A Hume nyelv

Bevezető

Patai Gergely

2006. november 6.

A beágyazott rendszerek követelményei

- megbízhatóság
- robusztusság
- rövid válaszidők
- kevés erőforrás
- párhuzamosság
- változatos perifériák és platformok

A funkcionális nyelvek gyengeségei

- komplex állapot nehézkes leírhatósága
- időviszonyok figyelmen kívül hagyása
- nehézkes kommunikáció a programrészek között
- korlátozott I/O
- hatékonysági problémák
- elfogadottság

Tervezési megfontolások

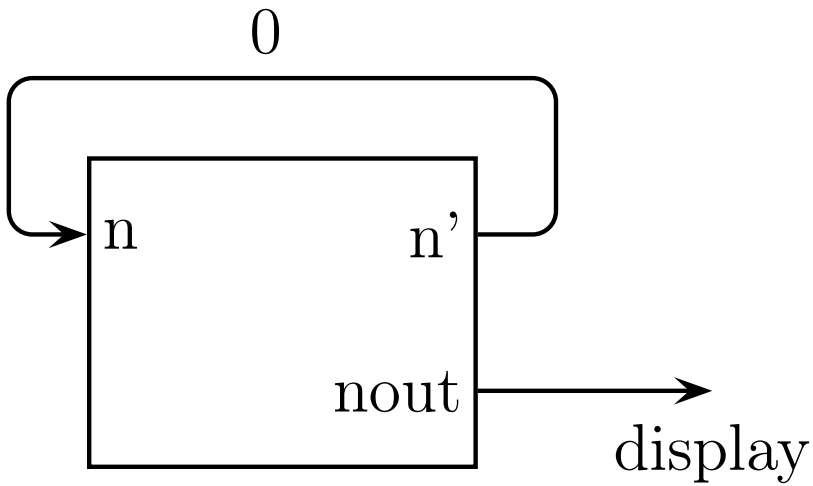
- megbízhatóság → jósolható és bizonyítható tulajdonságok
- örök dilemma: kifejezőkészség (Turing-gép) vs. eldönthetőség (FSM)
- a Hume-mal szemben támasztott követelmények:
  - erős garanciák tár- és időhasználatra magas kifejezőkészség mellett
  - csak (Neumann-architektúrán) hatékonyan megvalósítható magas szintű nyelvi elemek
  - jól skálázható bizonyítások
- másik dilemma: hatékonysági garanciák alacsony szinten, helyességi bizonyítások magas szinten adhatók...
- megoldás: funkciók („felelősségek”) szétválasztása

A három nyelvi réteg

- típusok és perifériák (*declaration language*):
  - adatok és alapműveletek
  - ki- és bemeneti eszközök
  - kivételek deklarálása
- koordináció (*coordination language*):
  - idő- és tárbeli viselkedés
  - párhuzamosság
  - sorrendezés
  - kivételek elkapása
- kifejezések (*expression language*):
  - absztrakció
  - állapotmentes
  - alrétegekre bontható
  - kivételek indítása

Példa: számláló

```
program

stream display to "std_out";

type integer = int 32;

inc x = x+1;

box counter
  in (n::integer)
  out (n'::integer, nout::integer)
match
  n -> (inc n, n);

wire counter
  (counter.n' initially 0)
  (counter.n, display);
```



Deklarációs nyelv

- összefogja a programot és keretet ad a többi funkciónak
- minden deklarációt pontosvessző zár le
- típusok: megkülönböztetett unió (`data`) és típusszinonima (`type`)
- értékek: függvények (Haskell-stílusú szintaktika) és konstansok (`constant`); típusmegadás lehetséges (`::`)
- dobozok és vezetékek: `box` és `wire` blokkok
- perifériák: `stream`, `port`, `interrupt`, `memory`, `fifo`
- kivételek: `exception`

Típusok

- alaptípusok:
  - n-bites számok: `word` n (előjel nélküli), `int` n (előjeles), `float` n (IEEE lebegőpontos), `fixed` n (fixpontos)
  - fix méretű számok: `bit` ($\equiv$ `word 1`), `byte` ($\equiv$ `word 8`)
  - logikai: `bool`
  - karakter: `char` (Latin-1), `unicode`
- összetett típusok:
  - vektor: fix méret, azonos típusú elemek, pl. `<<1, 2, 3>>`
  - ennes: fix méret, különböző típusú elemek, pl. `(5, true)`
  - lista: változó méret, azonos típusú elemek, pl. `[1, 2, 3]`
  - string: változó vagy fix méretű karaktersorozat
  - megkülönböztetett unió: adatkonstruktor
  - idő

