ExSpect Language Tutorial

written for ExSpect 6.0 and later

M.Voorhoeve
Eindhoven University of Technology

March 19, 1998

1 Introduction

The ExSpect language and tool can be used in two ways. One way is to combine predefined building blocks (processors and subsystems) to specify a large system. This way can be learnt from doing the tutorial. The building blocks are connected to one another via *channels* and *stores*, that transfer data between the blocks. However, not all systems can be built in this way. It may be necessary to adapt building blocks or even to create them from scratch.

For this use of ExSpect - as a high-level programming language - this tutorial is destined. The part that deals with these aspects is called the functional part of ExSpect. It resembles typed functional programming languages, whence this qualification. The most striking difference of the ExSpect language compared to ordinary (third-generation) programming languages are the absence of the sequencing (semicolon: '`;`') operator. In e.g. the C language, interchanging the values of variable `x` and `y` usually is performed by introducing an auxiliary variable e.g. `help`, and executing the following statements.

```
help = x; x = y; y = help;
```

In ExSpect, this is done by the following simultaneous assignment.

```
x <- y, y <- x
```

In a simultaneous assignment, the variables (called stores) retain their value until the complete assignment has been executed. The order in which they are given is of no effect.

Another difference is that ExSpect allows assignments of very complex datatypes in one stroke; for instance the assignment

```
s <- [x:  stretch(0,100)| x*x]
```

fills the store `s` with a table giving the squares for the numbers 0 up to 100.

The functional part of ExSpect is about defining stores, channels and processors. Stores and channels are defined by attaching types to them. A type represents a set of values. Processors are defined by attaching value expressions or terms to them. A term represents a value, that may depend upon some parameters. We give an example: the system `FINANCE` depicted in figure 1.

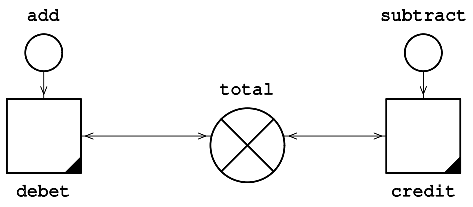

Figure 1: Finance system

The channels `add`, `subtract` and the store `total` are all typed with the basic type `num` (numeral). Non-basic types are constructed from simpler types by means of type constructors, e.g. `num><num` (numeral pair) or `[name:str,salary:num]` (a record or tuple). A *type definition* consists of an identifier (the defined type) and a type (its definition). The identifier can then be used in types instead of its definition. *Type expressions* are types with variables. They can be used in *signatures* to define *functions* and *processors*.

The processors in the figure are both derived from the processor `book` with the following definition. This definition contains *parameters* that are bound by *installing* the processor.

Connections:

Name	Kind	Type
`amount`	`in`	`num`
`total`	`store`	`num`

Pre: `amount > 0`

Val/Fun parameters:

Name	Kind	Type	Value
`kind`	`val`	`str`	

Definition

```
if kind='credit' then total <- total - amount
                  else total <- total + amount fi
```

The installations are given in the following tables.

Name: `debet`

Parameters:

Name	Kind	Value
`amount`	`in`	`add`
`total`	`store`	`total`
`kind`	`val`	`'debet'`

Name: `credit`

Parameters:

Name	Kind	Value
`amount`	`in`	`subtract`
`total`	`store`	`total`
`kind`	`val`	`'credit'`

In the example, the values `'debet'`, `'credit'`, the conditions `amount>0`, `kind='credit'` and the assigned values `total - amount`, `total + amount` all are terms.

Terms can be either atomic (constants), a variable or constructed by means of functions or *value constructors* from simpler terms. In the example, `'debet'`, `credit'`, `0`, are atomic terms, `kind`, `amount` and `total` are variables, whereas `>`, `=`, `-`, `+` are function symbols representing functions. Another way to make terms is by *quantors*, which we will discuss later.

We shall first deal with types and values and then turn to terms and functions. We then explain in more detail how processors and systems are defined and installed. We conclude by explaining how to create and use modules.

