A Small System-Structuring System in Scheme

Nilsson, Bernt

1988

Document Version:
Publisher's PDF, also known as Version of record

CODEN: LUTFD2/(TFRT-7379)/1-19/(1988)

A Small System-Structuring System in Scheme

Bernt Nilsson

Department of Automatic Control
Lund Institute of Technology
February 1988

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Report

Date of issue
February 1988

Document Number
CODEN: LUTFD2/(TFRT-7379)/1-19/(1988)

Author(s)
Bernt Nilsson

Supervisor

Sponsoring organisation

Title and subtitle
A Small System-Structuring System in Scheme

Abstract

This report is adocumentation of a project in the course *Tools and Metaphors of AI-programming.* The project is a prototype of a system-structuring system. The prototype is written in Scheme. This study shows that symbolic manipulation language is powerful in problems like this. A new idea for connecting systems is implemented in the second part of the project.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
19

Recipient's notes

Security classification

A Small System-Structuring System in Scheme

Abstract:
This report is adocumentation of a project in the course *Tools and Metaphors of AI-programming.* The project is a prototype of a system-structuring system. The prototype is written in Scheme. This study shows that symbolic manipulation language is powerful in problems like this. A new idea for connecting systems is implemented in the second part of the project.

Contents:

1. Introduction

This report is a documentation of a project in a course at the department of Automatic Control during the warm winter 1987-88. The name of the course was *Tools and Metaphors of AI-programming* and *Wolfgang Kreutzer* held the lectures. Three different programming languages was discussed in the course: Scheme (a dialect of Lisp), Smalltalk and Prolog. The project documented here is in Chez Scheme and it runs on Sun-Workstations.
The project chosen is a package for structuring systems in an interactive environment. The project has been developed in two separated parts.
The first part is a package of functions that can create systems with inputs, outputs and subsystems. There is also a `connect` function that creates connections between subsystems. This package is called *4s*.
The second part is a specialization of the connection problem. In this part a prototype for connecting cuts with different structure is developed. This package is called `CutSystem`.
In chapter two there is a tutorial to *4s*. Here are the Scheme functions shown and discussed. A tank system example is used to show how the available operations are used. `CutSystem` is shown and discussed in chapter three. There is no tutorial to `CutSystem` because it is not of any general interested. The ideas and the experiences are discussed. Finally in chapter four there is a summary of this project.

2. A tutorial to 4s

The first part of the project is a *Small System-Structuring System, 4s*. In *4s* you can create systems and give them inputs, outputs and subsystems. In systems, that has subsystems, is it possible to connect subsystems to each other in a nice interactive environment. The *4s* is a collection of Scheme functions that operates on system descriptions and these system descriptions are lists. This way of programming is called *message passing*. The Scheme code for the *4s* functions can be found in Appendix I.

2.1 Creation of systems

Systems are created with the function `make-system`. A system can also be created as a copy of an other system by using `system-copy`. The syntax for the functions are shown below.

```
(define newsystem (make-system 'newsystemname))
(define newsystem (system-copy 'newsystemname copiedsystem))
```

The result of the first function is a list with an identifier (newsystemname) and a body ().

```
(newsystem ())
```

Below follows two simple examples:

```
> (define tank1 (make-system 'tank1))
tank1
> tank1
(tank1 ())
> (define tank2 (system-copy 'tank2 tank1))
tank2
> tank2
(tank2 ())
```

2.2 Creation of cuts

A cut is a group of variable clustered under one identifier, the cut name. Creation of cuts in *4s* is similar to creation of systems. Cuts are, like systems, only lists and there are functions that can create and cluster them. Cut is a collection of other cuts in a hierarchical fashion. The lowest level is called subcuts which is composed of only one variable. A subcut is created by the function `make-subcut` with following syntax.

```
(define subcut (make-subcut 'subcutidentifier ''subcuttype''))
```

Cuts can now be created by the function `make-cut`. The cut is made of a list of cuts or subcuts and a cut type string. See below.

```
(define cut (make-cut 'cutidentifier listofsubcuts ''cuttype''))
```

Cuts and subcuts can be copied with a function called cut-copy.

```
(define newcut (cut-copy 'newcutidentifier copiedcut))
```

The result of the first function is a list with an identifier (cutidentifier) and a subcut type ("string").

```
(subcut ''subcuttype'')
```

