ПРОЕКТИРОВАНИЕ И ДЕКОМПОЗИЦИЯ ДВУНАПРАВЛЕННЫХ ПОТОКОВ ДАННЫХ ИНТЕРАКТИВНЫХ СИСТЕМ: ФОРМАЛЬНОЕ ОПИСАНИЕ ИНТЕРАКТИВНОЙ СИСТЕМЫ (ЧАСТЬ 2)

А.Г. ПИСКУНОВ

3 марта 2009 г.

АННОТАЦИЯ

Документ содержит предложения про проектированию архитектуры интерактивных приложений и декомпозицию двунаправленных потоков данных приложение - пользователь. Разделение потоков данных по уровням абстракции позволяет выделять наиболее независимые друг от друга компоненты приложения, для проектирования которых может применяться метод STS-декомпозиции Майерса.

Содержание

1 ВВЕДЕНИЕ

Документ содержит пример формального описание интерактивной системы на языке формальных спецификаций RAISE SPECIFICATION LANGUAGE (cm. [3]) . Введение и принципы этого описания изложены в [1] .

1.1 Руководство по методу проектирования типов из метода формальных спецификаций RAISE

Перевод краткого изложения метода проектирования типов из [4] разделы 2.6.6 стр. 47, 2.8.4.1 стр 90.

1.2 Выбор стиля спецификаций

В выборе стиля спецификаций существует четыре альтернативы:

- аппликативный последовательный - функциональное программирование без переменных и параллельных вычислений(concurrency);
- императивный последовательный - программирование с переменными, присваиванием, циклами и т.д., но без параллельных вычислений;
- аппликативынй конкурентный - функциональное программирование с параллельными вычислениями;
- императивный конкурентный - программирование с переменными, присваиванием, циклами и т.д., и с параллельными вычислениями;

Аппликативынй конкурентный стиль считается неподходящим как базис для программирования на языке реализации. Остается три альтернативы, которые будут далее называться аппликативная, императивная и конкурентная.

1.3 Абстракность

Также как и разделение между аппликативным и императивным; последовательным или конкурентным, отмечается разница между абстрактным и конкретным стилями. Под абстрактностью мы понимаем создание спецификаций, которые оставляют так много открытых альтернативных путей разработки как возможно. Другими словами, чем меньше решений проектирования мы записали в спецификацию тем более абстрактной она является. Под решениями проектирования мы понимаем вещи вроде таких

- решения как определить модуль;
- решения о конкретной структуре данных;
- решения о конкретных алгоритмов;

- решения о используемых переменных;
- решения какие каналы и образцы обмена информации для использования.

Противоположностью абстракности есть конкретность. Различия между ними не есть черно белыми, однако дают возможность охарактеризовать некоторый модуль, написанный в стиле одной из трех категорий, как более абстрактный или более конкретный.

- абстрактный аппликативный - модуль содержит абстрактные типы и сигнатуры функций с аксиомами, а не явные определения функций;
- конкретный аппликативный - модуль содержит конкретные типы и явные определения функций;
- абстрактный императивный - модуль не определяет переменных, зато использует ключевое слово any в описании его доступа. Содержит аксиомы;
- конкретный императивный - модуль содержит определение переменных и явные определения функций;
- абстрактный конкурентный - модуль не содержит определения каналов, однако содержит ключевое слово any в описании доступа. Содержит аксиомы;
- конкретный конкурентный - модуль содержит явные определения переменных, каналов и функций;

И опять надо подчеркнуть, что эти различия более относительные, чем абсолютные. Модуль может быть абстрактный в одном смысле и конкретный в другом. И, естественно, вся спецификация будет содержать модули сочетающие все три стилей и две степени абстрации.

