KTH Information and Communication Technology

ID1019
Johan Montelius

Programmering II (ID1019) 2018-06-07 08:00-12:00

Namn: ______________________________

Instruktioner

- Svaren skall lämnas på dessa sidor, använd det utrymme som finns under varje uppgift för att skriva ner ditt svar.
- Svar skall skrivas på svenska eller engelska.
- Du skall lämna in hela denna tentamen.
- Inga ytterligare sidor skall lämnas in.

Betyg

Tentamen har ett antal uppgifter där några är lite svårare än andra. De svårare uppgifterna är markerade med en stjärna, *[p*]*, och ger poäng för de högre betygen. Vi delar alltså upp tentamen in grundpoäng och högre poäng. Se först och främst till att klara de normala poängen innan du ger dig i kast med de högre poängen.

- E: 12 grundpoäng
- D: 15 grundpoäng
- C: 18 grundpoäng
- B: 20 grundpoäng och 8 högre poäng
- A: 20 grundpoäng och 10 högre poäng

De som skriver 4.5hp versionen så skall bara svara på frågorna 1-6. Gränsen för E, D och C är då 8, 10 och 12 poäng. Gränserna för B och A är 3 respektive 5 poäng.

Gränserna kan komma att justeras nedåt men inte uppåt.

Namn:________________________________ Persnr:________________________________

1 Lambdakalkyl [2p]

Evaluera följande lambdauttryck:

- $(\lambda x \to x + x)4$ **Svar:** 8
- $(\lambda x \to (\lambda y \to y + 2 * x)2)4$ **Svar:** 10
- $(\lambda x \to (\lambda x \to x + x)(x + 2))4$ **Svar:** 12

2 Operationell semantik [2p]

Givet de regler för en operationell semantik som finns i appendix, visa steg för steg vilka regler som används och evaluera följande uttryck:

Svar:

1.

$$\frac{\texttt{:a} \equiv a}{E\{y/b\}(\texttt{:a}) \to a}$$

2.

$$\frac{y/b \in \{y/b\}}{E\{y/b\}(\texttt{y}) \to b}$$

3.

$$\frac{\textit{from 1} \qquad \textit{from 2}}{E\{y/b\}(\texttt{\{:a, y\}}) \to \{a, b\}}$$

4.

$$\frac{E\{y/a, x/b\}(\texttt{y}) \to a \qquad E\{y/a, x/b\}(\texttt{x}) \to b}{E\{y/a, x/b\}(\texttt{\{y, x\}}) \to \{a, b\}}$$

5.

$$\frac{\textit{from 3} \qquad S(\{y/b\}, \texttt{\{y, x\}}) \to \{\} \qquad P\{\}(\texttt{\{y, x\}}, \{a, b\}) \to \{y/a, x/b\} \qquad \textit{from 4}}{E\{y/b\}(\texttt{ \{y, x\} = \{:a, y\}; \{y,x\}}) \to \{a, b\}}$$

6.

$$\frac{\texttt{:b} \equiv b}{E\{\}(\texttt{:b}) \to b}$$

$$\frac{\text{from 6} \qquad S(\{\}, \texttt{y}) \to \{\} \qquad P\{\}(\texttt{y}, b) \to \{y/b\} \qquad \text{from 5}}{E\{\}(\texttt{ y = :b; \{y, x\} = \{:a, y\}; \{y,x\}}) \to \{a, b\}}$$

Namn:______________________________ Persnr:______________________________

3 Mönstermatchning [2 p]

Givet nedanstående uttryck, vad blir den resulterande omgivningen i de fall där möstermatchninging lyckas? (om du använder Erlang så skall vänsterledet vara `[X, Y | Z]`)

a: [x, y | z] = [1, 2, 3] **Svar:** x = 1, y = 2, z = [3]

b: [x, y | z] = [1, [2, 3]] **Svar:** x = 1, y = [2,3] z = []

c: [x, y | z] = [1 | [2, 3]] **Svar:** x = 1, y = 2, z = [3]

d: [x, y | z] = [1 | [2, 3 | [4]]] **Svar:** x = 1, y = 2, z = [3,4]

e: [x, y | z] = [1, 2, 3, 4] **Svar:** x = 1, y = 2, z = [3, 4]

