Programowanie funkcyjne 2014
Lista 7

kzi

2 grudnia 2014

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

Zadania w OCaml'u

W zadaniach nie można wykorzystywać pętli.
Uwaga: W OCaml'u typ modułu to t samo co sygnatura, tyle że nazwana. (Francuzi...)

1. W pliku `scl.mli` (Server-Client Library) zadeklaruj:
   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 funkcje: `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 -> 'a`,
   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) oraz w punktach d) e) pomiędzy typami zachodzą równości C`.t_data =` S`.t_serial`, S`.t =` P`.t`. `(5pkt)`
2. W pliku `scl.ml` zdefiniuj:
   a) sygnatury z zadania 1 (czyli de facto skopiuj z `scl.mli`),
   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. `(4pkt)`
3. Skompiluj interfejs `scl.mli`, a następnie moduł `scl.ml`. (Kolejność jest ważna.) `(1pkt)`
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 funkcja 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, jeśli się nie uda na chwilę zasypia — np. Unix.sleep (o dziwo powinno działać też pod Windows'em) — i próbuje jeszcze raz,
   - `send (`*path_in*`, `*name_out*`)` próbuje otworzyć plik pod ścieżką *path_out* i zapisać do niego stringa, jeśli mu się nie uda zasypia, a potem próbuje jeszcze raz. `(5pkt)`
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. `(3pkt)`
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. `(5pkt)`
7. Zaaplikuj `Server` i `Client` do modułów zdefiniowanych w zadaniach 4-6 i przetestuj. `(1pkt)`

Uwaga: Zauważ, że wymienienie modułów kanału i serializacji wystarcza, żeby łączyć się np. przez TCP/UDP. Wymiana samej serializacji może się przydać, jeśli chcemy postawić jakąś aplikację pomiędzy klientem a serwerem. Zauważ też, że ta lista jest tylko szkicem architektury klient-serwer, a nie gotowym rozwiązaniem.