1.4 Проектирование типа данных для абстракного аппликативного стиля спецификации

Чтобы создать модель системы мы следуем методу, который будет подробно изложен в секции 2.8.4.1 в [4]. Но так как мы описываем систему, а не тип данных вроде очереди, мы будем формулировать свойства системы которые удатся специфицировать в текущий момент. Мы также сформулируем понятие целостности типа, которое выразим в виде предиката. Метод вкратце:

- Определить название проектируемого типа (типа интереса);
- Определить сигнатуры необходимых функций;

- Определить эти функции как функции перехода (generetor), ([2]) если тип интереса (или тип зависящий от него) появляется в типе результата; или как функции выхода (observer) в другом случае. (Забегая вперед, скажем, что императивная версия функций состояния есть функция, которая меняет состояние). {Мы обнаружили что имеем три генератора: arrives, docks, leaves; и указали два обсервера: wainting и occupancy};
- Сформулировать предусловия (predcondition) которые должны выполняться для каждой частичной функции. {Все три генератора нашего примера являются частичными: существуют ситуации, когда их нельзя применять}. Для этих ситуаций зададим функции защиты (guard) чтобы выразить это предусловия {: can_arrive, etc}. Все функции защиты должны быть выведены (то есть им можно дать конкретное определение я терминах функций выхода) из функций выхода;
- Определить функцию целостности (consistent) типа, делая ее еще одной выводимой функций состояния;
- Для каждой возможной пары невыводимой функции состояния и невыводимой функции выхода определить аксиому, выражающую отношение между ними. {Так как мы имеем три невыводимых генератора и два невыводимых обсервера, то получаем шесть таких аксиом};
- добавить аксиомы выражающие представление о то что невыводимые функции перехода поддерживают целостность типа. {Мы имеем три такие аксиомы};
- Инкапсулировать функцию целостности типа (consistent) и все то, в чем не нуждаются клиенты модуля.

2 ОПИСАНИЕ

2.1 Уровень Абстракции Форма

Описание объекта Форма (панель) дополняет описание системы, изложенное в [1] в разделах Уровень Абстракции Приложение и Уровень Абстракции Операционная Система. Схемы PAN_CON, PAN0_CON, PAN1_CON содержат уточнение типов и функций, обьявленных в схеме CNT_ABS и, таким образом, конкретизирует ее.

Для доказательства того, что все свойства абстрактной схемы наследуются (реализуются, implement) конкретной используется следующая схема

—- Start of Scheme

rsl/PAN_ABS, rsl/PAN_CON

devt_relation

pan_test(PAN_CON **for** PAN_ABS) : |– PAN_CON $\preceq$ PAN_ABS

—- End Of Scheme

2.1.1 Конкретное описание панели

Как видно по схеме, Panel состоит из названия, признака модальности, списка структур визуализации множеств, номера активной структуры, команды, по которой открывалось следующая модальная панель. Каждая структура визуализации множеств определяет свое окно данных на панели (область экрана через которую пользователь видит данные). Номер активной структуры - показывает окно данных с фокусом ввода.

Через канал входа PanIn передаются данные двух типов:

- Pointer, как обычно, обеспечивает навигацию между структурами setVisi - визуализация множества;
- PanCommand - команды, часть которых транслируется в SQL операторы, которые выполняются над множествами, базой данных и файловой системой;

Полностью описан тип MenuItem - он состоит из названия элемента меню, названия панели, которую должен открывать данный элемент меню и типа DataStats-list. Каждая величина DataStats содержит данные, позволяющие возможность трансляции некоторых команд типа PanCommand в операторы языка SQL. Например, для команды Select должны быть данные, позволяющие построить SQL оператор SELECT, извлечь множество записей из Базы Данных и заполнить им величину типа SetVisi. Каждый элемент списка DataStats создает отдельную величину типа SetVisi. Таким образом, определяется количество окон данных в панели.

—- Start of Scheme

rsl/T0, rsl/SET_ABS

```
scheme PAN1_CON =
  with T0 in
  extend SET_ABS with
  class
    type
      PanInput =
        Pointer – –          move focus to next or prev
        – –
        |
        PanCommand – –          panel, or close current panel
```

```
      – –   menu item text
      – –            name of panel
      – –
    ,
    MenuItem = Text × Text × DataStats* ,
    PanCommand ==
      Add |
      Delete |
      Edit |
      Select |
      Filter |
      Order |
      Export |
      Exit |
      Cancel,
    PanOutput
    – – panel name                    – – active set,
    – – how was next panel opened?
    ,
    Panel =
      Text × Bool × SetVisi* × Nat × PanCommand

  channel panIn : PanInput, panOut : PanOutput

  value
    mkPanOut :
      Text × Bool × SetVisi* × Nat → PanOutput,
    null : Answer
end
```

—- End Of Scheme

Функции isModal и save - понятные.