Namn:________________________________ Persnr:________________________________

4 Rekursion

Fizz-Buzz [2 p]

Vi skall implementera en funktion `fizzbuzz/1` som givet ett tal $n \geq 0$ returnerar en lista av de n första elementen i fizzbuzz-serien. Fizz-Buzz går till så att man räknar från 1 till n men byter ut all tal som är delbara med 3 mot `:fizz`, alla tal som är delbara med 5 med `:buzz` och all tal som är delbara med både 3 och 5 mot `:fizzbuzz`. De första fem elementen är således: `[1,2,:fizz,4,:buzz]`.

Du skall implementera den rekursiva funktionen `fizzbuzz/4` som hjälper oss att göra detta. Första argumentet är nästa element i listan, andra för att veta att vi är klara och tredje o fjärde håller reda på om talet är delbart med 3 eller 5. Du får bara använda addition, inte någon annan aritmetisk operation. Du skall inte göra funktionen svansrekursiv.

Om du tycker att detta är enkelt så prova att göra det i kväll efter tre öl!

```
def fizzbuzz(n) do fizzbuzz(1, n+1, 1, 1) end
```

Svar:

```
def fizzbuzz(n, n, _, _) do [] end
def fizzbuzz(i, n, 3, 5) do [:fizzbuzz| fizzbuzz(i+1, n,   1,   1)] end
def fizzbuzz(i, n, 3, b) do [:fizz    | fizzbuzz(i+1, n,   1, b+1)] end
def fizzbuzz(i, n, f, 5) do [:buzz    | fizzbuzz(i+1, n, f+1,   1)] end
def fizzbuzz(i, n, f, b) do [i        | fizzbuzz(i+1, n, f+1, b+1)] end
```

Namn:________________________________ Persnr:________________________________

hyggligt balanserat [2 p*]

Definiera en funktion `fairly/1` som returnera antingen `{:ok, depth}` om ett träd är hyggligt balanserat och `depth` är djupet på trädet eller `:no`. Ett träd anses var hyggligt balanserat om dess två grenar är hyggligt balanserade och skillnaden på djupen är som högst ett. Du bestämmer hur ett trädet skall representeras och du kan förutom `max/2` använda dig av en funktion `abs/1` som returnerar absolutbeloppet av ett tal.

Svar:

```
@type tree() :: nil | {:node, any(), tree(), tree()}

def fairly(nil) do {:ok, 0} end
  def fairly({:node, _, left, right}) do
    case fairly(left) do
      {:ok, l} ->
        case fairly(right) do
          {:ok, r} ->
            if abs(r-l) < 2 do
              {:ok, 1 + max(l,r)}
            else
              :no
            end
          :no -> :no
        end
      :no -> :no
    end
  end
```

5 Tidskomplexitet

sortera en lista [2 p]

Om vi har nedanstående implementation av en sorteringsfunktion för en lista, vad är den asymptotiska tidskomplexiteten för funktionen?

```
def sort([]) do [] end
def sort([h|t]) do
  insert(h, sort(t))
end

def insert(e, []) do [e] end
def insert(e, [h|t]=l) do
  if e  < h do
    [e|l]
  else
    [h|insert(e, t)]
  end
end
```

Svar: Tidskomplexiteten är $O(n^2)$ där n är längden på listan.