2 Types and values

Types represent sets of values. Basic types are `bool`, `str`, `num` and `real`. They contain atomic values called *constants*. Non-basic or constructed types contain basic types, one or more *type constructors* plus brackets '`(...)`' to define the order in which the type constructors are applied. Their values are represented by constructions containing constants and special function symbols called construction symbols. We first treat basic types and their values, then type constructors and constructed values. We conclude by treating type definitions.

2.1 Basic types and constants

The basic types are `bool`, `str`, `num` and `real`.

The type `bool` contains two values: the boolean constants `false` and `true`.

The type `str` contains all printable ASCII strings, represented between quotes. `'#%AC/DC!!'` is a constant of type `str`. To represent a quote in a string constant, it must be doubled. So `'''a'''` represents the string '`a`'. The empty string is `''`, which is also a valid constant. ASCII strings containing nonprintable characters, e.g. newlines are not in this type.

The type `num` contains the rational numbers, i.e. natural numbers (`0`,`1`,`2`...), negative integers (`-1`,`-2`...), and their quotients, like `5/17` or `-191/27`. The rational numbers have no upper or lower bound; ExSpect can handle 1000-digit numbers and their quotients. Of course, manipulations with large numbers will be rather time consuming.

The type `real` is a numeric type like `num`. To distinguish reals from nums, reals must have a single decimal dot somewhere. Note that e.g. `2` and `2.` are different constants, although they might mean the same to a user. So `-2.` and `3.14159` are reals, whereas `-2` and `314159/100000` are nums. Manipulation with reals is faster that with nums, but at a price: reals are approximated. So `2.5+2.5=5.` need not necessarily hold, the result might be e.g. `4.9999999999761`. The approximation made depends on the hardware platform used. There is also a maximum real; exceeding this maximum gives rise to errors similar to division by zero.

So we have our basic types. There is a fifth basic type `void`, which has no constants. It is used in type constructions to denote some "degenerate" values.

2.2 Constructed types and their values

To construct new types, we use our type constructors `$`, `><`, `->`, `*` and the record constructor. Basic types and type constructors give types that represent sets of constructed (non-atomic) values. These values are made by means of value constructors; to each type constructor corresponds a value constructor. Value constructors can be applied to constants, to constructed values and even to terms in general, that we treat later. We assume that A, B, A', ... are types.

The type `$`A contains finite sets of values from A. These sets are represented by enclosing its elements in braces. So `$num` contains e.g. the values `{}` (the empty set), `{1}`, `{-2,5/3}`, `{3599/333}` and so on. The value constructor is here `{_,...,_}`. In forming a set from elements, the order is neglected and duplicates are removed. So `{1,2,3}` and `{2,3,2,1}` are the same. Note that the empty set `{}` belongs to `$`A for any type A. This fact is reflected by giving it the degenerate type `$void`. It is not allowed to put values from different types within the same set, so e.g. `{1,-2.0}` is an illegal value. Note that `bool` and `{false,true}` have the same elements, but are nevertheless different: one is a type, the other a value.

The type `*`A contains finite lists (sequences) of values from A. These lists are represented by enclosing its elements between `<|` and `|>`. So e.g. `<|4,0,0,1,4,5|>` is a list containing six numerals; it is of type `*num`. There is an empty list, denoted `<||>`, having the degenerate type `*void`. Unlike sets, the order is maintained and no duplicates are removed. So `<|1,2,3|>`, `<|2,3,1|>` and `<|2,2,3,1|>` are all different.

The type A`><`B contains value pairs, the first from A the second from B. They are represented as `<<`a`,`b`>>`. Here `<<_,_>>` is the value constructor. `<<5,3.14>>` is a value from `num><real` and `<<'',{'','a'}>>` is from `str><$str`. The `$` and `*` operators have priority over `><` and `->`, so a "set of pairs" type needs brackets, e.g. `$(num><str)`.

The type A`->`B contains mappings: finite sets of value pairs. The values of this type can be constructed by combining the above two constructions. A restriction is imposed, though: the first components (from A) have to be different. So `{<<1,1>>,<<2,1>>}` is a value of `num->num`, but `{<<1,1>>,<<1,2>>}` is not. Note that the second value does have type `$(num><num)`. The first value also has type `$(num><num)`, since any value of `num->num` is also a value of `$(num><num)`. We say that `num->num` is *stronger* than `$(num><num)`.