Далее, рассмотрим функцию, которая обрабатывает команды из главного меню приложения: openNewOrSelectOldTable. Если в списке панелей находится панель с подходящим названием, она переставляет эту панель в голову списка, или, в противном случае, в начало списка ps ставит новую панель. Причем величины setVisi для новой панели создаются из данных типа DataStats.

Фукнция proc получает на вход pars - результаты работы ранее открытой панели и выполняет некоторую работу (зависящую от команды которая ранее привела к открытию упомянутой панели). В частности, по команде Export будет вызвана функция вывода данных в файл с указанным именем, в случаях команд Add, Delete, Filter, Order - должно быть выполнено обновление текущей

структуры данных SetVisi, по команде Edit - обновление отредактированной записи в текущей стуктуре SetVisi.

—- Start of Scheme

rsl/T0, rsl/PAN1_CON

```
scheme PAN0_CON =
  with T0 in
  extend PAN1_CON with
  class
    value
      isModal : Panel → Bool
      isModal(nm, isMdl, svs, cur, cmd) ≡ isMdl,

      save : Panel* → write any Unit
      save(ps) ≡
        if len ps > 0
        then
          let (nm, isMdl, svs, cur, cmd) = hd ps in
            − −                save (nm, isMdl, cur. cmd);save(svs);
            save(tl ps) end
        end,

      openNewOrSelectOldTable :
        MenuItem × Panel* →
          write any in any out any Panel*
      openNewOrSelectOldTable((mNm, pNm, dts), ps) ≡
        let l = len ps in
          if l > 0
          then
            if isModal(hd ps) then ps
            else
              let i = panIdx(ps, pNm) in
                if i > 0
                then
                  ⟨ ps(i)⟩ ⌢
                  ⟨ ps(k) | k in ⟨ 1 .. i − 1⟩ ⟩ ⌢
                  ⟨ ps(k) | k in ⟨ i + 1 .. l⟩ ⟩
                else
                  ⟨ (pNm, isModal(dts),
                    select(mkSetVisi(dts)), 1, Cancel)⟩ ⌢
                  ps
                end
              end
            end
```

```
    else
      ⟨ (pNm, isModal(dts), select(mkSetVisi(dts)), 1,
        Cancel)⟩
    end
  end,

panIdx : Panel* × Text → Nat
panIdx(ps, nm) ≡
  if len ps > 0
  then
    let (n, isMdl, svs, cur, cmd) = hd ps in
      if nm = n then 1 else panIdx(tl ps, nm) + 1 end
    end
  else 0
  end,

proc :
  Panel* × Answer →
    write any Panel
    * − − to procees result of prev panel
proc(pnls, pars) ≡
  if len pnls > 0
  then
    let (nm, isM, svs, cur, cmd) = hd pnls in
      ⟨ (nm, isM, proc(svs, cur, pars, cmd), cur, cmd)
      ⟩ ⌢ tl pnls
    end
  else ⟨ ⟩
  end,

proc :
  SetVisi* × Nat × Answer × PanCommand ~→
    write any SetVisi*
proc(svs, cur, pars, cmd) ≡
  if cur ≤ len svs ∧ cur > 0
  then
    case cmd of
      Delete → select(svs, cur),
      Add → select(svs, cur),
      Filter → selectWhere(svs, cur, pars),
      Order → selectOrderBy(svs, cur, pars),
      Edit → selectRecord(svs, cur),
      Export → export(svs(cur), pars) ; svs,
      _ → svs
    end
```

```
        else svs
        end
  end
```

—- End Of Scheme

Функция pan делит данные из входного канала между двумя фунциями. Данные типа Pointer используется для смены множества, Данные типа PanCommand транслируются действиями над множествами. Причем команды Add, Edit, Filter, Order, Export влекут создание новой модальной панели. Предыдущая панель запоминает по какой команды вызывалась следующая панель с тем, чтобы после закрытия последней правильно обработать ее результаты