Namn:______________________________ Persnr:______________________________

en graf [2 p*]

Antag att vi representerar en riktad acyklisk graf och en funktion som söker igenom grafen enligt nedan. Vad är tidskomplexiteten för att avgöra om ett element finns i grafen? Antag att noder i grafen har upp till k länkar.

```
@type graph :: {:node, any(), [graph()]} |  nil

def search(_, nil) do :fail end
def search(e, {:node, e, _}) do :found end
def search(e, {:node, _, paths}) do
  List.foldl(paths,
    :fail,
    fn(p,a) ->
      case a do
        :found -> :found
        :fail ->
          search(e, p)
      end
    end)
end
```

Svar: Tidskomplexiteten är $O(k^n)$ där n är antalet noder i grafen och k förgreningsfaktorn.

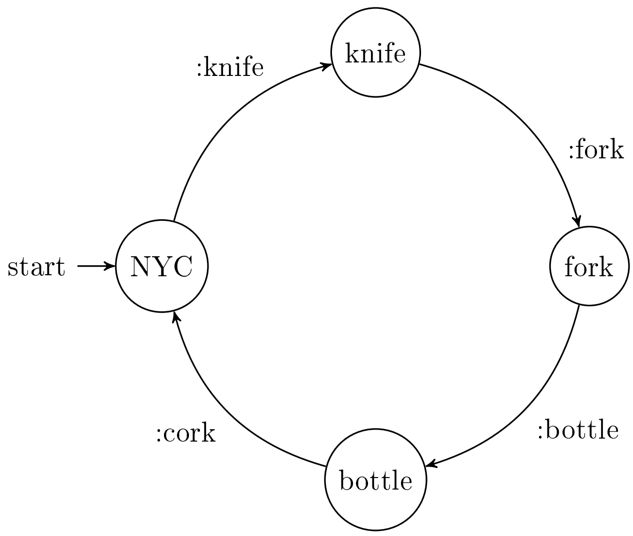
Namn:________________________________ Persnr:________________________________

6 Processbeskriving

A knife, a fork ... [2 p]

Nedan ser du ett enkelt tillståndsdiagram för en process. Varje gång processen går in i tillståndet `nyc` så skall den skriva “Hey Jim!” på stdout (genom att anropa `IO.puts(“Hey Jim!”)`). Meddelanden som inte är utritade skall ligga kvar i meddelandekön tills de kan hanteras.

Implementera processen och en funktion `dillinger/0` som startar en sådan process.



Svar:

```
def dillinger() do
  spawn(fn() -> nyc() end)
end
```

```
def nyc() do
 IO.puts("Hey Jim!")
 receive do
   :knife -> knife()
 end
end
```

```
def knife() do
 receive do
   :fork -> fork()
 end
end
```

```
def fork() do
 receive do
   :bottle -> bottle()
 end
end
```

```
def bottle() do
 receive do
   :cork -> nyc()
 end
end
```

Namn:______________________________ Persnr:______________________________

tic-tac-toe [2 p]

Antag att vi har följande definition av procedurerna `first/1`, `second/1` och `third/1`.

```
def first(p) do
  receive do
     :tic ->
       second(p, [:tic])
     :tac ->
       second(p, [:tac])
  end
end

def second(p,all) do
  receive do
    :tic -> third(p, [:tic|all])
    :toe -> third(p, [:toe|all])
  end
end

def third(p, all) do
  receive do
     x ->  send(p, {:ok, [x|all]})
  end
end
```

Vad blir resultatet när vi exekverar anropet `test/0`?

```
def test() do
   self = self()
   p = spawn(fn()-> first(self) end)
   send(p, :tic)
   send(p, :tac)
   send(p, :toe)
   receive do
     {:ok, res} -> res
   end
end
```

Svar: `[:tac, :toe, :tic]`

Namn:________________________________ Persnr:________________________________

parallell foldp/2 [2 p*]

Vi skall göra "fold" på elementen i en lista men istället för att göra fold från höger eller vänster skall vi göra en parallell version. Givet en lista av element skall vi dela listan i två lika stora delar, göra parallell fold på på vardera lista för att sedan applicera vår funktion på resultatet av de två rekursiva anropen. Utmaningen är att de två rekursiva anropen skall ske i olika processer så att vi kan snabba upp exekveringen med hjälp av parallellism.

