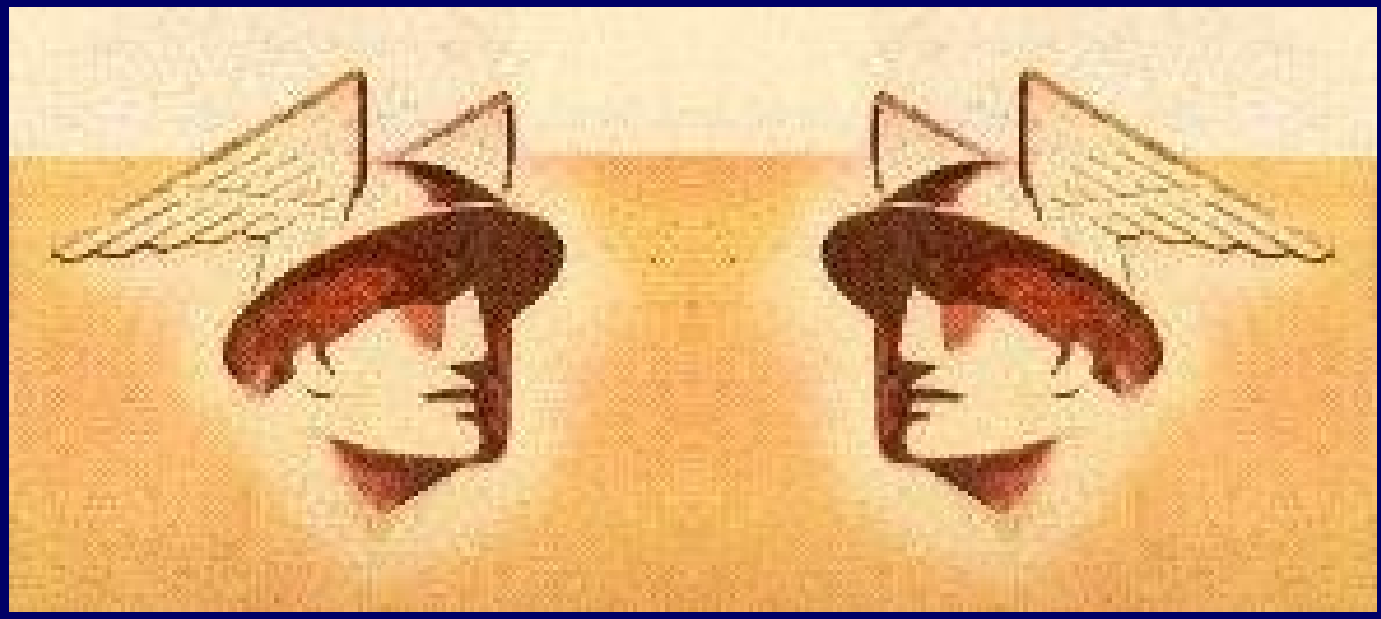


Parallel Mercury

Békés András Georgy

bekesa@sch.bme.hu

Tartalom

- A Mercury nyelv
- I/O Mercuryban
- Párhuzamos végrehajtás
- Független ÉS-párhuzamosság
- Explicit Mercury-szálak

Mercury

- A Mercury nyelv egy Prolog-variáns.
- Célja: “nagyhatékonyságú logikai programozás”
  - tisztán deklaratív logikai nyelv, plusz függvények
  - kötelező deklarációk => dokumentáció, csapatmunka
  - a deklarációk szigorú ellenőrzése => hibák korai felderítése
  - hatékony implementáció
- A Merbourne-i egyetemen fejlesztik, elérhető a GNU GPL licence alatt.
- A nyelv és a fordító együtt feljődik, 2002-ben készült hozzá egy kiegészítés a párhuzamos programozás támogatására.

Mercury

- A hagyományos logikai nyelvekkel probléma, hogy
- A fordító nagyon kevés hibát képes felderíteni. Ha egy Prolog program lefordul, az még nagyon messze áll attól, hogy helyesen fusson.
- Az implementációk lassúak. Azokban a projektekben, ahol nem egyértelműen előnyösebb a logikai programozás, és a sebesség számít, ott szóba sem kerül a Prolog.
- Sok a nemdeklaratív nyelvi elem: Dinamikus tudásbázis (assert-ek), globális, változtatható változók (blackboard), vágók, Input/Output műveletek. A program már nem értelmezhető a logikai jelentése alapján.