—- Start of Scheme

rsl/T0, rsl/PAN0_CON

```
scheme PAN_CON =
  with T0 in
  extend PAN0_CON with
  class
    value
      pan :
        Panel →
          write any in panIn, any out panOut, any
          Panel* − − it ≡ possible to open next panel
          × Answer
      pan(nm, isMdl, svs, cur, cmd) ≡
        panOut!mkPanOut(nm, isMdl, svs, cur) ;
        let
          c =
            if len svs > 0 then 1 + cur  len svs else 0 end,
          svs1 = setVisi(svs, cur),
          pCmd = panIn?
        in
          case pCmd of
            PanInput_from_Pointer(p) →
              (⟨ (nm, isMdl, svs1, handler(p, svs1, c), cmd)
              ⟩ , null),
            PanInput_from_PanCommand(cmd) →
              if cmd = Exit then (⟨ ⟩ , exit(svs1(cur)))
              elsif cmd = Cancel then (⟨ ⟩ , null)
              else
                (handler(cmd, (nm, isMdl, svs1, c, cmd)),
                null)
```

```
        end
    end
    []
    (⟨ (nm, isMdl, svs1, c, cmd)⟩ , null)
  end
− − to change active set
,

handler : Pointer × SetVisi* × Nat → Nat
handler(p, svs, cur) ≡
  if len svs > 0
  then
    case p of
      first → 1,
      prev →
        let pr = (cur − 1)  len svs in
          if pr = 0 then len svs else pr end
        end,
      next →
        let ne = (cur + 1)  len svs in
          if ne = 0 then len svs else ne end
        end,
      last → len svs,
      _ → cur
    end
  else 0
  end,

handler : PanCommand × Panel → write any Panel*
handler(pCmd, (nm, isMdl, svs, cur, cmd)) ≡
  case pCmd of
    Add →
      let
        dss =
          dataStats(svs(cur), false) − − add ↦ false
      in
        ⟨ (name(dss, "to add "^ nm), true,
          mkSetVisi(dss, keys(svs(cur))), 1, Cancel),
          (nm, isMdl, svs, cur, pCmd)⟩
      end,
    Edit →
      let
        dss =
          dataStats(svs(cur), true) − − edit ↦ true
      in
        ⟨ (name(dss, "to edit "^ nm), true,
```

```
                mkSetVisi(dss, keys(svs(cur))), 1, Cancel),
                (nm, isMdl, svs, cur, pCmd)⟩
            end,
          Filter →
            ⟨ ("to set filter for "^ nm, true,
              mkSetVisi(flds(svs(cur)), false), 1, Cancel),
              (nm, isMdl, svs, cur, pCmd)⟩ ,
          Order →
            ⟨ ("to set order for "^ nm, true,
              mkSetVisi(flds(svs(cur)), true), 1, Cancel),
              (nm, isMdl, svs, cur, pCmd)⟩ ,
          Export →
            ⟨ ("to set file for export of"^ nm, true,
              mkSetVisi(
                  "to put name of file for export
                  fileNames()), 1, Cancel),
              (nm, isMdl, svs, cur, pCmd)⟩ ,
          Delete →
            ⟨ (nm, isMdl, delete(svs, cur), cur, Cancel)⟩ ,
          Select → ⟨ (nm, isMdl, select(svs), cur, Cancel)⟩
        end
  end


—- End Of Scheme
```

2.1.2 Абстрактная схема Визуализация Множества

Схема обьявляет тип SetVisi для описания множества; тип DataStats должен, в частности, содержать операторы работы с базой данных; тип Answer - для передачи результатов работы следующей верхней формы (в которой задается сортировка наприме) в предыдущую (в которой надо выполнить запрос с измененным order by) сигнатуры функций которые потребовались в ходе проектирования функций уровня Форма.

```
    —- Start of Scheme

rsl/T0

scheme SET_ABS =
  with T0 in
  class
    type
      SetVisi − − to visualise   set
      ,
```

```
  DataStats − −   make query to  database select, insert, delete,
  − − update
  ,
  Answer − − data to make some select or export or so on...

value
  setVisi :
    SetVisi* × Nat ~→ in any out any SetVisi* ,
  delete :
    SetVisi* × Nat ~→ write any SetVisi* ,
  select : SetVisi* → write any SetVisi* ,
  select :
    SetVisi* × Nat ~→ write any SetVisi* ,
  selectWhere :
    SetVisi* × Nat × Answer →
      write any SetVisi* ,
  selectOrderBy :
    SetVisi* × Nat × Answer →
      write any SetVisi* ,
  selectRecord :
    SetVisi* × Nat → write any SetVisi* ,
  export : SetVisi × Answer → write any Unit
  − − to select list of fields of some SetVisi
  ,
  flds : SetVisi → Text* ,
  exit : SetVisi → write any Answer,
  dataStats : SetVisi × Bool → DataStats* ,
  keys : SetVisi → Text* ,
  isModal : DataStats* → Bool,
  mkSetVisi :
    Text* × Bool → SetVisi
      * − − to make panel for filter and order
  ,
  mkSetVisi : DataStats* → SetVisi* ,
  mkSetVisi :
    DataStats* × Text* → SetVisi* ,
  mkSetVisi : Text × Text* → SetVisi*
  − − dataSets to make next panel   false ↦ add,
  − − true ↦ edit
  ,
  name :
    DataStats* × Text →
      Text − − to make title of next panel
  ,
  fileNames :
```

```
      Unit → Text* − − to make list of name of files
  end
```