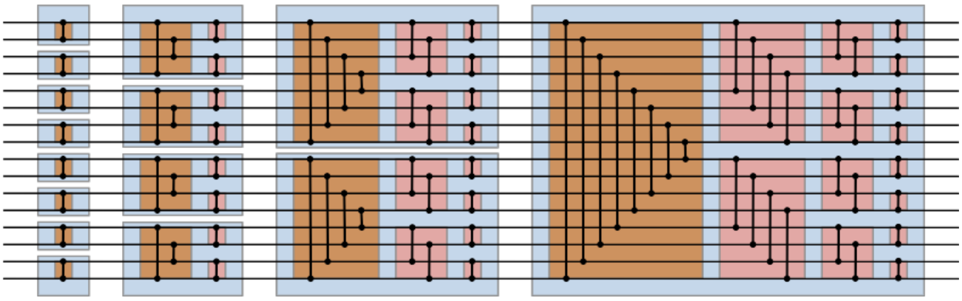
Vi har inget initialt värde för vår operation så vi kommer kräva att listan innehåller minst ett element. Att göra foldp på en lista av ett element returnerar elementet självt.

Svar:

```
def foldp([x], _) do x end
def foldp(list, f) do
  {l1, l2} = split(list, [], [])
  me = self()
  spawn(fn() -> res = foldp(l1, f); send(me, {:res, res}) end)
  spawn(fn() -> res = foldp(l2, f); send(me, {:res, res}) end)
  receive do
    {:res, r1} ->
      receive do
        {:res, r2} ->
          f.(r1,r2)
      end
  end
end

def split([], l1, l2) do {l1, l2} end
def split([h|t], l1, l2) do
  split(t, l2, [h|l1])
end
```

Namn:________________________________ Persnr:________________________________



Figur 1: A bitonic sort network of size 16.

7 Programmering

I den här uppgiften skall vi bygga upp ett nätverk för sortering. Man använder sig av liknande nätverk antingen när man vill parallellisera en sortering eller när man har flera ingående strömmar av meddelanden som skall sorteras till flera utgående strömmar. Den metod som vi skall använda oss av kallas för bitonisk sorterare och följer ett rekursivt mönster.

I fig 1 ser vi en sorterare av storlek 16, den har 16 ingående strömmar till vänster och 16 utgående strömmar till höger. Idén är att 16 tal går in genom nätverket och sorteras så att det minsta talet alltid kommer ut genom den översta strömen och det största genom den nedersta strömmen.

Där två strömmar är förbundna med ett vertikalt streck sker en jämförelse och det mindre av de två värdena skickas vidare på den övre strömmen och det större på den nedre.

Vi ser ett mönster som upprepar sig rekursivt. Ett nätverk för två strömmar är naturligtvis trivialt och kräver enbart en jämförelse. För fyra strömmar så 1/ gör vi först en rekursiv sorteringen av två strömmar, 2/ en operation som i bilden är inom en brun ruta och som vi kommer kalla *cross* följt av 3/ två operationer markerade i rött som vi kommer kalla *merge*.

Operationen merge består egentligen av en rekursiv upprepning av en operation som vi kommer kalla *zipc*. Om vi ser hur nätverket sorterar 8 strömmar så ser vi hur de två merge-operationerna (en för de fyra övre strömmarna och en för de fyra undre) internt består av en zipc-operation som involverar alla fyra strömmar följt av två zipc-operationer på två strömmar var.

När vi implementerar dessa nätverk så kommer jämförelserna vara processer och nätverket definieras av de vägar som processerna skickar sina resultat. En jämförelse process kommer vänta på två inkommande meddelande, jämföra dessa och sen skicka de två värdena vidare i grafen

Namn:________________________________ Persnr:________________________________

en jämförande process [2 p]

Det första vi skall göra är att definiera en process som tar emot två meddelanden, jämför dess värden och skickar det mindre vidare till en process och det större till en annan. Vi måste hålla kontroll på att talen vi jämför verkligen tillhör samma grupp av tal så vi kommer hålla koll på vilken *epok* vi skall hantera och accepterar bara två tal från nästa epok.

