{(let)} \quad \frac{\Gamma \vdash M_1 : \sigma \quad \Gamma, x : \sigma \vdash M_2 : t}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : t} \quad \text{jeśli } x \notin \text{dom}(\Gamma)$$

gdzie dla schematu typów $\sigma = \forall A(t_1)$, piszemy

$$\Gamma \vdash M_1 : \sigma$$

w znaczeniu $\Gamma \vdash M_1 : t_1$ oraz $A = \text{wzt}(t_1) - \text{wzt}(\Gamma)$.

Rysunek 31: System typów Mini-ML

$$\begin{array}{ll} \text{(var } \succ) \quad \Gamma \vdash x : t \text{ jeśli } (x : \sigma) \in \Gamma \text{ oraz } \sigma \succ t & \text{(fn)} \quad \frac{\Gamma, x : t_1 \vdash M : t_2}{\Gamma \vdash \lambda x(M) : t_1 \rightarrow t_2} \\ & \text{jeśli } x \notin \text{dom}(\Gamma) \\ \text{(bool)} \quad \Gamma \vdash B : \text{bool} \text{ jeśli } B \in \{\text{true}, \text{false}\} & \\ \text{(if)} \quad \frac{\Gamma \vdash M_1 : \text{bool} \quad \Gamma \vdash M_2 : t \quad \Gamma \vdash M_3 : t}{\Gamma \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : t} & \text{(ap)} \quad \frac{\Gamma \vdash M_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash M_2 : t_1}{\Gamma \vdash M_1 M_2 : t_2} \\ \text{(nil)} \quad \Gamma \vdash \text{nil} : t \text{ list} & \\ \text{(cons)} \quad \frac{\Gamma \vdash M_1 : t \quad \Gamma \vdash M_2 : t \text{ list}}{\Gamma \vdash M_1 :: M_2 : t \text{ list}} & \text{(let)} \quad \frac{\Gamma \vdash M_1 : \sigma \quad \Gamma, x : \sigma \vdash M_2 : t}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : t} \\ & \text{jeśli } x \notin \text{dom}(\Gamma) \\ \text{(match)} \quad \frac{\Gamma \vdash M_1 : t_1 \text{ list} \quad \Gamma \vdash M_2 : t_2 \quad \Gamma, x_1 : t_1, x_2 : t_1 \text{ list} \vdash M_3 : t_2}{\Gamma \vdash \text{match } M_1 \text{ with nil} \Rightarrow M_2} & \text{gdzie dla schematu typów} \\ & \sigma = \forall A(t_1), \text{ piszemy} \\ & \Gamma \vdash M_1 : \sigma \\ & \text{w znaczeniu } \Gamma \vdash M_1 : t_1 \text{ oraz} \\ & A = \text{wzt}(t_1) - \text{wzt}(\Gamma). \end{array}$$

Następnie zauważmy, że ponieważ $\forall\alpha(\alpha \rightarrow \alpha) \succ \text{bool} \rightarrow \text{bool}$, to przez (var \succ) mamy:

$$f : \forall\alpha(\alpha \rightarrow \alpha) \vdash f : \text{bool} \rightarrow \text{bool} .$$

Przez (bool), mamy także

$$f : \forall\alpha(\alpha \rightarrow \alpha) \vdash \text{true} : \text{bool}$$

i aplikując regułę (ap) do tych dwóch stwierdzeń dostajemy

$$f : \forall\alpha(\alpha \rightarrow \alpha) \vdash f \text{ true} : \text{bool} . \quad (2)$$

Podobnie, używając (ap) i (var \succ) oraz (nil), mamy

$$f : \forall\alpha(\alpha \rightarrow \alpha) \vdash f \text{ nil} : \text{bool list} . \quad (3)$$

Aplikując regułę (cons) do (2) i (3) mamy

$$f : \forall\alpha(\alpha \rightarrow \alpha) \vdash (f \text{ true}) :: (f \text{ nil}) : \text{bool list} .$$

Na koniec możemy aplikować regułę (let) do tego oraz (1) i konkludować, że

$$\{\} \vdash \text{let } f = \lambda x(x) \text{ in } (f \text{ true}) :: (f \text{ nil}) : \text{bool list}$$

co należało udowodnić. □

2.2 Przykłady dedukcji typów, ręcznie

Zarówno dla pełnego systemu typów języka ML, jak i również dla systemu typów który właśnie przedstawiliśmy, problem typowalności (slajd 17) okazuje się być decydowalny. Co więcej, spośród wszystkich możliwych schematów typu które może posiadać dane domknięte wyrażenie Mini-ML, istnieje najbardziej ogólny schemat typu—to jest taki, z którego wszystkie inne mogą być otrzymane przez zastąpienie. Zanim pokażemy dlaczego tak jest, podamy kilka specyficznych przykładów dedukcji typu w naszym systemie typów.

Mając dane środowisko typowania Γ i wyrażenie M , jak możemy zdecydować czy istnieje lub nie schemat typów σ dla którego zachodzi $\Gamma \vdash M : \sigma$? Pomocna okaże się w tym zadaniu *nakierunkowana syntaktycznie* (lub inaczej mówiąc “strukturalna”) natura aksjomatów i reguł na slajdach 27–30: jeżeli da się wywieść że $\Gamma \vdash M : \forall A(t)$, to jest jeśli da się wywieść że $A = \text{wzt}(t) - \text{wzt}(\Gamma)$ oraz $\Gamma \vdash M : t$, to wtedy najbardziej zewnętrzna forma wyrażenia M dyktuje jaki musi być ostatni aksjom lub reguła użyta w dowodzie stwierdzenia $\Gamma \vdash M : \sigma$ (zob. ostatni przykład). W konsekwencji, kiedy próbujemy zbudować dowód stwierdzenia typowania $\Gamma \vdash M : \sigma$ od dołu do góry, struktura wyrażenia M określa kształt drzewa, oraz to które reguły mają być używane w jego węzłach, oraz które aksjomaty w jego liściach.

Rysunek 34: Warunki generowane podczas dedukcji typu dla

$$\text{let } f = \lambda x_1(\lambda x_2(x_1)) \text{ in } f f$$

(C0)		$A = wzt(t_2)$
(C1)		$t_2 = t_3 \rightarrow t_4$
(C2)		$t_4 = t_5 \rightarrow t_6$
(C3)	$\forall\{\}(t_3) \succ t_6$, tj.	$t_3 = t_6$
(C4)		$t_7 = t_8 \rightarrow t_1$
(C5)		$\forall A(t_2) \succ t_7$
(C6)		$\forall A(t_2) \succ t_8$

Na przykład dla wyrażenia M na slajdzie 32, jakkolwiek dowód $\vdash M : \forall A_1(t_1)$ z aksjomatów i reguł, musi wyglądać jak drzewo na slajdzie 33.