Mercury

A Mercuryban pedig:

- Kötelező és ellenőrzött típus-, mód- és determinizmus-deklarációk. Sokkal több hiba kiderül már fordításkor.
- A fordító C nyelvre, mint “hordozható assembly”-re fordít. A sok deklarációnak köszönhetően a fordító nagyon sok optimalizációra képes.
- Tisztán deklaratív nyelv: A programot megérteni és helyességét bizonyítani könnyebb.

Típusdeklaráció

- Típus: egy eljárás argumentumainak típusa van. Például: `int`, `List(int)`, `List(T)`.
- A típusrendszer nagyon hasonlít a Haskell vagy az ML típusrendszeréhez.
- Az eljárások, függvények típusát kötelező deklarálni. Például:
  ```
  :- pred append(list(T), list(T), list(T)).
  ```
- A fordító ellenőrzi a deklarált és a levezetett típusok kompatibilitását.

Móddeklaráció

- Mód: egy eljárás argumentumainak módja az argumentum állapota: `Input` (híváskor be van helyettesítve), `Output` (az eljárás helyettesíti be)
- Egy predikátumnak több módja lehet. Például:

```
:- mode append(in, in, out).
:- mode append(in, in, in).
:- mode append(out, out, in).
```

- Egy predikátum kódját csak egyszer kell megírni, de meg kell mondani, hogy milyen módokban akarjuk használni.
- A fordító minden módhoz külön kódot fordít(hat). A fordító átrendez(het)i a hívásokat úgy, hogy minden híváshoz találjon megfelelő módot. Ha nem sikerül, az fordítási hiba.

Determinizmus

- Determinizmus: Egy predikátum különböző módokban különböző számszor sikerülhet. Például:
  `det`: pontosan egyszer sikerül
  `semidet`: egyszer sikerül vagy egyszer sem
  `multi`: legalább egyszer sikerül
  `nondet`: egyszer sem, vagy akárhányszor sikerülhet

- Minden módhoz meg kell adni a determinizmusát. Például:
  ```
  :- mode append(in, in, out) is det.
  :- mode append(in, in, in) is semidet.
  :- mode append(out, out, in) is multi.
  ```
  A törzsben található hívások determinizmusából levezethető a determinizmus. Az inkompatibilitás fordítási hibát jelent.

Nemdeklaratív műveletek Mercuryban

- Új fogalom: `di` (destructive input) és `uo` (unique output) módok
- `di`: A predikátum felhasználja a változó értékét, de azt megsemmisíti, ezért azt senki más fel nem használhatja. A fordító garantálja, hogy az ilyen változó az utolsó (egyetlen) referencia az értékre.
- `uo`: Az előállított változó egyedi, vagyis a változó az egyetlen referencia az előállított értékre.

Nemdeklaratív műveletek Mercuryban

- Új fogalom: `cc_multi` és `cc_nondet` determinizmusok
- cc = comitted choice: egy predikátum, ami többször is sikerülhet, de az eredmények (definíció szerint) ekvivalensek. Az ilyen predikátum (maximum) egyszer sikerül, de nem tudhatjuk, hogy melyik ágon sikerült (de nem is érdekelhet minket, hiszen ekvivalensek)
- Egy `cc_multi` (`cc_nondet`) preikátumból meghívható `multi` (`nondet`) cél, ilyenkor az csak egy megoldást ad vissza. (Ettől lesz az egész predikátum `cc_multi` (`cc_nondet`).)

Nemdeklaratív műveletek Mercuryban

- Új fogalom: "világállapot" (state of the world): `io__state`
- `io__state` csak `di` és `uo` módokkal adható át. Egy, a világ állapotát módosító predikátum végrehajtása megszünteti a világ régi állapotát, más predikátumok abban az állapotban már nem futtathatók. Ezt a fordító a `di` és `uo` módok segítségével garantálja. (Természetesen egy a világ állapotától nem függő predikátum bármikor végrehajtható, hiszen az eredmény csak az agumentumoktól függ.)
- `io__state` csak `det` illetve `cc_multi` predikátumoknak adható át, hiszen a világ korábbi állapotát nem lehet visszaállítani (I/O műveleteket visszavonni) egy visszalépés során.