Meddelanden till processen kommer ha följande struktur:

- `{:epoch, n, x}` : `x` är ett tal av två i epok `n`
- `{:done, n}` : processen skall terminera

När processen startas så förväntar den sig två tal i epok 0, två tal i epok 1 etc. Den skickar talen på samma form och med samma epoknummer. Ett "done-meddelande" skickas även det vidare till de två processerna så att hela nätverket terminerar.

Implementera en funktion `comp/2` som tar två argument, identifierarna till de två processerna dit de sorterade talen skall skickas, startar en jämförande process och returnerar dess processidentifierare. Processen skall förvänta sig meddelande med start från epok 0.

Svar:

```
def comp(low, high) do spawn(fn() -> comp(0, low, high) end)  end

def comp(n, low, high) do
  receive do
    {:done, ^n} ->
      send(low, {:done, n})
      send(high, {:done, n})
    {:epoc, ^n, x1} ->
      receive do
        {:epoc, ^n, x2} ->
          if x1 < x2 do
            send(low,  {:epoc, n, x1})
            send(high, {:epoc, n, x2})
          else
            send(low, {:epoc, n, x2})
            send(high,{:epoc, n, x1})
          end
          comp(n+1, low, high)
      end
  end
end
```

Namn:________________________________ Persnr:________________________________

sorter/1 [2 p]

Antag att vi har en funktion `setup/1` som tar en lista av processidentifierare och startar upp all processer som behövs i ett sorterande nätverk av storlek n. Listan av processidentifierare är de processer, i ordning, som nätverket skall leverera de sorterade talen till. Funktionen returnerar en lista av processidentifierare som varje epok skall skickas till.

Om vi bygger ett sorterande nätverk av storlek 4 så kommer processen starta sex stycken interna processer och returnera en lista `[i1,i1,i2,i2]` där `i1` och `i2` är identifierarna till de två första processerna i nätverket. Du skall inte implementera `setup/1` nu utan vi antar att den finns.

Implementera en funktion `sorter/1` som tar en lista av processidentiferare, de processer som vi skall skicka de sorterade epokerna till. Funktionen skall starta en process som tar emot två meddelande, ett meddelande som är en begäran att utföra en sortering och ett för att avsluta nätverket.

- `{:sort, epoch}` : Där `epoch` är en lista av tal som skall sorteras. Vi antar att det finns lika mång tal i listan som vi har ett sorterande nätverk för i.e. talen skickas till de olika ingångarna (ordningen spelar ingen roll). Glöm inte bort att sätta epoknummret och räkna upp en räknare så att nästa begäran om sortering får ett nytt nummer.
- `:done` : avsluta nätverket, skickas till alla ingångar.

Till din hjälp finns dessa funktioner:

- `each(list, fun)` Applicerar funktionen `fun` på varje element i listan.
- `zip(list1, list2)` Returnerar en lista av tupler som består av elementen från de två listorna. Ett anrop till `zip([1,2],[:a,:b])` skulle returnera `[{1,:a},{2,::b}]`.

Använd nästa sida för svaret.

Svar:

```
def start(sinks) do
  spawn(fn() -> init(sinks) end)
end

def init(sinks) do
  netw = setup(sinks)
  sorter(0, netw)
end

def sorter(n, netw) do
  receive do
    {:sort, this} ->
      each(zip(netw, this),
           fn({cmp, x}) -> send(cmp,{:epoc, n, x}) end)
      sorter(n+1, netw)
    :done ->
      each(netw, fn(cmp) -> send(cmp, {:done, n}) end)
  end
end
```

setup för $n = 2$ [2 p]