Műveletek összetett típusokon

- közös műveletek (az idő kivételével):
  - `length` e: elemszám
  - e `@` i: i. elem (1-től számozva)
  - e_1 `++` e_2: konkatenáció (ennesekre nincs)
  - `==`, `!=`, `<=`, `<`, `>`, `>=`: lexikografikus rendezés
- lista:
  - `hd` l: l feje
  - `tl` l: l farka
- vektor:
  - `vecdef` n f: n hosszú vektor az $f(i)$ $i = 1 \ldots n$ értékekből
  - `vecmap` f v: f alkalmazása v elemeire
  - `update` v i e: v-ben az i. elem kicserélése e-re

Kifejezések nyelve

- értékkel rendelkező kifejezések leírása
- értékek: változó, egyszerű érték (literál), összetett érték (vektor, lista, ennes, string)
- függvényalkalmazás: $f\ e_1\ \ldots\ e_n$ (f függvény vagy adatkonstruktor, e_i nem lehet operátor) vagy $e_1\ op\ e_2$; nincs részleges kiértékelés
- esetek: `case` e `of` p_1 `->` e_1 `|` $\ldots$ `|` p_n `->` e_n
- feltételek: `if` c `then` e_t `else` e_f
- lokális definíciók: `let` $d_1\ \ldots\ d_n$ `in` e
- kivételek: `raise` $id\ e$ (egy `exception` id-ként deklarált kivétel dobása e tartalommal)
- típusmegadás: e `::` t (információvesztés nélküli nézet), e `as` t (számítást igénylő konverzió)

Koordinációs nyelv

- alapvető programszervezési elv:
  - dobozok: pufferelt kombinációs hálózatok
  - vezetékek: adatot tároló összekötések
- minden be- és kimenetet be kell kötni
- a dobozok bemenetei és kimenetei közötti kapcsolatot tiszta függvény írja le, amely ennesről ennesre képez
- a dobozokat valamilyen stratégiával ütemezzük
- egy doboz futása:
  1. mintaillesztés a bemenetekre
  2. siker esetén az adatok elvétele a vezetékekről
  3. a művelet elvégzése (függvényalkalmazás)
  4. az eredmény kiírása, ha lehet; ha nem, akkor blokkolódás és várakozás a használni kívánt kimenő vezetékek kiürülésére
- mindegyik lépést egyszerre hajtják végre a dobozok