Nemdeklaratív műveletek Mercuryban

- A világ állapotát módosító predikátum:

```
:-type iot_hasznalo_predikatum_tipus ==
   pred(..., io__state,io__state)

:-mode iot_hasznalo_predikatum(... ,di,uo) is det.
```

vagy:

```
:-mode iot_hasznalo_predikatum(... ,di,uo) is cc_multi.
```

- Egy Mercury program belépési pontja ezért:

```
:-pred main(io__state,io__state).

:-mode main(di,uo) is det.
```

vagy:

```
:-mode main(di,uo) is cc_multi.
```

Párhuzamos végrehajtás

- Engine: végrehajtó gép, ami Mercury célok végrehajtását végzi
- Task: feladat, egy konkrét Mercury cél
- Az alap Mercuryban: egy végrehajtó, ami egy feladatot hajt végre
- Továbbfejlesztés két irányban:
  - olyan végrehajtó, ami konkurrens módon több feladatot hajt végre (saját ütemező)
  - több végrehajtó (több processzoron) egyszerre futtatásának a lehetősége (OS szintű szál, külső ütemező)
- Végeredmény: több végrehajtó gép, amik egyenként több taszkot hajt(hat)nak végre

Párhuzamos végrehajtás

- A végrehajtó gépek az operációs rendszer szintjén szálak (thread), a feladatok egy közös feladattárban vannak, ahonnan a végrehajtók egyet-egyet kiemelnek és futtatnak
- a feladatok egymásra nem lehetnek hatással, például az egyik célban történő változó-lekötés vagy visszalépés nem lehet hatással egy másik célra
- a fordító garantálja, hogy csak egy cél próbál meg egy változót lekötni (ez a Mercury nyelv sajátossága), így a hozzáféréseknél még csak lock-olni sem kell
- A mód-rendszer segítségével megoldható, hogy a fogyasztó célok a termelő célok lefutása után kerülnek végrehajtásra

Független ÉS-párhuzamosság

- Explicit párhuzamos ÉS-operátor: `'&'`
- A deklaratív szemantika szerint azonos a `','` szekvenciális operátorral
- Módhelyesség: egy predikátum módhelyes (well moded), ha a `','` operátorral elválasztott céloknak létezik olyan rendezése, hogy minden változó minden fogyasztó (Input) felhasználása a termelő (Output) felhasználás után van.
- A `'&'` operátorra más szabályok vonatkoznak: `A & B` akkor módhelyes, ha az `A` és `B` által lekötött változók (Output változók) halmaza diszjunkt, azaz `A` nem függ olyan változótól, amit `B` köt le, és viszont. További megkötés, hogy `A` és `B` determinizmusa `det`.

semidet cél párhuzamos futtatása

```
% semidet A esetén A & B helyett:
:- type maybe(T) ---> no ; yes(T).
(
   (
      A-> X=yes({output_of_A});
      X=no
   )&
   B
),
X=yes({.....}),
```

- `A` (gyors) meghiúsulása esetén `B` végrehajtása felesleges, így pazarlás. Ha ez gyakori, akkor inkább a szekvenciális ÉS operátort kell használni!

nondet cél párhuzamos futtatása

```
% nondet A esetén A & B helyett:
( solutions(A,ASolutions) & B ),
member(ASolution,ASolutions)
```

- `A` (gyakori) meghiúsulása esetén `B` végrehajtása itt is pazarlás.
- Viszont (a képzeletbeli) `A & B`-nél sokkal hatékonyabb, mert az `B`-t `A` minden visszalépésekor végrehajtaná. Megj: `A,B` esetnén ugyanez az optimalizálás automatikus: a `A` és `B` függetlesége miatt a fordító `B,A`-vá alakíthatja, hogy `B`-t csak egyszer kelljen végrehajtani.