Dags för den svåra biten men vi börjar enkelt. Du skall nu implementera `setup/2` som tar två argument, ett tal n och en lista av n processidentifierare. Processidentifierarna är nätverkets utgångar och funktionen skall sätta upp nödvändiga processer och returnera en lista av nätverkets ingångar. Funktionen använder vi sedan för att implementera `setup/1` som är given nedan. Vi antar fortsättningsvis att n är en jämn multipel av 2 dvs $2, 4, 8, ...$.

```
  def setup(sinks) do
    n = length(sinks)
    setup(n, sinks)
 end
```

Eftersom vi skulle börja enkelt så skall du i denna uppgift enbart hantera fallet då n är lika med 2 (vilket betyder att listan bara har två processidentifierare). Använd med fördel en funktion som du skapat i den första uppgiften för att starta en jämförelseprocess.

Svar:

```
def setup(2, [s1, s2]) do
  cmp = comp(s1, s2)
  [cmp, cmp]
end
```

Namn:____________________ Persnr:____________________

merge för $n = 2$ [2 p]

Du har nu allt som behövs för att starta ett nätverk av storlek 2 men det är ju i det närmaste löjligt lite så vi skall utvidga det storlek 4. Vi börjar med en funktion som skall göra det vi kallade för *merge* dvs de röda delarna i Fig reffig:bitonic. Funktion `merge/2` skall ta ett tal n och en lista av n processidentifierare. Funktionen skall returnera en lista av processindentifierare som är ingångarna till en nätverk som gör mergeoperationen.

Vi börjar enkelt, du skall implementera `merge/2` så att den fungerar för $n = 2$.

Svar:

```
def merge(2, [s1,s2]) do
  cmp1 = comp(s1,s2)
  [cmp1, cmp1]
end
```

Namn:________________________________ Persnr:________________________________

cross/2 [2 p]

Nu till en något knepigare operation, den vi kallar *cross* dvs den bruna operationen i Fig 1. Du skall implementera en funktion `cross/2` som tar två listor av processidentifierare. Listorna representerar den $n/2$ övre och $n/2$ nedre utgående strömmarna från cross-operationen. Funktionen skall returnera en tuple bestående av två listor, de $n/2$ övre ingångarna och de $n/2$ nedre ingångarna. Funktionen skall fungera för godtyckliga n.

Till din hjälp får du använda funktionen `reverse/1` som vänder på en lista.

Svar:

```
def cross(low, high) do
  cross(low,  reverse(high), [])
end

def cross([], [], crossed) do
  {reverse(crossed), crossed}
end
def cross([l|low], [h|high], crossed) do
  cmp = comp(l, h)
  cross(low, high, [cmp | crossed])
end
```

Namn:______________________________ Persnr:______________________________

setup för $n = 4$ [2*]

Vi antar nu att funktionerna `setup/2`, `merge/2` och `cross/2` fungerar då $n = 2$, du skall nu använda dessa för att utöka `setup/2` så att den även fungerar då $n = 4$.

Svar:

```
def setup(4, [s1,s2,s3,s4]) do
  [m1, m2] = merge(2, [s1,s2])
  [m3, m4] = merge(2, [s3,s4])

  {[c1, c2], [c3, c4]} = cross([m1, m2], [m3, m4])

  [i1, i2] = setup(2, [c1, c2])
  [i3, i4] = setup(2, [c3, c4])

  [i1, i2, i3, i4]
end
```

Namn:________________________________ Persnr:________________________________

merge/2 [2*]

Dags att utöka `merge/2` så att den fungerar för godtyckliga n $(2, 4, 8, ...)$. Funktionen blir nu en rekursiv funktion där man i varje rekursionsteg gör vad vi kallade för en *zipc-operation*. Själva zipc-operationen är den som är röd i Fig 1 så börja med att implementera en funktion `zipc/2` som utför operationen på två listor av längd $n/2$ och returnerar en lista av de ingångar som du behöver.