Węzeł (C0) powinien być wystąpieniem reguły (let); węzły (C1) i (C2) wystąpieniami reguły (fn); liście (C3) (C5) (C6) wystąpieniami aksjomatu (var \succ); a węzeł (C4) wystąpieniem reguły (ap). Aby rzeczywiście były to prawidłowe wystąpienia, muszą być spełnione warunki (C0)-(C6), wyszczególnione na slajdzie 34.

Tak więc M jest typowalne wtedy i tylko wtedy gdy możemy znaleźć typy t_1, \dots, t_8 spełniające warunki na slajdzie 34. Po pierwsze zauważmy, że implikują one

$$t_2 \stackrel{(C1)}{=} t_3 \rightarrow t_4 \stackrel{(C2)}{=} t_3 \rightarrow (t_5 \rightarrow t_6) \stackrel{(C3)}{=} t_6 \rightarrow (t_5 \rightarrow t_6) .$$

W takim razie niech t_5, t_6 będą zmiennymi typu, powiedzmy odpowiednio α_2, α_1 . A więc przez (C0), $A = wzt(t_2) = wzt(\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)) = \{\alpha_1, \alpha_2\}$. Wtedy (C4), (C5) i (C6) wymagają że

$$\forall \alpha_1, \alpha_2 (\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)) \succ t_8 \rightarrow t_1 \quad \text{oraz} \quad \forall \alpha_1, \alpha_2 (\alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1)) \succ t_8 .$$

Innymi słowy muszą istnieć jakieś typy t_9, \dots, t_{12} , takie że

$$\begin{aligned} (C7) \quad & t_9 \rightarrow (t_{10} \rightarrow t_9) = t_8 \rightarrow t_1 \\ (C8) \quad & t_{11} \rightarrow (t_{12} \rightarrow t_{11}) = t_8 . \end{aligned}$$

Teraz (C7) może zachodzić tylko wtedy gdy

$$t_9 = t_8 \quad \text{oraz} \quad t_{10} \rightarrow t_9 = t_1$$

a więc

$$t_1 = t_{10} \rightarrow t_9 = t_{10} \rightarrow t_8 \stackrel{(C8)}{=} t_{10} \rightarrow (t_{11} \rightarrow (t_{12} \rightarrow t_{11}))$$

z t_{10} , t_{11} , t_{12} nie związanymi w przeciwnym razie. W takim razie jeśli byłyby one zmiennymi typu, odpowiednio α_3 , α_4 , α_5 , to koniec końców, możemy spełnić wszystkie warunki ze slajdu 34 definiując

$$\begin{aligned} A &= \{\alpha_1, \alpha_2\} \\ t_1 &= \alpha_3 \rightarrow (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4)) \\ t_2 &= \alpha_1 \rightarrow (\alpha_2 \rightarrow \alpha_1) \\ t_3 &= \alpha_1 \\ t_4 &= \alpha_2 \rightarrow \alpha_1 \\ t_5 &= \alpha_2 \\ t_6 &= \alpha_1 \\ t_7 &= (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4)) \rightarrow (\alpha_3 \rightarrow (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4))) \\ t_8 &= \alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4) . \end{aligned}$$

z tymi wyborami, slajd 33 staje się prawidłowym dowodem stwierdzenia

$$\{\} \vdash \mathbf{let} \ f = \lambda x_1(\lambda x_2(x_1)) \ \mathbf{in} \ f \ f : \alpha_3 \rightarrow (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4))$$

na podstawie aksjomatów typowania i reguł ze slajdów 27–30, tj. zachodzi

$$\vdash \mathbf{let} \ f = \lambda x_1(\lambda x_2(x_1)) \ \mathbf{in} \ f \ f : \forall \alpha_3, \alpha_4, \alpha_5 (\alpha_3 \rightarrow (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4))) \quad (4)$$

Jeśli przejdziemy przez ten sam proces dedukcji typu dla wyrażenia M' na slajdzie 32, to wygenerujemy drzewo i zestaw warunków jak na slajdzie 35. To implikuje w szczególności że

$$t_7 \stackrel{(C13)}{=} t_4 \stackrel{(C12)}{=} t_6 \stackrel{(C11)}{=} t_7 \rightarrow t_5 .$$

Ale nie ma typów t_5 , t_7 spełniających $t_7 = t_7 \rightarrow t_5$, ponieważ $t_7 \rightarrow t_5$ zawiera przynajmniej jeden symbol ' \rightarrow ' więcej niż t_7 . Tak więc możemy zakonkludować że $(\lambda f (f f)) \lambda x_1(\lambda x_2(x_1))$ nie jest typowalne w ramach systemu typów ML.

2.3 Pryncypalne schematy typów

Schemat typu $\forall \alpha_3, \alpha_4, \alpha_5 (\alpha_3 \rightarrow (\alpha_4 \rightarrow (\alpha_5 \rightarrow \alpha_4)))$ nie tylko spełnia (4), jest to w istocie najbardziej ogólny lub *pryncypalny* (ang. *principal*) schemat typu dla wyrażenia $\mathbf{let} \ f = \lambda x_1(\lambda x_2(x_1)) \ \mathbf{in} \ f \ f$, jak zdefiniowano na slajdzie 36. Definicja używa przy tym naturalnego rozszerzenia relacji “generalizacji”, \succ (Slajd 24), do relacji binarnej

Rysunek 35: Szkielet drzewa dowodu i warunków dla

$$(\lambda f(f f)) \lambda x_1(\lambda x_2(x_1))$$

$$\frac{\frac{\frac{}{f : t_4 \vdash f : t_6} (C12) \quad \frac{}{f : t_4 \vdash f : t_7} (C13)}{\frac{}{f : t_4 \vdash f f : t_5} (C10)}{\frac{}{\{ \} \vdash \lambda f(f f) : t_2} (C10)}{\frac{}{\{ \} \vdash (\lambda f(f f)) \lambda x_1(\lambda x_2(x_1)) : t_1} (C9)}{\frac{\frac{\frac{}{x_1 : t_8, x_2 : t_{10} \vdash x_1 : t_{11}} (C16)}{\frac{}{x_1 : t_8 \vdash \lambda x_2(x_1) : t_9} (C14)}{\frac{}{\{ \} \vdash \lambda x_1(\lambda x_2(x_1)) : t_3} (C9)}{\frac{}{\{ \} \vdash (\lambda f(f f)) \lambda x_1(\lambda x_2(x_1)) : t_1} (C9)}{\frac{}{\{ \} \vdash (\lambda f(f f)) \lambda x_1(\lambda x_2(x_1)) : t_1} (C9)}$$