A cut has following structure.

```
(cut ('list-of-subcuts') ''cuttype'')
```

Examples:

```
> (define flowcut (make-subcut 'q ''flow''))
flowcut
> flowcut
(q ''flow'')
> (define compcut (make-subcut 'x ''composition''))
compcut
> (define inflow (make-cut 'inflow (list flowcut compcut)
                                   ''pipe''))
inflow
> inflow
(inflow ((q ''flow'')(x ''composition'')) ''pipe'')
> (define outflow (cut-copy 'outflow inflow))
outflow
> outflow
(outflow ((q ''flow'')(x ''composition'')) ''pipe'')
```

2.3 Creation of inputs and outputs

With two functions, create-input and create-output, is it possible to create inputs to a system and create outputs from a system.

```
(define system (create-input system cut))
```

```
(define system (create-output system cut))
```

The syntax of the resulting system is following.

```
(system
   ((input 'list-of-cuts')
    (output 'list-of-cuts')))
```

Examples of how to create systems with inputs and outputs are shown below. This example is a continuation of the example above.

```
> (define tank1 (create-input tank1 inflow))
tank1
> (define tank1 (create-output tank1 outflow))
tank1
> tank1
(tank1
```

```
((input (inflow ((q ''flow'') (x ''composition''))
                ''pipe''))
 (output (outflow ((q ''flow'') (x ''composition''))
                  ''pipe''))))
```

2.4 Creation of subsystems

To create hierarchical system descriptions, there is a function that creates a defined system as a subsystem in an other, on forehand defined, system. This second system then becomes the super system. The syntax of this function is shown below.

```
(define supersystem (make-subsystem subsystem supersystem))
```

The subsystem description is inserted in the super system description. Subsystem in a new list in the super system body list. See super system structure below.

```
(system
   ((input 'list-of-cuts')
    (output 'list-of-cuts')
    (subsystem 'list-of-subsystem-description')))
```

Below follows an example where a supersystem is created and two systems are assigned as subsystems. The example is based on the examples above. First is a copy of `tank1` created and after that is the super system `tanksystem` created. The `tank1` and `tank2` systems are then defined as subsystems of the super system `tanksystem`.

```
> (define tank2 (system-copy 'tank2 tank1))
tank2
> (define tanksystem (make-system 'tanksystem))
tanksystem
> (define tanksystem (make-subsystem tank1 tanksystem))
tanksystem
> (define tanksystem (make-subsystem tank2 tanksystem))
tanksystem
> tanksystem
(tanksystem
   ((subsystem
       (tank1
          ((input (inflow ((q "flow") (x "composition"))
                          "pipe"))
           (output (outflow ((q "flow") (x "composition"))
                            "pipe"))))
       (tank2
          ((input (inflow ((q "flow") (x "composition"))
                          "pipe"))
           (output (outflow ((q "flow") (x "composition"))
                            "pipe")))))))
```

2.5 A connection mechanism

In a hierarchical system description is it important to connect subsystems to each other in a convenient way. Here in *4s*, a function called `connect` creates the desired connection and insert it in the super system.

```
(define supersystem (connect supersystem 'inputidentifier
                                          'outputidentifier))
```

The structure of the cut identifiers is.

```
(cutidentifier (subsystem))
```

The connection that is created with the `connect` function looks like as follows.

```
((inputcut (subsystem1)) (outputcut (subsystem2)))
```

Continuing with our example, we can connect `tank2` to `tank1`.

```
> (define tanksystem (connect tanksystem '(inflow (tank2))
                                         '(outflow (tank1))))
tanksystem
> tanksystem
(tanksystem
   ((subsystem
       (tank1
          ((input (inflow ((q "flow") (x "composition"))
                          "pipe"))
           (output (outflow ((q "flow") (x "composition"))
                            "pipe"))))
       (tank2
          ((input (inflow ((q "flow") (x "composition"))
                          "pipe"))
           (output (outflow ((q "flow") (x "composition"))
                            "pipe")))))
    (connection
       ((inflow ((q "flow") (x "composition")) "pipe" (tank2))
        (outflow ((q "flow") (x "composition")) "pipe" (tank1))
        ))))
```

2.6 Creation of super functions

In *4s* there are possibilities to create functions that uses already defined functions. One example of a function like this is the `make-siso-system` function. This function creates a system with one input and one output. These input and output cuts has the typing "nt", which denotes 'no type'.

```
(define sisosystem (make-siso-system 'sisosystemname))
```