—- End Of Scheme

2.2 Уровень Абстракции Визуализация Множества

Попытаемся разобраться с визуализацией. Показывать пользователю будем данные ввиде таблицы через некоторое окно просмотра. Для этого определим

- Pair - пару точек или размеров;
- Data - данные, которые можно будет просматривать через окно;
- Конкретизируем тип SetVisi. Он состоит из названия, размер множества (размер дается в ячейках таблицы, то есть, задано количество колонок, количество записей), размер окна (в символах), через которое мы смотрим на множество, расположение окна (в ячейках), расположение фокуса на множестве (в ячейках). Желательно, чтобы фокус был в области окна. Еще нужен список размеров каждой колонки в символах. Начало координат в левоей верхней точке. Координаты увеличиваются вправо, вниз;
- конкретизируем тип Answer - просто список строчек;
- SetInput - тип является прямым произведение типа Pointer и логического типа. Команды этого типа будут управлять навигацией по таблице, а не по списку как раньше. То есть, фокус и окно будут двигаться в двух направлениях - вертикальном и горизонтальном.
- SetOutput.

—- Start of Scheme

rsl/T0, rsl/SETDT_CON

```
scheme SETVSDT_CON =
  with T0 in
  extend SETDT_CON with
  class
    type
      Pair ::
        x : Nat ↔ x
        y :
          Nat ↔ y − −
```

```
        − − set size   window_position window_size
    ,
    SetVisi =
      Text × Pair × Pair × Pair × Nat*
      − − focus_position
      × Pair × Data
    − − to visualise   set
    ,
    Data,
    Answer =
      Text
      * − − data to make some select or export or so on...
    ,
    SetOutput,
    SetInput = Pointer × Bool − − vertical or horizontal

  channel setIn : SetInput, setOut : SetOutput

  value
    wf : SetVisi → Bool
    wf(nm, sSz, wPos, wSz, ls, fPos, dt) ≡
      x(sSz) = len ls ∧ x(wPos) ≥ 0 ∧ x(wPos) < x(sSz),

    mkSetOut : SetVisi → SetOutput,
    item : SetVisi → Item,
    update : SetVisi × Item → SetVisi,
    exit : Data → Text* − − to return some list
end
```

—- End Of Scheme

2.2.1 Конкретное описание схемы Визуализация Множества

Доказательства того, что все свойства абстрактной схемы Визуализация Множества наследуются (реализуются, implement) конкретной используется следующая схема

```
  —- Start of Scheme
```

rsl/SET_ABS, rsl/SETLST_CON

devt_relation
set_test(SETLST_CON **for** SET_ABS) : |− SETLST_CON ≼ SET_ABS

—- End Of Scheme

2.2.1.1 Переход от списка к одному окну просмотра Схема SetLst_con описывает функции, имеющие в своей сигнатуре список окон просмотра.