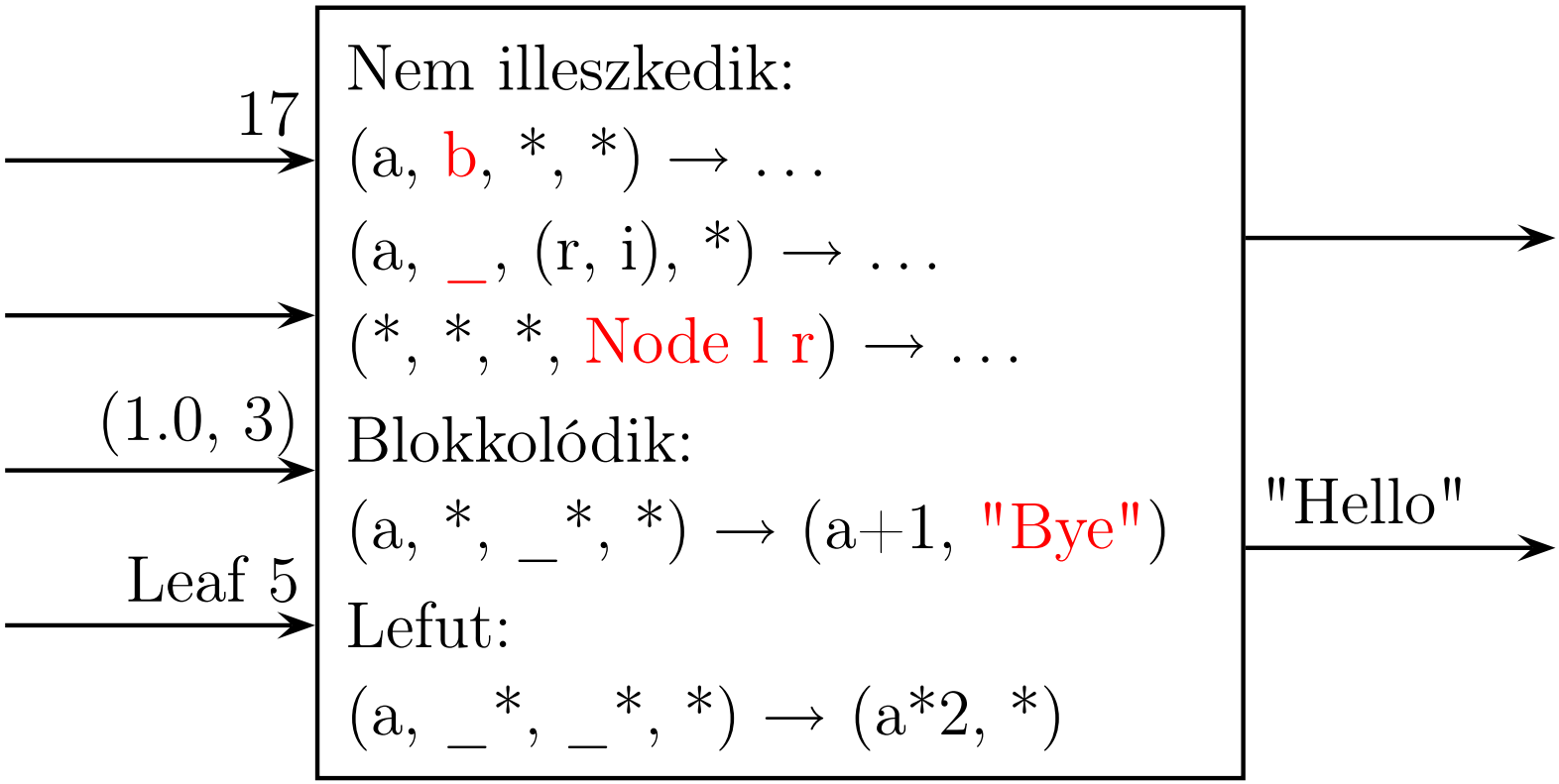
Építőelemek deklarálása

- doboz (k bemenet, l kimenet, m szabály):
  `box` *doboznév*
  `in` $(ni_1$ `::` ti_1`,` $\ldots$ `,` ni_k `::` $ti_k)$
  `out` $(no_1$ `::` to_1`,` $\ldots$ `,` no_l `::` $to_l)$
  `match` vagy `fair`
  $rule_1$ `|` $\ldots$ `|` $rule_2$`;`
- bekötés:
  `wire` *doboznév*
  $(pi_1$ `[initially` vi_1`]` `[trace]`, $\ldots$ `)`
  $(po_1$ `[initially` vo_1`]` `[trace]`, $\ldots$ `);`
- szabály: $tuple_i$ `->` $tuple_o$ (a kimenet olyan kifejezés, ami l-esre értékelődik ki, lehet pl. egy nagy feltételes szerkezet is)
- bemeneti minták: `*` (közömbös; kimeneten a semmi jele), `_` (elvesz és eldob), `_*` (elvesz és eldob, ha van mit)

Példák: paritás, multiplexer

```
box parity
  in (pin::bit, s::bit)
  out (pout::bit, s'::bit)
match
  (0, n) -> (n,   n) |
  (1, n) -> (1-n, 1-n);

wire parity
  ( ... , parity.s' initially 0)
  ( ... , parity.s);

box merge
  in (min1::bit, min2::bit)
  out (mout::bit)
fair
  (x, *) -> x |
  (*, x) -> x;
```

Példák a szabályok tüzelésére



Superstep: az összes doboz egyszeri futása

1. aki éppen blokkolt, az kihagyja a következő három lépést
2. mintaillesztés: minden doboz, amelyben egyik szabály bal oldala sem volt illeszthető, felfüggeszti a tevékenységét a superstep végéig, míg a többiek kiválasztják az első sikeresen illesztett mintát
3. fogyasztás: mindenki elveszi az összes olyan bemenő vezetékéről az adatot, amelyhez nem * minta tartozik, és beállítják a mintában szereplő lokális változókat
4. futás: mindenki lefuttatja a kiválasztott szabályát, és puffereli az eredményt
5. kiírás: mindenki megpróbálja kiírni az eredményt a kimenő vezetékeire; ha egy nem * („void”) kifejezéshez tartozó vezetéken van adat, az adott doboz blokkolódik, és a következő ciklusban újra próbálkozik a kiszámolt érték kiírásával