Warunki : (C9)	$t_2 = t_3 \rightarrow t_1$
(C10)	$t_2 = t_4 \rightarrow t_5$
(C11)	$t_6 = t_7 \rightarrow t_5$
(C12)	$\forall \{ \} (t_4) \succ t_6, \mathbf{tj.} \ t_4 = t_6$
(C13)	$\forall \{ \} (t_4) \succ t_7, \mathbf{tj.} \ t_4 = t_7$
(C14)	$t_3 = t_8 \rightarrow t_9$
(C15)	$t_9 = t_{10} \rightarrow t_{11}$
(C16)	$\forall \{ \} (t_{11}) \succ t_8, \mathbf{tj.} \ t_{11} = t_8$

Rysunek 36: Pryncypalny σ dla wyrażeń domkniętych

System typów σ jest *pryncypalnym* (ang. *principal*) schematem typów domkniętego wyrażenia M w języku Mini-ML, jeśli

(a) $\vdash M : \sigma$ daje się udowodnić

(b) dla wszystkich σ' , jeśli $\vdash M : \sigma'$ daje się udowodnić, to wtedy $\sigma \succ \sigma'$

gdzie z definicji $\boxed{\sigma \succ \sigma'}$ zachodzi jeśli $\sigma' = \forall A'(t')$ razem z $A' \cap \text{wzt}(\sigma) = \{\}$ oraz $\sigma \succ t'$ (zob. definicję na slajdzie 24).

(Zauważmy, że ponieważ identyfikujemy schematy typów aż do alfa-konwersji \forall -związanych zmiennych typu, możemy zawsze spełnić warunek $A' \cap \text{wzt}(\sigma) = \{\}$ przez odpowiednie przemianowanie związanych zmiennych typu w schemacie σ' .)

między (domkniętymi) schematami typów. Warto zauważyć, że w obecności (a), zachodzi odwrotność warunku (b) na slajdzie 36: jeśli $\sigma \succ \sigma'$ oraz $\vdash M : \sigma$, to wtedy $\vdash M : \sigma'$.

Slajd 37 przedstawia główny rezultat na temat problemu typowalności wyrażeń Mini-ML. Został on po raz pierwszy udowodniony dla prostego systemu typów bez polimorficznych wyrażeń let przez Hindley'a (1969) i rozszerzony do pełnego systemu przez Damas'a i Milner'a (1982).

3 Algorytm dedukcji typów

Celem tego rozdziału jest naszkicowanie dowodu twierdzenia Hindley-Damas-Milner'a podanego na slajdzie 37, przez opisanie algorytmu, *pt*, do decydowania o typowalności i zwracania najbardziej ogólnego schematu typu. Algorytm *pt* jest zdefiniowany rekurencyjnie, podążając za strukturą wyrażeń (a jego zakończenie jest dowodzone przez indukcję na strukturze wyrażeń). Jak przykłady w rozdziale 2.2 powinny sugerować, algorytm znacząco zależy od *unifikacji*—faktu, że rozwiązywalność skończonego zbioru równań między algebraicznymi frazami jest decydowalna oraz że istnieje najbardziej ogólne rozwiązanie, jeśli tylko jakieś rozwiązanie istnieje. Ten fakt został odkryty przez Robinson'a (1965) i stał się kluczowym składnikiem w wielu odnoszących się do logiki obszarach informatyki (można tu wymienić trzy przykłady: automatyczne dowodzenie twierdzeń, programowanie w logice, i oczywiście systemy typów). Formę algorytmu unifikacji, *nou*, o którą nam chodzi przedstawia slajd 38. Jakkolwiek nie będziemy

Rysunek 37: Twierdzenie (Hindley; Damas-Milner)

Jeśli domknięte wyrażenie Mini-ML jest typowalne (tj. zachodzi $\vdash M : \sigma$ dla jakiegoś schematu typów σ), to istnieje pryncypalny schemat typów dla M .

Rzeczywiście, istnieje algorytm, który mając dane M jako parametr wejściowy, zdecydować czy lub nie wyrażenie to jest typowalne, oraz zwróci pryncypalny schemat typów jeśli zachodzi to pierwsze.

Rysunek 38: Unifikacja typów ML

Istnieje algorytm *nou* który mając podane dwa typy Mini-ML t_1 i t_2 zdecydować czy t_1 i t_2 są *unifikowalne* (ang. *unifiable*), tj. czy istnieje zastąpienie typu $S \in \text{Sub}$ takie że

$$(a) \ S(t_1) = S(t_2)$$

Co więcej, jeśli one są unifikowalne, $nou(t_1, t_2)$ zwróci *najbardziej ogólny unifikator*— S satysfakcjonujący zarówno (a) jak i

$$(b) \ \text{dla wszystkich } S' \in \text{Sub}, \text{ jeśli } S'(t_1) = S'(t_2), \text{ to wtedy } S' = TS \text{ dla jakiegoś } T \in \text{Sub}.$$

Przyjmujemy, że $nou(t_1, t_2) = \text{FAIL}$ jeśli (i tylko wtedy) gdy t_1 i t_2 nie są unifikowalne.

przycząca tutaj implementacji algorytmu *nou*, musimy przynajmniej wyjaśnić notację dla zastąpienia typów, wprowadzoną na slajdzie 38.

Definicja 1 (Zastąpienia typów). Zastąpienie typów (ang. *type substitution*) S jest (całkowicie zdefiniowaną) funkcją ze zmiennych typu do typów języka Mini-ML, z własnością że $S(\alpha) = \alpha$ dla wszystkich oprócz skończonej liczby zmiennych α .