An example

```
> (define s1 (make-siso-system 's1))
s1
> s1
```

```
(s1
  ((input (u (u) ''nt''))
   (output (y (y) ''nt'')))))
```

2.7 Implementation in Scheme

The implementation of *4s* is done in Chez Scheme on Sun 3/50. The Scheme code is found in appendix I. This part of the project is the first larger Lisp program that the author has done. Therefore is the programming style not so well developed.
In *4s* there are some general function operating on lists. Examples are `insert-list` and `add-to-list`. But most of the functions are quite large and specialized. They are quite hard to read because they are fiddeling with the structure of the lists that they are operating on.
On the other hand is it easy to read specialized functions that are similar to each other. The opposite programming style is to create general function and specialize them by calling them by particular arguments. Functions like this can sometimes be hard to understand in a specialized operation. This is also discussed in chapter 3.4.

2.8 Conclusion

When dealing with problems on a symbolic form is it powerful to use tools like Scheme. As shown above Scheme code is very efficient. A small amount of code can create powerful programs like *4s*. An important side effect in this application is the interactive environment provided by Scheme. In structuring systems and in modelling in general is it not common to work in an interactive environment like this.
When working with large numbers of systems is it important to make the working process more flexible and interactive. Traditional modelling can be compared with programming in languages like Pascal. The woking process become an iteration process, editing - compiling - linking - executing cycle.
A new working process is in the spirit of exploratory programming developed in the area of AI. Using languages like Scheme one have an interactive environment to work with in which it is possible to test new ideas.
To create a nice interactive environment for a modelling language is a interesting task. Languages like Scheme and Smalltalk are well suited for this task.
This problem is also well suited for *object oriented programming.*

3. the CutSystem

The `CutSystem` is a collection of Scheme functions for checking and transforming cuts with different structures. When working with lot of different subsystems is it nice to have intelligent connection mechanisms, that can connect cuts that express the same thing but with other variables.
`CutSystem` is a prototype which contains functions that operates on cuts. They can study the typing of cuts and subcuts. There are functions that can divide cuts in two parts. The first part is compatible with an other reference cut. The second part is not compatible.
When using functions like this, is it possible to transform uncompatible cut parts to compatible by using some kind of transformation rules.
In `CutSystem`, is a simple prototype with a transformation rule implemented. The transformation rule is based on the ideal gas law and it can transform cuts that express properties of gas.

3.1 Cut type control

In *4s*, the `connect` function is only checking if the different cuts has identical typing. If they have identical types, then it is assumed that the internal structure of the cuts are identical and therefore assumes that the connection is possible. If the internal structures of the cuts not are identical, then there should be a mechanism that forbid the connection. Functions that handles problems like this, are available in `CutSystem`.
The function `sort-cut-type-list` can be used to check the internal structure of a cut. This function sorts a list of the subcut types into two different parts. One part is compatible with the reference cut and one part is not. If the uncompatible part of the cut type list is empty, then the cuts are identical.

3.2 Cut transformation

Modelling large systems with a lot of different subsystems can be very hard, if there is no flexibility in the modelling language. If everything in the model must have the same notation and syntax then modelling becomes complicated. Therefore is it interesting to create mechanisms that provides tools for a more flexible way to work with large systems.
When cuts with different internal structures, but with the intention of expressing the same thing, are connected to each other, then the modelling language should have the possibility to transform the cuts, so that the connection is possible.
Examples of desired transformations are mass flow to volume flow, concentration to mass parts etc.
`CutSystem` has a function that can create a interpreter for cuts that expresses variables in the ideal gas law.

3.3 A cut interpreter

A simple prototype for transformation of cuts that express properties of a gas tube, following the ideal gas law.

$$pV = RnT$$

Rewritten on an other form

$$\text{pressure} = \text{temperature density constant}$$

This means that one of the variables pressure, temperature and density can be expressed in the other two.
The function that creates the interpreter is called
`make-vapour-cut-interpreter`.
Example:

```
> (define p (make-subcut 'p "pressure"))
p
> p
(p "pressure")
> (define t (make-subcut 't "temperature"))
t
> (define d (make-subcut 'd "density"))
d
> (define in (make-cut 'in (list p t) "vapour-pipe1"))
in
> in
(in ((p "pressure") (t "temperature")) "vapour-pipe1")
> (define out (make-cut 'out (list d p) "vapour-pipe2"))
out
> out
(out ((d "density") (p "pressure")) "vapour-pipe2")
> (define out2 (make-cut 'out2 (list d t) "vapour-pipe3"))
out2
> (define inouttrans (make-vapour-cut-interpreter in out))
inouttrans
> inouttrans
(t ((p 1) (d -1) (r -1)))
> (define outintrans (make-vapour-cut-interpreter out in))
outintrans
> outintrans
(d ((p 1) (t -1) (r -1)))
```

