Programowanie funkcyjne 2016

Grupa kzi
Lista 7
(6.12.2016)

Uwaga†: Jeśli zadanie jest niedospecyfikowane, to braki w specyfikacji można uzupełnić w dowolny sensowny(!) sposób.

Zadania w OCamlu

W zadaniach nie można wykorzystywać pętli.

1. W pliku `scl.mli` (Server-Client Library) zdefiniuj:

   a) sygnaturę `channel`, która zawiera funkcje:
   `init : t_init -> t`,
   `close : t -> unit`,
   `recieve : t -> t_data`,
   `send : t_data -> t -> unit`,

   b) sygnaturę `serial`, która zawiera funkcje:
   `deserial : t_serial -> t`,
   `serial : t -> t_serial`,

   c) sygnaturę `processor`, która zawiera funkcję `process : t -> t`,

   d) funktor `Server`, który przyjmuje moduł C o sygnaturze `channel`, moduł S o sygnaturze `serial` i moduł P o sygnaturze `processor`, i zwraca moduł, który zawiera funkcję `run :` C`.t_init -> unit`,

   e) funktor `Client`, który przyjmuje moduł C o sygnaturze `channel` i moduł S o sygnaturze serial, i zwraca moduł, który zawiera funkcję `ask :` C`.t_init ->` S`.t ->` S`.t`,

   gdzie `t` , `t_init` , `t_data` , `t_serial` to typy abstrakcyjne zadeklarowane lokalnie w każdej sygnaturze (w której występują) takie, że w punktach d), e) zachodzą równości C`.t_data =` S`.t_serial` i S`.t =` P`.t`. (5 pkt)

2. W pliku `scl.ml` zdefiniuj:

   a) sygnatury z Zadania 1,

   b) funktor `Server`, gdzie `run` i inicjalizuje kanał za pomocą i, odbiera z niego dane, deserializuje, przetwarza, wysyła wynik, zamyka kanał, a następnie zaczyna wszystko od początku,

   c) funktor `Client`, gdzie `ask` i q inicjalizuje kanał za pomocą i, serializuje q, wysyła, odbiera wynik r, zamyka kanał i zwraca r. (4 PKT)

3. Skompiluj interfejs `scl.mli`, a następnie moduł `scl.ml`. (Kolejność jest ważna.) (1 PKT)

4. Zdefiniuj jakiś moduł implementujący sygnaturę `channal`. Jeśli nie chcemy się przepracowywać może on wyglądać, na przykład, tak:

   - `t_init`, `t` to typy pary ścieżek do plików,
   - `init` może być funkcją identyczności,
   - `close` może po prostu zwracać zwracać `()`,
   - `recieve` (*path_in*, *path_out*) próbuje otworzyć plik pod ścieżką *path_in*, wczytać go do stringa s, a następnie zamknąć, usunąć i zwrócić s, a jeśli się nie uda na chwilę zasypia – np. `Unix.sleep` (o dziwo powinno działać też pod Windowsem) – i próbuje jeszcze raz,
   - `send` (*path_in*, *path_out*) próbuje otworzyć plik pod ścieżką *path_out* i zapisać do niego stringa, jeśli mu się nie uda zasypia, a potem próbuje jeszcze raz. (5 PKT)

5. Zdefiniuj moduł implementujący sygnaturę `serial`, który serializuje listy napisów do stringów. Dla ułatwienia można założyć, że jakiś znak nigdy nie występuje w naszych napisach. (3 PKT)

6. Zdefiniuj moduł implementujący sygnaturę `processor`, w którym `process ["ping"]` zwraca `["pong"]`, `process ["zostaw wiadomosc", `*temat*`, `*treść*` ]` zapamiętuje (w jakiejś referencji) skojarzenie *temat* $\mapsto$ *treść* i zwraca `[]`, `process ["odczytaj wiadomosc", `*temat*`]` zwraca singleton z najnowszą zapamiętaną wiadomość o wskazanym temacie. (5 PKT)

7. Zaaplikuj `Server` i `Client` do modułów zdefiniowanych w zadaniach 4-6 i przetestuj. (1 PKT)

Uwaga: Zauważ, że wymienienie modułów kanału i serializacji wystarcza, żeby łączyć się np. przez TCP/UDP.
Uwaga: Oczywiście ta lista jest tylko szkicem architektury klient-serwer, a nie rozwiązaniem, z którym można iść „na produkcję".