Piszemy Sub mając na myśli zbiór wszystkich takich funkcji. Dziedzina zbioru $S \in \text{Sub}$ jest to skończony zbiór zmiennych

$$\text{dom}(S) \stackrel{\text{def}}{=} \{\alpha \in \text{TyVar} \mid S(\alpha) \neq \alpha\}$$

Mając dane zastąpienie typów S , efekt zastosowania zastąpienia do typu jest zapisywany $S t$; tak więc jeśli $\text{dom}(S) = \{\alpha_1, \dots, \alpha_n\}$ i $S(\alpha_i)$ jest typem t_i dla każdego

Rysunek 39: Pryncypalny σ dla wyrażeń otwartych

Rozwiązaniem dla problemu typowania $\Gamma \vdash M : ?$ jest para $\boxed{(S, \sigma)}$ składająca się z zastąpienia typów S i schematu typów σ spełniająca

$$S \Gamma \vdash M : \sigma$$

(gdzie $S \Gamma = \{x_1 : S \sigma_1, \dots, x_n : S \sigma_n\}$, jeśli $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$).

Takie rozwiązanie jest *pryncypalne* jeśli mając dane jakiekolwiek inne rozwiązanie, (S', σ') , istnieje jakieś $T \in \text{Sub}$ z $TS = S'$ i $T\sigma \succ \sigma'$.

$i = 1..n$, wtedy $S t$ jest typem otrzymanym przez równoczesne zastąpienie każdego wystąpienia α_i w t przez t_i (dla wszystkich $i = 1..n$), tj.

$$S t = t [t_1/\alpha_1, \dots, t_n/\alpha_n]$$

używając notacji dla zastąpienia jak na slajdzie 24. Nie odrzucając notacji po prawej stronie powyższej formuły, będziemy woleli pisać zastosowanie funkcji zastąpienia typów S po lewej stronie typu do którego ta funkcja jest zastosowana (czyli $S t$ zamiast $t S$).

Jako rezultat, *kompozycja* TS dwóch zastąpień typów $S, T \in \text{Sub}$ oznacza najpierw zastosuj S a potem T . Wtedy z definicji TS jest funkcją odwzorowującą każdą zmienną typu α typowi $T(S(\alpha))$ (stosujemy zastąpienie typu T do typu $S(\alpha)$). Zauważmy, że funkcja TS spełnia warunek skończoności wymagany przez zastąpienie i mamy $TS \in \text{Sub}$; faktycznie, $\text{dom}(TS) \subseteq \text{dom}(T) \cup \text{dom}(S)$.

Bardziej ogólnie, jeśli $\text{dom}(S) = \{\alpha_1, \dots, \alpha_n\}$ i σ jest schematem typu Mini-ML, to wtedy $S \sigma$ będzie oznaczać rezultat podmiany każdego wolnego wystąpienia α_i w σ przez $S(\alpha_i)$ (dla $i = 1..n$), zmieniając odpowiednio nazwy zmiennych tam gdzie to konieczne.

Nawet jeśli ostatecznie zainteresowani jesteśmy typowalnością wyrażeń *domkniętych*, ponieważ algorytm *pt* schodzi rekurencyjnie przez podwyrażenia wyrażenia wejściowego, nieuchronnie musi on wygenerować typowania dla wyrażeń z wolnymi zmiennymi. Dlatego więc musimy zdefiniować znaczenie typowalności i pryncypalnych schematów typów dla wyrażeń otwartych w obecności niepustego środowiska typowania. Wynik mamy na slajdzie 39. Aby obliczyć pryncypalne schematy typów, wystarczy obliczyć 'pryncypalne rozwiązania' w sensie slajdu 39: dlatego że, jeśli M jest w rzeczywistości domknięte, to wtedy jakiekolwiek pryncypalne rozwiązanie (S, σ) dla problemu typowania $\{\} \vdash M : ?$ ma własność taką że σ jest pryncypalnym schematem typów dla M w sensie slajdu 36.

Rysunek 40: Specyfikacja dla algorytmu typowania pt

pt operuje na problemach typowalności $\Gamma \vdash M : ?$ (Γ - środowisko typowania, M - wyrażenie języka Mini-ML).

pt albo zwraca parę (S, t) składającą się z zastąpienia typów $S \in \text{Sub}$ oraz typu t Mini-ML, albo generuje wyjątek FAIL.

- jeśli $\Gamma \vdash M : ?$ ma rozwiązanie (zob. slajd 39), wtedy $pt(\Gamma \vdash M : ?)$ zwraca (S, t) dla jakiegoś S i t ;
co więcej, ustalając $A = \{wzt(t) - wzt(S \Gamma)\}$, mamy że $(S, \forall A(t))$ jest pryncypalnym rozwiązaniem dla problemu $\Gamma \vdash M : ?$.
 - jeśli $\Gamma \vdash M : ?$ nie ma rozwiązania, to wtedy $pt(\Gamma \vdash M : ?)$ zwraca FAIL.
-

Slajd 40 zawiera więcej szczegółów na temat tego, co jest wymagane od algorytmu pryncypalnego typowania, pt . Jeden z możliwych algorytmów tego rodzaju, opisany nieformalnym pseudo-kodem (z pominięciem przypadków dla `nil`, `cons` i wyrażień `match`) został naszkicowany na slajdach 41 i 42. Proszę zauważyć następujące punkty w odniesieniu do tych definicji:

1. Ukrycie zakładamy że wszystkie związane zmienne w wyrażeniach i związane zmienne typu w schematach typów różnią się jedne od drugich, oraz od jakichkolwiek innych zmiennych w kontekście. Tak więc, dla przykładu, klauzula dla abstrakcji funkcji po cichu zakłada że $x \notin \text{dom}(\Gamma)$; oraz klauzula dla zmiennych zakłada że $A \cap \text{wzt}(\Gamma) = \{\}$.
2. Zastąpienie typów Id występujące w klauzulach dla zmiennych i dla wartości Boolean jest zastąpieniem *identycznym*, które odwzorowuje każdą zmienną typu α na siebie.
3. Nie pokazaliśmy klauzul definicji dla `nil`, `cons`, i wyrażień `match`.
4. Nie daliśmy dowodu że definicja na slajdzie 42 jest prawidłowa (tj. że spełnia specyfikację na slajdzie 40). Prawidłowość algorytmu zależy od istotnej własności typowania Mini-ML: a mianowicie, że *typowanie jest zachowane przez operację zastąpienia zmiennych typów przez typy*.