3.4 Implementation of CutSystem

The programming style in this part of the project are quite different compared with *4s*. `CutSystem` is really a *layered design* with a lot of small general functions . No function are bigger then six rows of Scheme code, except for one larger function.
The price is of cause a lot of function calls, but that is cheap compared with what you gain with readable code. Of cause there must be a compromise between small readable function and efficiency.

3.5 Conclusion

The cut interpreter developed in `CutSystem` is very hard to generalize.
In this implementation , the transformations are wired in the code and to change them one have to change every transformation rules.
To use production system for a problem like this is perhaps a solution. A production system is easier to generalize and it would be easy to add new transformation rules.

4. Summary

As already discussed, is Scheme a good programming language for solving problems on symbolic form, like the project discussed here. There is of cause other symbolic manipulating languages that also are well suited for problems like this.
The project shows the power and how easy it is to write quite advanced programs in languages like Scheme. To test new ideas and explore its oppertuneties is easy.
One major drawback with Scheme is the lack of graphic tools. Therefore is Scheme not a language suited for future developments.
An other aspect is the programming style. As mentioned above can one use object oriented programming style with great success. The problem is well suited for *object oriented programming.*
Including the need of graphic based interface to a program like this, *Smalltalk* seems to be a good tool. It supports both object oriented programming and nice graphics. The programming environment in Smalltalk is also of great advantage.
Other programming environment of interest is KEE. KEE supportes both object oriented programming and graphics. Production system environment is also implemented in KEE.