—- Start of Scheme

rsl/T0, rsl/SET_CON

```
scheme SETLST_CON =
  with T0 in
  extend SET_CON with
  class
    value
      setVisi :
        SetVisi* × Nat ~→ in any out any SetVisi*
      setVisi(svs, cr) ≡
        if len svs > 0 ∧ cr > 0
        then
          if cr = 1 then ⟨ setVisi(hd svs)⟩ ⌢ tl svs
          else ⟨ hd svs⟩ ⌢ setVisi(tl svs, cr − 1)
          end
        else svs
        end
      − −
      ,

      select : SetVisi* → write any SetVisi*
      select(svs) ≡
        if len svs > 0
        then
          ⟨ select(hd svs, false, ⟨ ⟩ , ⟨ ⟩ )⟩ ⌢
          select(tl svs)
        else ⟨ ⟩
        end,

      select :
        SetVisi* × Nat ~→ write any SetVisi*
      select(svs, no) ≡ select(svs, no, false, ⟨ ⟩ , ⟨ ⟩ ),

      selectRecord :
        SetVisi* × Nat ~→ write any SetVisi*
      selectRecord(svs, no) ≡
        select(svs, no, true, ⟨ ⟩ , ⟨ ⟩ ),
```

```
    selectWhere :
      SetVisi* × Nat × Text* →
        write any SetVisi*
    selectWhere(svs, no, wheres) ≡
      select(svs, no, false, wheres, ⟨ ⟩ ),

    selectOrderBy :
      SetVisi* × Nat × Text* →
        write any SetVisi*
    selectOrderBy(svs, no, oBys) ≡
      select(svs, no, false, ⟨ ⟩ , oBys),

    select :
      SetVisi* × Nat × Bool × Text* × Text* ~→
        write any SetVisi*
    select(svs, no, allCur, wheres, oBys) ≡
      if no > 1
      then
        ⟨ hd svs⟩ ⌢
        select(tl svs, no − 1, allCur, wheres, oBys)
      elsif no = 1
        then
          ⟨ select(hd svs, allCur, wheres, oBys)⟩ ⌢
          (tl svs)
      else svs
      end,

    delete :
      SetVisi* × Nat ~→ write any SetVisi*
    delete(svs, no) ≡
      if no > 1 then ⟨ hd svs⟩ ⌢ delete(tl svs, no − 1)
      elsif no = 1 then ⟨ delete(hd svs)⟩ ⌢ (tl svs)
      else svs
      end
end
```

—- End Of Scheme

2.2.1.2 Основная обработка команд - функция SetVisi Схема описывает управление окна просмотра, через которое пользователь может смотреть некую таблицу (которая и содержит множество) и фокусом ввода. Функция setVisi отличается от аналогичных pan и app, тем, что ввод/вывод в/из каналы(ов) другого уровня (в данном случае itemIn, itemOut) выполняется не всегда, а по команде curr. См.

```
update(sv, item(item(sv)))
```

Из данных таблицы - sv извлекается текущий Item, затем происходит редактирование извлеченной структуры с вводом-выводом данных из каналов уровня Item (вторая функция item), после чего выполняется обновление данных таблицы (update). Остальные команды занимаются перемещением фокуса ввода и окна просмотра через функцию focus.

—- Start of Scheme

rsl/T0, rsl/SETVS_CON, rsl/SETDB_CON

```
scheme SET_CON =
  with T0 in
  extend SETVS_CON with
  extend SETDB_CON with
  class
    value
      setVisi :
        SetVisi → in setIn, any out setOut, any SetVisi
      setVisi(sv) ≡
        setOut!mkSetOut(sv) ;
        let (cmd, dir) = setIn? in
          if cmd = curr then update(sv, item(item(sv)))
          else focus(sv, (cmd, dir))
          end
        end
        []
        sv,

      focus : SetVisi × SetInput ~→ SetVisi
      focus(sv, (cmd, dir)) ≡
        case (cmd, dir) of
          (first, true) → move(sv, d_left, di_max),
          (prev, true) → move(sv, d_left, di_item),
          (next, true) → move(sv, d_right, di_item),
          (last, true) → move(sv, d_right, di_max),
          (first, false) → move(sv, d_up, di_max),
          (prev, false) → move(sv, d_up, di_item),
          (next, false) → move(sv, d_down, di_item),
          (last, false) → move(sv, d_down, di_max),
          _ → sv
        end
  end
```

—- End Of Scheme

2.2.1.3 Навигация фокуса и окна просмотра При движении окна просмотра требуется видеть хотя бы одну колоку и хотя бы одну строчку.

- canMvFcs - проверяет возможность сдвинуть фокус;
- canMvWnd - проверяет возможность сдвинуть окно просмотра;
- mvFcs - движение фокуса;
- mvWnd - движение окна;
- sum - сумма чисел в списке. В данном контексте (функция canMvFcs) вычисляется сумма длин всех колонок от левого края окна просмотра до фокуса, если полученная сумма меньше длины окна, то фокус можно сдвинуть вправо.