Bardziej efektywne algorytmy wykorzystują inne podejście do zastąpienia i unifikacji, bazujące na relacjach równoważności (ang. equivalence relations) kierunkowych

Rysunek 41: Wybrane klauzule w definicji algorytmu pt (1/2)

- *Abstrakcje funkcji:* $pt(\Gamma \vdash \lambda x(M) : ?) \stackrel{def}{=}$

let $\alpha = \text{fresh}$ **in**
let $(S, t) = pt(\Gamma, x : \alpha \vdash M : ?)$ **in** $(S, S(\alpha) \rightarrow t)$

- *Aplikacje funkcji:* $pt(\Gamma \vdash M_1 M_2 : ?) \stackrel{def}{=}$

let $(S_1, t_1) = pt(\Gamma \vdash M_1 : ?)$ **in**
let $(S_2, t_2) = pt(S_1 \Gamma \vdash M_2 : ?)$ **in**
let $\alpha = \text{fresh}$ **in**
let $S_3 = \text{nou}(S_2 t_1, t_2 \rightarrow \alpha)$ **in** $(S_3 S_2 S_1, S_3(\alpha))$

acyklicznych grafów oraz algorytmach znajdowania unii; zob. np. (Rémy 2002, rozdział 2.4.2). W tym odnośniku, a także w książce Pierce’a (Pierce 2002, Rozdział 22.3), możemy zobaczyć podejście do algorytmów dedukcji typów, które prezentuje je jako część bardziej ogólnego problemu generowania i rozwiązywania *problemów ograniczeń* (ang. *constraints problems*). Wydaje się, że jest to podejście owocne, gdyż obejmuje szeroką klasę różnych algorytmów dedukcji typów.

4 Polimorficzne typy referencji

4.1 Problem

Przypomnijmy sobie ze Wstępu, że ważnym celem systemu typów jest zapewnienie *bezpieczeństwa* (slajd 10) poprzez rezultaty *niezawodności typowania* (slajd 11). Nawet jeśli język programowania jest pomyślany jako bezpieczny przez siłę swojego systemu typów, to wciąż może się zdarzyć że oddzielne cechy tego języka, każda z nich pożądana z osobna, mogą składać się razem w nieprzewidziany sposób, produkując zawodny system typów. W tym rozdziale spojrzymy na przykład tego zjawiska, który wystąpił podczas opracowywania rodziny języków ML. Dwie cechy języka które składają się w nieprzyjemny sposób to:

- ML-owy styl implikatywnie typowanego **let**-związanego polimorfizmu, oraz
- typy referencji.

Rysunek 42: Wybrane klauzule w definicji algorytmu pt (2/2)

- **Zmienne:** $pt(\Gamma \vdash x : ?) \stackrel{def}{=} \text{let } \forall A(t) = \Gamma(x) \text{ in } (Id, t)$
 - **Wyrażenia let:** $pt(\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : ?) \stackrel{def}{=} \text{let } (S_1, t_1) = pt(\Gamma \vdash M_1 : ?) \text{ in}$
 $\text{let } A = wzt(t_1) - wzt(S_1 \Gamma) \text{ in}$
 $\text{let } (S_2, t_2) = pt(S_1 \Gamma, x : \forall A(t_1) \vdash M_2 : ?) \text{ in } (S_2 S_1, t_2)$
 - **Boolean** ($B = \text{true}, \text{false}$): $pt(\Gamma \vdash B : ?) \stackrel{def}{=} (Id, bool)$
 - **Warunki:** $pt(\Gamma \vdash \text{if } M_1 \text{ then } M_2 \text{ else } M_3 : ?) \stackrel{def}{=} \text{let } (S_1, t_1) = pt(\Gamma \vdash M_1 : ?) \text{ in}$
 $\text{let } S_2 = nou(t_1, bool) \text{ in}$
 $\text{let } (S_3, t_3) = pt(S_2 S_1 \Gamma \vdash M_2 : ?) \text{ in}$
 $\text{let } (S_4, t_4) = pt(S_3 S_2 S_1 \Gamma \vdash M_3 : ?) \text{ in}$
 $\text{let } S_5 = nou(S_4 t_3, t_4) \text{ in } (S_5 S_4 S_3 S_2 S_1, S_5 t_4)$
-

Rysunek 43: Typy i wyrażenia w ML dla referencji
Typy

$t ::= \dots$	zob. slajd 23
$unit$	typ elementu (ang. unit type)
$t\ ref$	typ referencji

Wyrażenia

$M ::= \dots$	zob. slajd 26
$()$	element (ang. unit)
$ref\ M$	utworzenie referencji
$!M$	dereferencja (odczyt)
$M := M$	przypisanie

Pierwsze zagadnienie już omawialiśmy w rozdziale 2. Drugie zagadnienie dotyczy cech imperatywnych języka ML, które bazują na zdolności do dynamicznego tworzenia widzianych lokalnie lokalizacji pamięci, które mogą być zapisywane i odczytywane. Rozpocznijemy przez podanie dla nich składni i reguł typowania. Rozszerzymy gramatykę dla typów Mini-ML (slajd 23) o typ wartości elementu (tj. typ z pojedynczą wartością “unit”) oraz typy (modyfikowalnych) *referencji* (ang. *reference types*); i odpowiednio, rozszerzymy gramatykę dla wyrażeń Mini-ML (slajd 26) o wartość elementu (ang. unit value), oraz operacje do tworzenia, odczytywania i przypisywania referencji. Te dodatki są pokazane na slajdzie 43. Nazwijmy tak otrzymany język *Midi-ML*. Reguły typowania dla tych nowych form wyrażeń podano na slajdzie 44.

Przykład Oto przykład użycia reguł typowania ze slajdu 44. Wyrażenie podane na slajdzie 45 ma typ *unit*.

Dowód. Można to wydedukować przez aplikowanie reguły **let** (slajd 30) to stwierdzeń:

$$\begin{aligned} & \{\} \vdash \mathbf{ref}\ \lambda x(x) : (\alpha \rightarrow \alpha)\ ref \\ & r : \forall \alpha ((\alpha \rightarrow \alpha)\ ref) \vdash \mathbf{let}\ u = (r := \lambda x'(\mathbf{ref}\ !x'))\ \mathbf{in}\ (!r)() : unit . \end{aligned}$$

Pierwsze ze stwierdzeń ma następujący dowód:

$$\frac{\frac{\frac{}{} (\mathbf{var}\ \succ)}{x : \alpha \vdash x : \alpha} (\mathbf{fn})}{\{\} \vdash \lambda x(x) : \alpha \rightarrow \alpha} (\mathbf{ref})}{\{\} \vdash \mathbf{ref}\ \lambda x(x) : (\alpha \rightarrow \alpha)\ ref} (\mathbf{ref})$$

Rysunek 44: Dodatkowe reguły typowania dla Midi-ML

(unit) $\Gamma \vdash () : unit$

(ref)
$$\frac{\Gamma \vdash M : t}{\Gamma \vdash \mathbf{ref} M : t \mathit{ref}}$$

(get)
$$\frac{\Gamma \vdash M : t \mathit{ref}}{\Gamma \vdash !M : t}$$

(set)
$$\frac{\Gamma \vdash M_1 : t \mathit{ref} \quad \Gamma \vdash M_2 : t}{\Gamma \vdash M_1 := M_2 : unit}$$

Rysunek 45: Przykład referencji

To wyrażenie

$$\begin{aligned} &\mathbf{let} \ r = \mathbf{ref} \ \lambda x(x) \ \mathbf{in} \\ &\quad \mathbf{let} \ u = (r := \lambda x'(\mathbf{ref} \ !x')) \ \mathbf{in} \\ &\quad \quad (!r)() \end{aligned}$$

ma typ $unit$.