Állapot megőrzése

- a dobozok nem őrzik meg a belső állapotukat két futás között
- a program állapotát a vezetékek tárolják
- belső állapottal rendelkező komponens szimulálása: közvetlen visszacsatolás
- nem jelent problémát, mert a vezetékek írása és olvasása szigorúan el van választva időben
- sablon:

```
box foo
  in (xin::TI, state::S)
  out (xout::TO, state'::S)
match
  ...
  (..., old_state) -> (..., new_state);

wire foo
  (..., foo.state' initially FOO_START)
  (..., foo.state);
```

Doboziteráció

```
f :: TA -> bool;
g :: TA -> TR;
h :: TA -> TA;
i :: TI -> TA;

fiter a =
  let iter x = if f x then g x else iter (h x)
  in iter (i a);

box biter
  in (a::TI, s::TA)
  out (res::TR, s'::TA)
match
  (*, s) -> if f s then (g s, *)
                   else (*, h s) |
  (a, *) -> (*, i a);

wire biter (..., biter.s') (..., biter.s);
```

Doboziteráció – lista hossza

```
type TA = (integer, [integer]);
type TR = integer;
type TI = [integer];

f :: TA -> bool;
f (_, xs) = xs == [];

g :: TA -> TR;
g (l, _) = l;

h :: TA -> TA;
h (l, x:xs) = (l+1, xs);

i :: TI -> TA
i l = (0, l);
```

A kifejezésnyelv rétegei

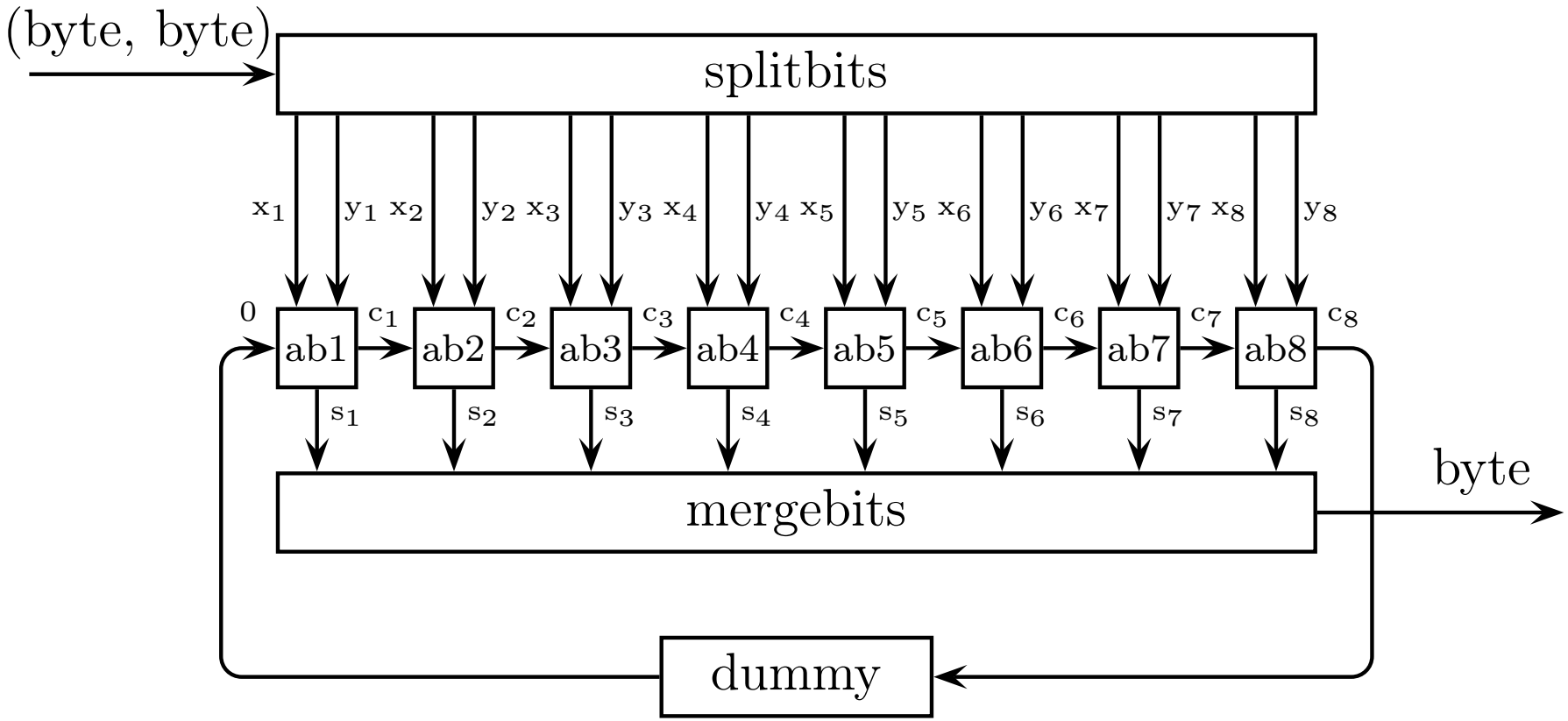
1. HW-Hume: nincsenek függvények, csak egyszerű adatok vannak (bitek és ennesek)
2. FSM-Hume: nemrekurzív adatszerkezetek, feltételes kifejezések
3. HO-Hume: nemrekurzív elsőrendű függvények, jól definiált viselkedésű magasabbrendű függvények (map, fold)
4. PR-Hume: egyszerű rekurzió
5. Full Hume: teljes rekurzió