Appendix I: the 4s code.

```
(define (insert-list superlist identifier sublist)
 (cond ((eq? (assoc identifier (list superlist)) #f) superlist)
        (else
         (set! superlist
               (append superlist (list sublist))))))

(define (add-to-list holesuperlist identifier sublist)
  (cond ((eq? holesuperlist ()) ())
  (else
    (set! holesuperlist
        (append (list
                 (insert-list (car holesuperlist) identifier sublist))
                 (add-to-list (cdr holesuperlist) identifier sublist))
        ))))

(define (unique-test identifiertype identifier system)
  (cond ((eq?
              (assoc identifier
                     (cdr (assoc identifiertype (cadr system)))) ())
              #t)
        (else #f)))

(define (make-system system-name)
  (list system-name '()))

(define (create-input system input)
  (cond ((eq? (assoc 'input  (cadr system)) ())
           (set! system
                 (list (car system)
                       (append (cadr system)
                               (list (list 'input input))))))
         (else
           (cond ((eq? (unique-test 'input (car input) system) #t)
                       (set! system
                          (list (car system)
                                (add-to-list (cadr system)
                                             'input input))))
                 (else (display "Not a unique name on input")
                       (newline)))))
  system)

(define (create-output system output)
   (cond ((eq? (assoc 'output  (cadr system)) ())
           (set! system
                 (list (car system)
                       (append (cadr system)
                               (list (list 'output output))))))
         (else
           (set! system
                 (list (car system)
                       (add-to-list (cadr system) 'output output))))))

(define (make-subsystem subsystem supersystem)
   (cond ((eq? (assoc 'subsystem  (cadr supersystem)) ())
           (set! supersystem
                 (list (car supersystem)
                       (append (cadr supersystem)
                               (list (list 'subsystem subsystem))))))
         (else
           (set! supersystem
                 (list (car supersystem)
                       (add-to-list (cadr supersystem)
                                    'subsystem subsystem))))))
```

```
(define (make-siso-system system-name)
  (let ((system (make-system system-name)))
    (set! system (create-input system '(u (u) "nt")))
    (set! system (create-output system '(y (y) "nt")))
    system))

(define (system-copy system-name system)
  (list system-name (cadr system)))

(define (make-connection supersystem inputidentifier outputidentifier)
 (set! connected-input
        (append
        (assoc (car inputidentifier)
               (cdr (assoc 'input
                           (cadr (assoc (caadr inputidentifier)
                                        (cdr (assoc 'subsystem
                                              (cadr supersystem))))))))
        (cdr inputidentifier)))
   (set! connected-output
        (append
        (assoc (car outputidentifier)
               (cdr (assoc 'output
                           (cadr (assoc (caadr outputidentifier)
                                        (cdr (assoc 'subsystem
                                              (cadr supersystem))))))))
        (cdr outputidentifier)))
   (cond ((equal? (caddr connected-input)
                  (caddr connected-output))
          (set! connection
             (list connected-input connected-output)))
        (else (set! connection
                    '())))
   connection)

(define (connect supersystem inputidentifier outputidentifier)
        (set! connection
         (make-connection supersystem inputidentifier outputidentifier))
        (cond ((eq? connection '())
               (display "Not allowed connection")
               (newline)
               (display "Wrong typing of cut")
               (newline))
              (else
               (cond ((eq? (assoc 'connection (cadr supersystem)) ())
                      (set! supersystem
                          (list (car supersystem)
                             (append (cadr supersystem)
                                     (list (list 'connection connection))))))
                     (else
                        (set! supersystem
                           (list (car supersystem)
                              (add-to-list (cadr supersystem)
                                           'connection connection)))))))
   supersystem)

(define (make-subcut identifier identifiertype)
  (list identifier identifiertype))

(define (make-cut cut-identifier cut-list cutidentifiertype)
  (list cut-identifier cut-list cutidentifiertype))

(define (cut-copy newcut-name oldcut)
  (append (list newcut-name) (cdr oldcut)))
```

```
;;;;;;;;;;;;;;;;EXEMPEL siso;;;;;;;;;;;;;;;

(define s1 (make-siso-system 's1))
(define s2 (system-copy 's2 s1))

(define ss1 (make-system 'ss1))
(define ss1 (make-subsystem s1 ss1))
(define ss1 (make-subsystem s2 ss1))
(define ss1 (connect ss1 '(u (s1)) '(y (s2))))
(define ss1 (connect ss1 '(u (s2)) '(y (s1))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define flowcut (make-subcut 'q "flow"))
(define compcut (make-subcut 'x "composition"))
(define inflow (make-cut 'inflow (list flowcut compcut) "pipe"))
(define outflow (cut-copy 'outflow inflow))

(define tank (make-system 'tank))
(define tank (create-input tank inflow))
(define tank (create-output tank outflow))
(define tank1 (system-copy 'tank1 tank))
(define tank2 (system-copy 'tank2 tank))

(define tank-system (make-system 'tank-system))
(define tank-system (make-subsystem tank1 tank-system ))
(define tank-system (make-subsystem tank2 tank-system ))

(define tank-system (connect tank-system
                             '(inflow (tank2)) '(outflow (tank1))))
```