Drugie ze stwierdzeń może być udowodnione przez aplikowanie reguły `let` do

$$r : \forall\alpha((\alpha \rightarrow \alpha) \text{ ref}) \vdash r := \lambda x'(\mathbf{ref} !x') : \text{unit} \quad (5)$$

$$r : \forall\alpha((\alpha \rightarrow \alpha) \text{ ref}), u : \text{unit} \vdash (!r)() : \text{unit} . \quad (6)$$

Pisząc Γ dla środowiska typowania $\{r : \forall\alpha((\alpha \rightarrow \alpha) \text{ ref})\}$, dowód (5) jest następujący

$$\frac{\frac{\frac{\frac{\Gamma, x' : \alpha \text{ ref} \vdash x' : \alpha \text{ ref}}{\Gamma, x' : \alpha \text{ ref} \vdash !x' : \alpha} \text{(get)}}{\Gamma, x' : \alpha \text{ ref} \vdash \mathbf{ref} !x' : \alpha \text{ ref}} \text{(ref)}}{\Gamma \vdash \lambda x'(\mathbf{ref} !x') : \alpha \text{ ref} \rightarrow \alpha \text{ ref}} \text{(fn)}}{\Gamma \vdash r : (\alpha \text{ ref} \rightarrow \alpha \text{ ref}) \text{ ref}} \text{(var } \succ) \quad \frac{\Gamma \vdash \lambda x'(\mathbf{ref} !x') : \alpha \text{ ref} \rightarrow \alpha \text{ ref}}{\Gamma \vdash r := \lambda x'(\mathbf{ref} !x') : \text{unit}} \text{(set)}$$

Podczas gdy dowód (6) to

$$\frac{\frac{\frac{\Gamma, u : \text{unit} \vdash r : (\text{unit} : \text{unit}) \text{ ref}}{\Gamma, u : \text{unit} \vdash !r : \text{unit} \rightarrow \text{unit}} \text{(get)}}{\Gamma, u : \text{unit} \vdash (!r)() : \text{unit}} \text{(ap)}}{\Gamma, u : \text{unit} \vdash r : (\text{unit} : \text{unit}) \text{ ref}} \text{(var } \succ)$$

□

Jakkolwiek reguły typowania dla referencji wydają się być bez zarzutu, to jednak w połączeniu z poprzednimi regułami typowania, a w szczególności z regułą (`let`), produkują one system typów dla którego niezawodność systemu typów nie sprawdza się w odniesieniu do semantyki operacyjnej języka ML. Aby przekonać się o co chodzi, rozważmy co będzie się działo kiedy wyrażenie na slajdzie 45, nazwijmy je M , jest ewaluowane.

Ewaluacja (wykonanie) najbardziej zewnętrznego przypisania `let` w M utworzy świeżą lokalizację pamięci przypisaną r i zawierającą wartość $\lambda x(x)$. Ewaluacja drugiego `let`-związania uaktualni zawartość r do wartości $\lambda x'(\mathbf{ref} !x')$ i zwiąże wartość elementu (unit value) z u ¹. Następnie $(!r)()$ jest ewaluowane. To oznacza aplikowanie bieżącej zawartości r , to jest funkcji $\lambda x'(\mathbf{ref} !x')$, do wartości $()$. To natomiast powoduje w rezultacie próbę ewaluowania $!()$, tj. dereferencji czegoś co nie jest lokalizacją pamięci, a więc próbę wykonania niebezpiecznej operacji która powinna być uchwycona przed wykonaniem programu. Wyrażając się bardziej formalnie, mamy

$$\langle M, \{\} \rangle \rightarrow \text{FAIL}$$

w systemie tranzycji zdefiniowanym na slajdach 47, 48 oraz na slajdzie 46 (używając stylu “ewaluacyjnego kontekstu” Wright’a i Felleisen’a (1994)). Konfiguracje systemu tranzycji są dwojakiego rodzaju:

¹Ponieważ zmienna u nie pojawia się w ciele wyrażenia M , najbardziej wewnętrzne wyrażenie `let` w M jest jedynie sposobem zakodowania sekwencji $(r := \lambda x'(\mathbf{ref} !x')); (!r)()$ we fragmencie języka ML który używamy celem ilustracji przedstawianych zagadnień.

Rysunek 46: Tranzycje Midi-ML włączające referencje (1/3)

$$\begin{aligned}
\langle !x, s \rangle &\rightarrow \langle s(x), s \rangle && \text{jeśli } x \in \text{dom}(s) \\
\langle !V, s \rangle &\rightarrow \text{FAIL} && \text{jeśli } V \text{ nie jest zmienną} \\
\langle x := V', s \rangle &\rightarrow \langle (), s[x \mapsto V'] \rangle \\
\langle V := V', s \rangle &\rightarrow \text{FAIL} && \text{jeśli } V \text{ nie jest zmienną} \\
\langle \text{ref } V, s \rangle &\rightarrow \langle x, s[x \mapsto V] \rangle && \text{gdzie } x \notin \text{dom}(s)
\end{aligned}$$

gdzie V zmienia się po wartościach:

$$V ::= x \mid \lambda x(M) \mid () \mid \text{true} \mid \text{false} \mid \text{nil} \mid V :: V$$

- Para $\langle M, s \rangle$, gdzie M jest wyrażeniem ML, zaś s jest *stanem* (ang. *state*), zdefiniowanym jako skończona funkcja przyporządkowująca zmiennym, x , (tutaj będącymi używane jako lokalizacje pamięci) syntaktyczne *wartości* (ang. *values*), V . (Możliwe formaty V dla naszego fragmentu języka ML są zdefiniowane na slajdzie 46.) Co więcej, aby taka para mogła być konfiguracją, musi być spełniony następujący warunek “dobrego uformowania”: zmienne wolne wyrażenia M oraz każdej wartości $s(x)$ (gdzie x przyjmuje wartości z $\text{dom}(s)$), powinny być zawarte w skończonym zbiorze $\text{dom}(s)$.