—- Start of Scheme

rsl/T0, rsl/SETVSFS_CON

```
scheme SETVS_CON =
  with T0 in
  extend SETVSFS_CON with
  class
    type
      Direction == d_left | d_right | d_up | d_down,
      Distance == di_item | di_max

    value
      move : SetVisi × Direction × Distance → SetVisi
      move(sv, dir, dis) ≡
        case (dir, dis) of
          (d_left, di_max) →
            mvFcs(mvWnd(sv, d_left, di_max), d_left, di_max),
          (d_left, di_item) →
            if canMvFcs(sv, d_left)
            then mvFcs(sv, d_left, di_item)
            elsif canMvWnd(sv, d_left)
              then
                mvFcs(
                  mvWnd(sv, d_left, di_item), d_left,
                  di_item)
            else sv
            end,
          (d_right, di_item) →
            if canMvFcs(sv, d_right)
            then mvFcs(sv, d_right, di_item)
```

```
      elsif canMvWnd(sv, d_right)
        then
          mvFcs(
            mvWnd(sv, d_right, di_item), d_right,
            di_item)
      else sv
      end,
    (d_right, di_max) →
      mvFcs(
        mvWnd(sv, d_right, di_max), d_right, di_max),
    (d_up, di_max) →
      mvFcs(mvWnd(sv, d_up, di_max), d_up, di_max),
    (d_up, di_item) →
      if canMvFcs(sv, d_up)
      then mvFcs(sv, d_up, di_item)
      elsif canMvWnd(sv, d_up)
        then
          mvFcs(
            mvWnd(sv, d_up, di_item), d_up, di_item)
      else sv
      end,
    (d_down, di_item) →
      if canMvFcs(sv, d_down)
      then mvFcs(sv, d_down, di_item)
      elsif canMvWnd(sv, d_down)
        then
          mvFcs(
            mvWnd(sv, d_down, di_item), d_down,
            di_item)
      else sv
      end,
    (d_down, di_max) →
      mvFcs(mvWnd(sv, d_down, di_max), d_down, di_max),
    _ → sv
  end
– –     ,canMvFcs : SetVisi ×    Direction × Distance
– – ~→ Bool
– –     canMvFcs ((sNm, sSz, wPos, wSz, ls,   fPos,
– – sDt),   dir, dis) is
,

canMvFcs : SetVisi × Direction ~→ Bool
canMvFcs((sNm, sSz, wPos, wSz, ls, fPos, sDt), dir) ≡
  case dir of
    d_left →
```

```
      if x(fPos) > x(wPos) then true else false end,
    d_right →
      if sum(ls, x(wPos), x(fPos)) < x(wSz) then true
      else false
      end
    − − if length from x(wPos) to x(fPos) less  then
    − − x(sSz) it ≡
    − − possible to move
    ,
    d_up →
      if y(fPos) > y(wPos) then true else false end,
    d_down →
      if y(fPos) < y(wPos) then true else false end
  end,

mvFcs : SetVisi × Direction × Distance ~→ SetVisi
mvFcs(sv, dir, dis) ≡
  let (sNm, sSz, wPos, wSz, ls, fPos, sDt) = sv in
    case (dir, dis) of
      (d_left, di_max) →
        (sNm, sSz, wPos, wSz, ls, mk_Pair(0, y(fPos)),
        sDt),
      (d_left, di_item) →
        if x(fPos) > x(wPos)
        then
          (sNm, sSz, wPos, wSz, ls,
          mk_Pair(x(fPos) − 1, y(fPos)), sDt)
        else sv
        end,
      (d_right, di_max) →
        (sNm, sSz, wPos, wSz, ls,
        mk_Pair(x(sSz) − 1, y(fPos)), sDt),
      (d_right, di_item) →
        if x(fPos) < x(wPos)
        then
          (sNm, sSz, wPos, wSz, ls,
          mk_Pair(x(fPos) + 1, y(fPos)), sDt)
        else sv
        end,
      (d_up, di_max) →
        (sNm, sSz, wPos, wSz, ls, mk_Pair(x(fPos), 0),
        sDt),
      (d_up, di_item) →
        if y(fPos) > y(wPos)
        then
```

```
          (sNm, sSz, wPos, wSz, ls,