Garanciák a nyelv függvényében

LHS	RHS	ekviv.	leállás	tár	idő
véges	műveletek	I	I	pontos	pontos
véges	feltételek	I	I	korlátos	korlátos
verem*	műveletek	I	I	–	pontos
véges*	műveletek	N	I	pontos	pontos
véges	prim. rek.	N	I	korlátos	korlátos
végtelen	ált. rek.	N	N	–	–

- LHS: a vezetékeken lévő adatok típusa
- RHS: a dobozokban megengedett kifejezések
- csillag: a vezetékek figyelmen kívül hagyhatók
- I/N: eldönthető/nem eldönthető

Bináris összeadó



Bitösszeadó dobozok sablonokkal

```
template adder
  in (i1::bit, i2::bit, ic::bit)
  out (os::bit, oc::bit)
match
  (0, 0, c) -> (c, 0) |
  (1, 1, c) -> (c, 1) |
  (_, _, 0) -> (1, 0) |
  (_, _, 1) -> (0, 1);

instantiate adder as addbit * 8;

for i = 2 to 7 wire addbit{i}
  (splitbits.x{i}, splitbits.y{i}, addbit{i-1}.oc)
  (mergebits.x{i}, addbit{i+1}.ic);

wire addbit1
  (splitbits.x1, splitbits.y1, dummy.oc)
  (mergebits.x1, addbit2.ic);

wire addbit8
  (splitbits.x8, splitbits.y8, addbit7.oc)
  (mergebits.x8, dummy.ic);
```

dummy átnevezése addbit0-ra

```
template adder
  in (i1::bit, i2::bit, ic::bit)
  out (os::bit, oc::bit)
match
  (0, 0, c) -> (c, 0) |
  (1, 1, c) -> (c, 1) |
  (_, _, 0) -> (1, 0) |
  (_, _, 1) -> (0, 1);

instantiate adder as addbit * 8;

for i = 1 to 8 wire addbit{i}
  (splitbits.x{i}, splitbits.y{i}, addbit{i-1}.oc)
  (mergebits.x{i}, addbit{(i+1)%9}.ic);

box addbit0
  in (ic::bit)
  out (oc::bit)
match
  _* -> 0;

wire addbit0 (addbit8.oc) (addbit1.ic);
```

A jövőben várható

- hierarchikus dobozok: a lapos struktúra nem skálázódik jól a készítők szerint, így nincs értelme olyan definíciót adni a befoglaló dobozok viselkedésére, amely megegyezne a kilapított programéval
- tár- és időkorlátok megadása függvények és dobozok szintjén: a nyelv leírásában már most is szerepelnek `within`-deklarációk, ezek azonban egyáltalán nem működnek
- reaktív rendszerek programozásának támogatása: a jelenlegi ütemezés mellett nem lehetséges pl. a megszakítások kellően gyors kiszolgálása Hume szinten
- HW-Hume alkalmazása hardverfejlesztésben (FPGA)
- párhuzamosítás: az írási és olvasási szakaszok időbeni elválasztása miatt a dobozok gond nélkül futtathatók egyszerre
- dinamikus, futásidőben változtatható koordinációs szint?