- Symbol FAIL, reprezentujący błąd w czasie wykonania (ang. run-time error).

(Notacja $s[x \mapsto V]$ użyta na slajdzie 46 oznacza stan o dziedzinie zdefiniowanej jako $\text{dom}(s) \cup \{x\}$, odzwierciedlający x do V , a w innym przypadku zachowujący się jak s .)

Aksjomaty i reguły indukcyjnie definiujące system tranzycji dla Midi-ML są przedstawione na slajdach 46-48.

4.2 Przywrócenie niezawodności systemu typów

Sedno problemu opisanego w poprzednim podrozdziale leży w fakcie, że typowanie wyrażeń takich jak `let $r = \text{ref } M_1$ in M_2` używając reguły (let) pozwala lokalizacji pamięci (związanej z) r posiadać schemat typów σ generalizujący typ referencji typu

Rysunek 47: System tranzycji dla języka Midi-ML (2/3)

- $\langle \text{if true then } M_1 \text{ else } M_2, s \rangle \rightarrow \langle M_1, s \rangle$
 - $\langle \text{if false then } M_1 \text{ else } M_2, s \rangle \rightarrow \langle M_2, s \rangle$
 - $\langle \text{if } V \text{ then } M_1 \text{ else } M_2, s \rangle \rightarrow \text{FAIL}$, jeśli $V \notin \{\text{true, false}\}$
 - $\langle (\lambda x(M)) V', s \rangle \rightarrow \langle M[V'/x], s \rangle$
 - $\langle V V', s \rangle \rightarrow \text{FAIL}$, jeśli V nie jest abstrakcją funkcji
 - $\langle \text{let } x = V \text{ in } M, s \rangle \rightarrow \langle M[V/x], s \rangle$
 - $\langle \text{match nil with nil} \Rightarrow M \mid x_1 :: x_2 \Rightarrow M', s \rangle \rightarrow \langle M, s \rangle$
 - $\langle \text{match } V_1 :: V_2 \text{ with nil} \Rightarrow M \mid x_1 :: x_2 \Rightarrow M', s \rangle \rightarrow \langle M'[V_1/x_1, V_2/x_2], s \rangle$
 - $\langle \text{match } V \text{ with nil} \Rightarrow M \mid x_1 :: x_2 \Rightarrow M', s \rangle \rightarrow \text{FAIL}$, jeśli V nie jest ani nil ani cons-wartością
-

Rysunek 48: System tranzycji dla języka Midi-ML (3/3)

$$\bullet \frac{\langle M, s \rangle \rightarrow \langle M', s' \rangle}{\langle \mathcal{E}[M], s \rangle \rightarrow \langle \mathcal{E}[M'], s' \rangle} \quad \bullet \frac{\langle M, s \rangle \rightarrow \text{FAIL}}{\langle \mathcal{E}[M], s \rangle \rightarrow \text{FAIL}}$$

gdzie V przyjmuje wartości (ang. *values*):

$$V ::= x \mid \lambda x(M) \mid () \mid \text{true} \mid \text{false} \mid \text{nil} \mid V :: V$$

\mathcal{E} przyjmuje konteksty ewaluacyjne (ang. *evaluation contexts*):

$$\begin{aligned} \mathcal{E} ::= & - \mid \text{if } \mathcal{E} \text{ then } M \text{ else } M \mid \mathcal{E} M \mid V \mathcal{E} \mid \text{let } x = \mathcal{E} \text{ in } M \\ & \mid \mathcal{E} :: M \mid V :: \mathcal{E} \mid \text{match } \mathcal{E} \text{ with nil} \Rightarrow M' \mid x :: x \Rightarrow M \\ & \mid \text{ref } \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} := M \mid V := \mathcal{E} \end{aligned}$$

i $\mathcal{E}[M]$ oznacza wyrażenie w języku Midi-ML, które powstało przez zamianę wszystkich wystąpień “-” w \mathcal{E} na M .

wyrażenia M_1 . Wystąpienia r w M_2 odnoszą się do tej samej, dzielonej lokalizacji, i ewaluacja M_2 może spowodować przypisania do tej dzielonej lokalizacji, co ogranicza możliwy typ dalszych wystąpień r . Ale z drugiej strony reguła typowania pozwala tym wszystkim wystąpieniom r mieć *dowolny* typ który jest specjalizacją schematu σ , i to właśnie może prowadzić do przypisania typów do wyrażeń które nie są bezpieczne, tak jak to widzieliśmy w naszym przykładzie.

Możemy uniknąć tego problemu przez wymyślenie systemu typów, który zapobiega generalizacji zmiennych typu występujących w typach dzielonych lokalizacji pamięci. Kilka sposobów takiego działania zostało zaproponowanych w literaturze: zob. ich przegląd w (Wright 1995). Jedno z rozwiązań zaadaptowane w oryginalnej definicji Standardowego ML z 1990 roku (Milner, Tofte, i Harper 1990), zostało zaproponowane przez Tofte’go (1990). Zakładało ono podzielenie zbioru zmiennych typu na dwa (przeliczalne, nieskończone) połowy, “aplikatywne zmienne typu” (przybierające nazwy α) oraz “imperatywne zmienne typu” (nazwy $_ \alpha$). Wtedy reguła (ref) jest ograniczona przez wymóg aby t jedynie zaprzęgało imperatywne zmienne typu; innymi słowy pryncypalny schemat typu wyrażenia $\lambda x(\text{ref } x)$ staje się $\forall _ \alpha(_ \alpha \rightarrow _ \alpha \text{ ref})$, raczej niż $\forall \alpha(\alpha \rightarrow \alpha \text{ ref})$. Ponadto, i co najistotniejsze, reguła (let) (slajd 30) jest ograniczona przez wymaganie aby kiedy schemat typu $\sigma = \forall A(t)$ przypisany do wyrażenia M_1 jest taki, że A zawiera imperatywne zmienne typu, to M_1 musi być wtedy wartością (a więc w szczególności jego ewaluacja nie tworzy świeżej lokalizacji pamięci).