Explicit Mercury szálak

- A független ÉS-párhuzamosság lehetősége főként szekvenciális programok felgyorsítására való.
- Gyakran hasznos, ha le lehet írni (egymással kommunikáló) párhuzamos programrészeket. Például, ha az algoritmus eleve konkurrens, vagy egy párhuzamos külvilággal kell kapcsolatot tartani (webszerver, adatbázis-szerver, operációs rendszer, GUI).
- A Mercuryban lehetőség van ilyenre. Ez azonban nem nyelvi konstrukció, mint a `&` operátor, hanem függvénykönyvtár.
- A párhuzamosan futó Mercury programrészeket (Mercury-)szálaknak nevezzük.

Kommunikáló párhuzamos célok más nyelvekben

- Az explicit szálak legfőbb előnye a független párhuzamos ÉS operátorhoz képest az, hogy a szálak kommunikálhatnak egymással.
- Más párhuzamos logikai nyelvekben ezt közösen használt változókkal szokták megoldani: egy lekötetlen változót több cél is megkap argumentumként, a szálak progresszív módon kötik le a tartalmát, ill. figyelik a más szálak által végrehajtott lekötéseket.
- Tipikus kommunikációs minta: "stream"-pár, ami nem más, mint egy (nyitott végű) lista-pár. Az egyik listában `A` szál helyezhet el üzenetet `B`-nek, a másikban fordítva. A kommunikáció végeztével a listát lezárják.

Kommunikáló párhuzamos célok más nyelvekben

- Ennek a megközelítésnek a fő hátránya: hiába tudja a programozó, hogy melyik szál akarja lekötni a változót, és melyik amelyik csak megvizsgálni, a predikátum-logikában ez nem leírható (az egyesítés, az "=" mindkét irányban működik). Metalogikai konstrukciókra van szükség, például a `var/1` predikátum, vagy block deklaráció használatára.
- Másik hátrány: a közös változókat is vissza kell léptetni az egyes részek visszalépései esetén. Ez vagy bonyolult, nehezen érthető szemantikához vezet, vagy a hatékony implementációt gátolja. Fél-megoldás: "don't care" vagy "committed choice" nondeterminism: a potenciális VAGY ágak közül az első olyan hajtódik végre, amelyiknek a fejillesztése (és esetleg egy őr (guard) kifejezése) sikerül.

Kommunikáló párhuzamos célok más nyelvekben

- A közös változókkal való munkának további hátránya, hogy a kommunikációs csatornák "állapotát" (a rajta átment összes üzetetet) nyilvántartó struktúra folyamatosan növekszik: csak helyes használat esetén tudja a szemétgyűjtő kidobni a "régi állapotokat" illetve a kezelésük lassúvá válik, ha mindig le kell mászni a mélyére az aktuális állapot eléréséhez. További nehézséget jelent, ha egy csatornába többen is ír(hat)nak.
- Ezt megoldja a kommunikációs változó destruktív felülírása.
- Mind a nemdeterminizmus szőnyeg alá söprése, mind a meta-logikai elemek használata és a nemdeklaratív felülírás nehezíti a program deklaratív szemantika szerinti vizsgálatát, így a programozást.

Explicit Mercury szálak

- A Mercury nyelv tervezői a kommunikációs eszközök destruktív változtatását választották, azonban a csatorna korábbi állapotát az absztrakt "világállapot-változó" tárolja (hasonlóan az I/O-hoz és minden más nemdeklaratív művelethez).
- Egy Mercury-szál ezért:

```
:-type thread == pred(io__state,io__state).
```

- Szál létrehozása:

```
:-pred spawn(pred(io__state,io__state),io__state,io__state).
:-mode spawn(pred(di,uo) is cc_multi,di,uo) is cc_multi.
:-mode spawn(pred(di,uo) is   det   ,di,uo) is cc_multi.
```

Explicit Mercury szálak - példa

```
:- pred main(io__state,io__state).
:- mode main(di,uo) is cc_multi.
main(IO0,IO):-
   spawn(hello,IO0,IO1),
   world(IO1,IO).

:- pred hello(io__state,io__state).
:- mode hello(di,uo) is det.
hello(IO0,IO):-print("hello ",IO0,IO).

:- pred world(io__state,io__state).
:- mode world(di,uo) is det.
world(IO0,IO):-print("world ",IO0,IO).
```