Du kan använda dig av den aritmetiska funktionen `div/2` som gör heltalsdivision och biblioteksfunktionen `split/2` som tar en lista och ett tal n och returnerar en tuple med två dellistor, de n första elementen och resten.

Svar:

```
def merge(n, sinks) do
  n = div(n,2)
  {sink_low, sink_high} = split(sinks, n)
  merged_low =  merge(n, sink_low)
  merged_high =  merge(n, sink_high)
  zipced = zipc(merged_low, merged_high)
  zipced ++ zipced
end

def zipc([], []) do [] end
def zipc([l|low], [h|high]) do
  cmp = comp(l,h)
  [cmp | zipc(low, high)]
end
```

Namn:______________________________ Persnr:______________________________

ett nätverk för 2^k [4*]

Nu är det bara `setup/2` kvar som skall utökas till att hantera godtyckliga n $(2, 4, 8, ...)$. Utöka funktionen med ett generellt fall, använd dig av `div/2`, `split/2` och naturligtvis funktionerna `merge/2` och `cross/2` som vi kan anta fungera för alla n.

Svar:

```
def setup(n, sinks) do
  n = div(n,2)
  {sink_low, sink_high} = split(sinks, n)
  merge_low = merge(n, sink_low)
  merge_high = merge(n, sink_high)
  {cross_low, cross_high} = cross(merge_low, merge_high)
  in_low = setup(n, cross_low)
  in_high = setup(n, cross_high)
  in_low ++ in_high
```

Appendix - operational semantics

pattern matching

$$\frac{a \equiv s}{P\sigma(a, s) \to \sigma} \qquad \frac{a \not\equiv s}{P\sigma(a, s) \to \text{fail}}$$

$$\frac{v/t \notin \sigma}{P\sigma(v, s) \to \{v/s\} \cup \sigma} \qquad \frac{v/t \in \sigma \qquad t \not\equiv s}{P\sigma(v, s) \to \text{fail}}$$

$$\frac{v/s \in \sigma}{P\sigma(v, s) \to \sigma} \qquad \frac{}{P\sigma(\_, s) \to \sigma}$$

$$\frac{P\sigma(p_1, s_1) \to \sigma' \wedge P\sigma'(p_2, s_2) \to \theta}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \to \theta}$$

$$\frac{P\sigma(p_1, s_1) \to \text{fail}}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \to \text{fail}} \qquad \frac{P\sigma(p_1, s_1) \to \sigma' \wedge P\sigma'(p_2, s_2) \to \text{fail}}{P\sigma(\{p_1, p_2\}, \{s_1, s_2\}) \to \text{fail}}$$

scoping

$$\frac{\sigma' = \sigma \setminus \{v/t \quad | \quad v/t \in \sigma \quad \wedge \quad v \quad \text{in} \quad p\}}{S(\sigma, p) \to \sigma'}$$

expressions

$$\frac{a \equiv s}{E\sigma(a) \to s} \qquad \frac{v/s \in \sigma}{E\sigma(v) \to s} \qquad \frac{v/s \notin \sigma}{E\sigma(v) \to \perp}$$

$$\frac{E\sigma(e_1) \to s_1 \qquad E\sigma(e_2) \to s_2}{E\sigma(\{e_1, e_2\}) \to \{s_1, s_2\}} \qquad \frac{E\sigma(e_i) \to \perp}{E\sigma(\{e_1, e_2\}) \to \perp}$$

$$\frac{E\sigma(e) \to t \qquad S(\sigma, p) \to \sigma' \qquad P\sigma'(p, t) \to \theta \qquad E\theta(\text{sequence}) \to s}{E\sigma(p = e; \text{sequence}) \to s}$$

$$\frac{E\sigma(e) \to t \qquad S(\sigma, p) \to \sigma' \qquad P\sigma'(p, t) \to \text{fail}}{E\sigma(p = e; \text{sequence}) \to \perp}$$

$$\frac{E\sigma(e) \to \perp}{E\sigma(p = e; \text{sequence}) \to \perp}$$