          mk_Pair(x(fPos), y(fPos) − 1), sDt)
        else sv
        end,
      (d_down, di_max) →
        (sNm, sSz, wPos, wSz, ls,
        mk_Pair(x(fPos), y(sSz) + 1), sDt),
      (d_down, di_item) →
        if y(fPos) < y(wPos)
        then
          (sNm, sSz, wPos, wSz, ls,
          mk_Pair(x(fPos), y(fPos) + 1), sDt)
        else sv
        end
    end
  end,

mvWnd : SetVisi × Direction × Distance → SetVisi
mvWnd(sv, dir, dis) ≡
  let (sNm, sSz, wPos, wSz, ls, fPos, sDt) = sv in
    if len ls > 0
    then
      case (dir, dis) of
        (d_left, di_max) →
          (sNm, sSz, mk_Pair(0, y(wPos)), wSz, ls,
          fPos, sDt),
        (d_left, di_item) →
          if x(wPos) > 0
          then
            (sNm, sSz, mk_Pair(x(wPos) − 1, y(wPos)),
            wSz, ls, fPos, sDt)
          else sv
          end,
        (d_right, di_max) →
          (sNm, sSz, mk_Pair(len ls − 1, y(wPos)),
          wSz, ls, fPos, sDt),
        (d_right, di_item) →
          if x(wPos) < x(sSz) − 1
          then
            (sNm, sSz, mk_Pair(x(wPos) + 1, y(wPos)),
            wSz, ls, fPos, sDt)
          else sv
          end,
        (d_up, di_max) →
          (sNm, sSz, mk_Pair(x(wPos), 0), wSz, ls,
          fPos, sDt),
```

```
          (d_up, di_item) →
            if y(wPos) > 0
            then
              (sNm, sSz, mk_Pair(x(wPos), y(wPos) − 1),
              wSz, ls, fPos, sDt)
            else sv
            end,
          (d_down, di_max) →
            (sNm, sSz, mk_Pair(x(wPos), y(sSz) − 1),
            wSz, ls, fPos, sDt),
          (d_down, di_item) →
            if y(wPos) < y(sSz) − 1
            then
              (sNm, sSz, mk_Pair(x(wPos), y(wPos) + 1),
              wSz, ls, fPos, sDt)
            else sv
            end
        end
      else sv
      end
    end,

canMvWnd : SetVisi × Direction ~→ Bool
canMvWnd((sNm, sSz, wPos, wSz, ls, fPos, sDt), dir) ≡
  case dir of
    d_left → if x(wPos) > 0 then true else false end,
    d_right →
      if x(wPos) < x(sSz) − 1 then true else false end,
    d_up → if y(wPos) > 0 then true else false end,
    d_down →
      if y(wPos) < y(sSz) − 1 then true else false end
  end,

sum :
  Nat* × Nat × Nat →
    Nat − − to sum ∀ the length
sum(ls, s, e) ≡ − − from s to e
  if ls ≠ ⟨ ⟩
  then
    if s = 1 ∧ e ≥ 1
    then hd ls + sum(tl ls, 1, e − 1)
    elsif s > 1 ∧ e ≥ s
      then sum(tl ls, s − 1, e − 1)
    else 0
    end
```

```
        else 0
        end
  end
```

—- End Of Scheme

2.2.2 Абстрактная схема Элемент

—- Start of Scheme

rsl/T0

```
scheme ITEM_ABS =
  with T0 in
  class
    type
      Item − − to visualise   set

    value
      item : Item → in any out any Item
  end
```

—- End Of Scheme

2.3 Уровень Абстракции Элемент

Список литературы

[1] А.Г. Пискунов, . Raise Specification Language: проектирование и декомпозиция потоков данных интерактивного приложения. http://i.com.ua/~agp1/ru/dataFlow.html.

[2] А.Г. Пискунов. Формализация парадигмы обьектно-ориентированного программирования: критика определения Гради Буча, 2007. http://i.com.ua/~agp1/ru/oopFormalizm.html.

[3] Chris George. Introduction to RAISE. UNU-IIST report No. 249, 2002. http://users.iptelecom.net.ua/~agp1/arts/report249.pdf.

[4] The RAISE Method Group. The RAISE Development Method, 1999. http://users.iptelecom.net.ua/~agp1/arts/book.pdf.