Explicit Mercury szálak - példa

- Eredménye: “hello world ” vagy “world hello ”
  - feltéve, hogy a print predikátum működése atomi
- erre az ismeretlen kimenetelre utal a spawn predikátum `cc_multi` típusa, ami egyben a hívó predikátum típusára is rákényszeríti a `cc_multi` típust
- A `spawn` párhuzamos szál elágazására való. A szál befejezésére várakozás automatikus: a hívó predikátum akkor fut le, ha a belőle indított összes `spawn` és a predikátumban található többi cél is lefutott.

Kommunikáció

- Mivel a szál végezhet I/O-t, a szálak kommunikálhatnak fileokon, pipeokon, socketeken és még ezer más módon. De ez nem hatékony.
- Ezért a Mercuryban lehetőség van az `io_state`-ben adatot tárolni a Prolog blackboard konstrukciójához hasonlóan.
- Ezekhez a globálisan elérhető értékekhez egy leképezés segítségével lehet hozzáférni a
  `set_global(Key, Data, io__state, io__state)`
  és
  `get_global(Key, Data, io__state, io__state)`
  predikátumok használatával.

Kommunikáció

- A Key és Data típusosztályok, amiknek új példányait (típusok) a programozó definiálhat (ebben az előadásban ennél bővebben erre nem térek ki). A típusosztályok használatával elkerülhető, hogy különböző modulok, különböző programozók által írt programrészek ugyanolyan típusú kulcsot használva összekeverjék egymás adatait.
- Mivel összetett struktúrák változtatása több szálból lehetséges, ilyen esetekben szinkronizáció szükséges, amihez a Mercury szemaforokat és mutexeket biztosít (lásd később).

Kommunikáció

- Másik lehetőség globális adatokon keresztüli adatcserére: mutatók.
- `pointer__new(T, pointer(T), io__state, io__state)`, `pointer__get(pointer(T), T, io__state, io__state)`, `pointer__set(pointer(T), T, io__state, io__state)`
- Egy mutató természetesen másolható, így több szál egy mutatón keresztül közös adatokat érhet el.
- A mutatókat a Mercury szemétgyűjtője is ismeri, így a megszűnő mutatók által mutatott adatterületeket felszabadítja.

Szinkronizáció

- Szemaforok:
  `new_sem(int,sem, io__state, io__state),`
  `sem_wait(sem, io__state, io__state),`
  `sem_signal(sem, io__state, io__state)`
- Mutexek:
  `new_mutex(mutex, io__state, io__state),`
  `lock_mutex(mutex, io__state, io__state),`
  `unlock_mutex(mutex, io__state, io__state)`
- Használatuk a más nyelvekben szokásos szemaforokhoz és mutexekhez hasonló.

Szemantika

- Az IO modulba tartozó predikátumokat úgy kell kezelni, hogy a világ állapotát módosítják, a régi állapotot megszüntetik és egy újat hoznak létre, majd kilépésük után további hatásuk nincs. Explicit párhuzamos programok esetén ez nem teljesül. Nézzük a korábbi `main` predikátumot:

```
main(IO0,IO):-
   spawn(hello,IO0,IO1),
   world(IO1,IO).
```

A lefutása kétféle eredménnyel járhat: "hello world " vagy "world hello ". Ez előbbi megfelel a szemantikának, az utóbbi nem.

Szemantika

- Ez kis szépséghiba, és a párhuzamos kiegészítést dokumentáló disszertációban nem próbálják meg feloldani ezt az ellentmondást.
- Viszont a `spawn` predikátumot hívó predikátumok nem lehetnek `det`, csak `cc_multi` determinizmusúak, ami jól tükrözi a tényt, hogy a predikátum többféle lehetséges lefutása közül az egyik (véletlenszerűen választott) fog bekövetkezni, és, hogy számunkra a többféle lefutás ekvivalens.