To rozwiązanie tę ma zaletę, że w tym nowym systemie typów typowalność wy-

Rysunek 49: Warunkowa reguła typowania dla let-wyrażeń

$$(letv) \quad \frac{\Gamma \vdash M_1 : t_1 \quad \Gamma, x : \forall A(t_1) \vdash M_2 : t_2}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2 : t_2} \quad (*)$$

(*) jeśli jest spełnione że $x \notin \text{dom}(\Gamma)$ oraz

$$A = \begin{cases} \{\} & \text{gdy } M_1 \text{ nie jest wartością,} \\ \text{wzt}(t_1) - \text{wzt}(\Gamma) & \text{gdy } M_1 \text{ jest wartością.} \end{cases}$$

(Przypomnijmy, że wartości zdefiniowaliśmy następująco

$V ::= x \mid \lambda x(M) \mid () \mid \text{true} \mid \text{false} \mid \text{nil} \mid V :: V.$)

rażeń nie zaprzęgających referencji jest dokładnie taka sama jak w starym systemie. Jednakże, ma też i wadę, że system typów czyni niestety różnicę między wyrażeniami które pod względem zachowania są równoważne (tj. takie które powinny być kontekstowo ekwiwalentne). Dla przykładu, jest wiele funkcji operujących na listach, które mogą być zdefiniowane w czysto funkcyjnym fragmencie języka ML przez rekurencyjne definicje, ale które mają bardziej efektywne definicje używając lokalnych referencji. Niestety, jeśli schemat typów poprzedniej definicji jest czymś na kształt $\forall \alpha (\alpha \text{ list} \rightarrow \alpha \text{ list})$, to schematem typów tej ostatniej może równie dobrze być inny schemat typów $\forall _ \alpha (_ \alpha \text{ list} \rightarrow _ \alpha \text{ list})$. Tak więc nie będziemy w stanie używać dwóch wersji takiej funkcji zamiennie!

Autorzy ponownie przemyślanej definicji Standardowego ML (Milner, Tofte, Harper, MacQueen), z roku 1996, zaadoptowali prostsze rozwiązanie, zaproponowane niezależnie przez Wright'a (1995). Usuwa ono rozróżnienie między aplikatywnymi i imperatywnymi zmiennymi typu (w efekcie, wszystkie zmienne typu są imperatywne, choć zwyczajowo używane są symbole α, α'), podczas gdy utrzymana jest ograniczona do wartości formuła reguły (let), tak jak ją pokazano na slajdzie 49. Tak więc nasza wersja tego systemu typów bazuje dokładnie na tej samej formule typu, schematu typów, oraz środowiska typowania jak wcześniej, z relacją typowania generowaną indukcyjnie przez aksjomaty i reguły na slajdzie 27–30 i 44, za wyjątkiem że aplikatywność reguły (let) jest ograniczona, w taki sposób jak pokazano na slajdzie 49.

Przykład Wyrażenie na slajdzie 45 nie jest typowalne w systemie typów dla Midi-ML, otrzymanym w wyniku zastąpienia reguły (let) przez warunkową, ograniczającą wartość regułę (letv) zdefiniowaną na slajdzie 49 (utrzymując nie zmienione wszystkie

inne aksjomaty i reguły).

Dowód. Z uwagi na format tego wyrażenia, ostatnia reguła użyta w dowodzie jego typowości musi kończyć się użyciem (`letv`). Z powodu ubocznego warunku w tej regule oraz faktu że `ref λx(x)` nie jest wartością, reguła musi być aplikowana z $A = \{\}$. To powoduje próbę typowania

$$\text{let } u = (r := \lambda x'(\text{ref } !x')) \text{ in } (!r)() \quad (7)$$

w środowisku typowania $\Gamma = \{r : (\alpha \rightarrow \alpha) \text{ ref}\}$. Ale to nie jest możliwe, ponieważ zmienna typu α nie jest uniwersalnie kwantyfikowana ($\forall \alpha$) w tym środowisku, podczas gdy dwie instancje zmiennej r w (7) mają dwa różne implikowane typy (a mianowicie $(\alpha \text{ ref} \rightarrow \alpha \text{ ref}) \text{ ref}$ oraz $(\text{unit} \rightarrow \text{unit}) \text{ ref}$). \square

Powyższy przykład jest w całkowitym porządku, ale skąd mamy wiedzieć, że osiągnęliśmy pełne bezpieczeństwo z tym systemem typów dla języka Midi-ML? Odpowiedź leży w formalnym dowodzie własności *niezawodności systemu typów* (ang. *type soundness*), zdefiniowanej na slajdzie 11. Aby udowodnić ten rezultat, należy wprawdzie sformułować definicję typowania dla ogólnych konfiguracji $\langle M, s \rangle$ kiedy stan s nie jest pusty, i wówczas pokazać że

- typowanie jest utrzymane przez kroki tranzycji, \rightarrow ;
- jeśli konfiguracja może być typowana, to nie może pozwolić na tranzycję (przejście) do stanu FAIL.

A więc sekwencja tranzycji od tego rodzaju dobrze typowanej konfiguracji nigdy nie może prowadzić do konfiguracji FAIL. Nie mamy czasu aby podać szczegóły takiego dowodu w tym cyklu wykładów: osoby zainteresowane odsyłamy do (Wright, Felleisen 1994; Harper 1994), gdzie są przykłady podobnych rezultatów niezawodności systemu typów.

Jakkolwiek reguła typowania (`letv`) rzeczywiście pozwala na osiągnięcie niezawodności systemu typów dla polimorficznych referencji w przyjemnie łatwy sposób, to jednak nie oznacza to, że pewne wyrażenia nie zawierające referencji które są typowane w oryginalnym systemie typów języka ML są wciąż typowane w systemie z warunkową regułą (`letv`). Wright (1995, rozdziały 3.2 i 3.3) przeanalizował tego konsekwencje, i wykazał, że (na szczęście) nie jest to zawadą w używaniu Standardowego ML w praktyce.

Rysunek 50: Niezawodność systemu typów dla Midi-ML

Dla dowolnego domkniętego wyrażenia M w języku Midi-ML, jeśli istnieje jakiś schemat typów σ dla którego

$$\vdash M : \sigma$$

jest udowodnialne w systemie typów z warunkową regułą typowania dla let (aksjomaty i reguły na Slajdach 27–29, 44 i 49), to wtedy *ewaluacja* M nie prowadzi do *błędu*, t.j. nie istnieje sekwencja tranzycji w formie

$$\langle M, \{\} \rangle \rightarrow \dots \rightarrow \text{FAIL}$$

dla systemu tranzycji \rightarrow zdefiniowanego na slajdach 46–48 (gdzie $\{\}$ oznacza stan pusty).