Appendix II: the CutSystem code.

```
(define (make-subcut identifier identifiertype)
  (list identifier identifiertype))

(define (make-cut cut-identifier cut-list cutidentifiertype)
  (list cut-identifier cut-list cutidentifiertype))

(define (make-cut-copy newcut-name oldcut)
  (append (list newcut-name) (cdr oldcut)))

(define (make-cut-type-list cutidentifier)
  (let ((typelist
         (map (lambda (x) (cadr x)) (cadr cutidentifier))))
  typelist))

(define (make-cut-type-sorted-list newcutname)
  (list newcutname '()))

(define (insert-list superlist identifier sublist)
 (cond ((eq? (assoc identifier (list superlist)) #f) superlist)
        (else (append superlist (list sublist))))))

(define (add-to-list holesuperlist identifier sublist)
  (cond ((eq? holesuperlist ()) holesuperlist)
        (else
          (append (list
                    (insert-list (car holesuperlist) identifier sublist))
                    (add-to-list (cdr holesuperlist) identifier sublist)))))

(define (identifier-in-list? identifier list)
  (cond ((eq? (assoc identifier list) ()) #f)
        (else #t)))

(define (create-identifier-list newcuttypelist identifier cuttype)
  (list (car newcuttypelist)
        (append (cadr newcuttypelist)
                (list (list identifier cuttype)))))

(define (add-identifier-list newcuttypelist identifier cuttype)
  (list (car newcuttypelist)
        (add-to-list (cadr newcuttypelist)
                     identifier
                     cuttype)))

(define (create-cut-type-member newcuttypelist cuttype)
  (cond ((identifier-in-list? 'member (cadr newcuttypelist))
         (add-identifier-list newcuttypelist 'member cuttype))
        (else
         (create-identifier-list newcuttypelist 'member cuttype))))

(define (create-cut-type-nonmember newcuttypelist cuttype)
  (cond ((identifier-in-list? 'nonmember (cadr newcuttypelist))
         (add-identifier-list newcuttypelist 'nonmember cuttype))
        (else
         (create-identifier-list newcuttypelist 'nonmember cuttype))))

(define (member-of-list? element list)
  (cond ((eq? (member element list) ()) #f)
        (else #t)))

(define (insert-cut-type cuttypesortedlist cuttype referencelist)
  (cond ((member-of-list? cuttype referencelist)
         (create-cut-type-member cuttypesortedlist cuttype))
        (else
         (create-cut-type-nonmember cuttypesortedlist cuttype))))
```

```
(define (empty-list? list)
  (null? list))

(define (sort-cut-type-list-recurse sortlist cuttypelist referencelist)
  (cond ((empty-list? cuttypelist)
         sortlist)
        (else
         (insert-cut-type (sort-cut-type-list-recurse sortlist
                                                      (cdr cuttypelist)
                                                      referencelist)
                          (car cuttypelist)
                          referencelist))))

(define (sort-cut-type-list cuttypelist referencelist)
  (let ((sortedcuttypelist (make-cut-type-sorted-list 'sortedcuttypelist)))
    (sort-cut-type-list-recurse sortedcuttypelist cuttypelist referencelist)))

(define (common-cut-type? cuttypeidentifier cuttypelist)
  (cond ((null? (member cuttypeidentifier
                        (assoc 'member (cadr cuttypelist)))) #f)
        (else #t)))

(define (uncommon-cut-type? cuttypeidentifier cuttypelist)
  (cond ((null? (member cuttypeidentifier
                        (assoc 'nonmember (cadr cuttypelist)))) #f)
        (else #t)))

(define (pressureinterpreter)
  (list 'p (list '(t 1) '(d 1) '(R 1))))

(define (temperatureinterpreter)
  (list 't (list '(p 1) '(d -1) '(R -1))))

(define (densityinterpreter)
  (list 'd (list '(p 1) '(t -1) '(R -1))))

(define (make-cut-interpreter output)
  (cond ((equal? "pressure" output) (pressureinterpreter))
        ((equal? "temperature" output) (temperatureinterpreter))
        ((equal? "density" output) (densityinterpreter))))

(define (make-vapour-cut-interpreter firstcut secondcut)
  (let ((firstcuttypelist (sort-cut-type-list (make-cut-type-list firstcut)
                                              (make-cut-type-list secondcut)))
        (secondcuttypelist (sort-cut-type-list (make-cut-type-list secondcut)
                                               (make-cut-type-list firstcut))))
    (cond ((common-cut-type? "pressure" firstcuttypelist)
           (cond ((uncommon-cut-type? "temperature" firstcuttypelist)
                  (make-cut-interpreter "temperature"))
                 (else
                  (make-cut-interpreter "density"))))
          ((common-cut-type? "temperature" firstcuttypelist)
           (cond ((uncommon-cut-type? "pressure" firstcuttypelist)
                  (make-cut-interpreter "pressure"))
                 (else
                  (make-cut-interpreter "density"))))
          ((common-cut-type? "density" firstcuttypelist)
           (cond ((uncommon-cut-type? "temperature" firstcuttypelist)
                  (make-cut-interpreter "temperature"))
                 (else
                  (make-cut-interpreter "pressure"))))
          (else
           (display "No cut interpreter created")
```

```
           (newline)))))

;;;;;;;;;;;;;;;;;;

(define p (make-subcut 'p "pressure"))
(define t (make-subcut 't "temperature"))
(define d (make-subcut 'd "density"))
(define in (make-cut 'in (list p t) "vapour-pipe1"))
(define out (make-cut 'out (list d p) "vapour-pipe2"))
(define out2 (make-cut 'out2 (list d t) "vapour-pipe3"))

(define intype (make-cut-type-list in))
(define outtype (make-cut-type-list out))

(define intypesort (sort-cut-type-list intype outtype))
(define outtypesort (sort-cut-type-list outtype intype))

(define inouttrans (make-vapour-cut-interpreter in out))
(define inout2trans (make-vapour-cut-interpreter in out2))
(define outintrans (make-vapour-cut-interpreter out in))
;;;;;;;;;;;;;;;;
```