Implementáció

- A párhuzamos kiegészítés a közös memóriát használó (shared memory) többprocesszoros számítógépeinek lehetőségeit használja ki.
- A dokumentációban relatíve kevés mérési eredmény található. A mérések tanúsága szerint az elérhető gyorsulás közel sem lineáris. Egy nagyon jól párhuzamosítható algoritmus (Monte Carlo integrálás) az egy processzoron történő futtatáshoz képest két processzoron 1.3-szoros, három processzoron 1.7-szeres gyorsulást mutatott. Ennek a legfőbb oka a szemétgyűjtő alrendszerben keresendő: szemétgyűjtés közben csak az egyik végrehajtó gép dolgozik. Remélik, hogy az új, jövőbeli szemétgyűjtővel nagyobb hatékonyságot érhetnek el.

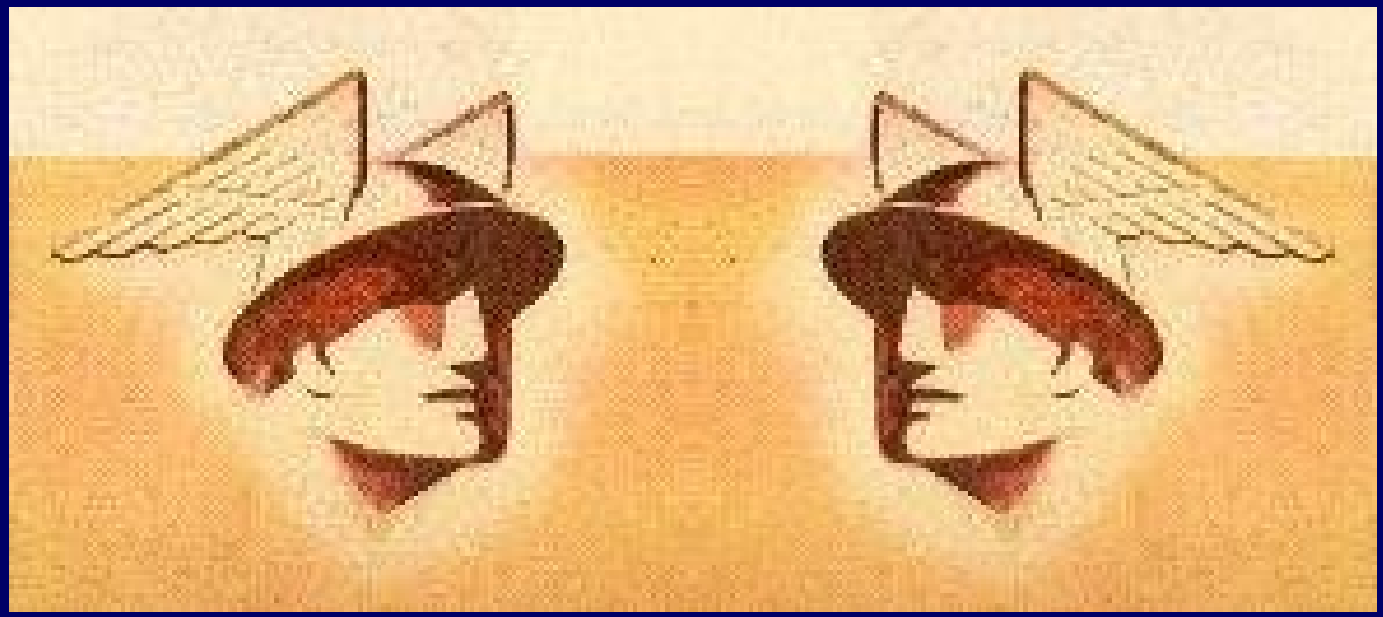
További információ

- Az implementáció részletezése túlmutat ezen az előadáson. A Mercury rendszer és a hozzá tartozó dokumentációk megtalálhatóak a következő helyen:

  The Mercury Project homepage
  http://www.cs.mu.oz.au/research/mercury/

- Az előadás főként a következő mű alapján készült:

  Thomas Conway: *Towards parallel Mercury*
  Ph.D. thesis, Department of Computer Science and Software Engineering, The University of Melbourne
  http://www.cs.mu.oz.au/research/mercury/information/papers.html#conway-thesis



Kérdések?