The mapping operator can be used to model arrays: `num->`A can be used to represent an A array, with elements `{<<1,`a_1`>>,<<2,`a_2`>>, ...}`. This an alternative for the list (`*`A) operator. When sequential access is only needed, the list approach is preferred, but for random access the type `num->`A is more profitable. For combinations of the `><` and `->` operator, it is advised always to use brackets to indicate the order.

Record types are constructed by means of attribute identifiers. If l_1, l_2, ... are identifiers, `[`l_1`:`A`, `l_2`:`A'`,...]` is a record type. Its values are e.g. `[`l_1`:`a`, `l_2`:`a'`,...]`. The type and value constructor thus look the same in this case. This is like the record construction in programming languages. The order of the attributes is not important, e.g. `[a:num,b:str]` and `[b:str,a:num]` denote the same (record) type and the values `[a:4,b:'a']` and `[b:'a',a:4]` denote the same record within this type.

2.3 Type definitions

Type constructions can be given a name: a type definition. Names must obey the syntax for identifiers: they must begin with an alphabetic character and may not contain spaces. A defined type can be used in type constructions, thus (maybe) creating more complex defined types. However, it must be possible to expand every type expression, replacing the name by the construction. So-called *recursive* types are not allowed. We show an example, where a client file is defined. The defined types are address and client (in that order), with the following constructions.

Name	Definition
`address`	`[street: str, city: str, zip: str]`
`client`	`[nr: num, name: str, addr: address]`

The type `$client` can be used to model e.g. a file containing client data (if clients are unique). A sequential file containing the same could be modeled by the type `*client`.

3 Functions and terms

Terms are atoms (constants), variables or constructed out of these by means of functions and quantors. We first concentrate on functions. Functions transform input values into a result value. Function applications are terms involving the function symbol and simpler argument terms. For instance the term `sin(3.14)` signifies the application of the function `sin` (sinus) onto the term `3.14`; the *evaluation* of this term results in a term of type `real` close to `0.0`. The term `1+2` signifies the application of `add` (addition) onto `1` and `2`; the evaluation result is `3`. The term `a+2` signifies the application of `add` (addition) onto `a` and `2`; the evaluation result depends upon the value that is substituted for `a`.

Which input values of a function are accepted and which result value is given, is determined by the function's *signature*. The signature of `sin` is `real` for input and `real` for result, of `add` it is `num,num` for input and `num` for result We represent these signatures by `real` → `real` and `num,num` → `num` respectively.

ExSpect has a large number of standard functions. From them, new functions can be defined. We first describe some standard functions and their applications. Then we discuss quantors. Finally we describe how new functions can be defined.

3.1 Simple functions

A simple function has a single signature containing only types (without type variables). In Figure 2, we give a selection of simple functions with their signatures.

name	signature	example		
`not`	`bool` → `bool`	`not(true)`	=	`false`
`and`	`bool,bool` → `bool`	`false and true`	=	`false`
`or`	`bool,bool` → `bool`	`false or true`	=	`true`
`div`	`num,num` → `num`	`7/4 div 1/3`	=	`5`
`mod`	`num,num` → `num`	`9/4 mod 2`	=	`1/4`
`sin`	`real` → `real`	`sin(3.14)`	∼	`0.0016`
`cos`	`real` → `real`	`cos(3.14)`	∼	`-1`
`log`	`real` → `real`	`ln(3.14)`	∼	`1.144`
`exp`	`real` → `real`	`exp(1.0)`	∼	`2.718`

Figure 2: Simple functions

Reading the table should present little problems. The third line e.g. defines the simple function `or`, with two boolean parameters and a boolean result. From the example can be deduced that the function can be used in infix mode, i.e. the function name or symbol between the parameter terms. Its effect is the disjunction of its parameters. Elsewhere in this manual the exact nature of these functions is explained.

Every function with two parameters can be used in infix mode. However, in nested function applications, like `(a and b) or c` one is advised to use brackets for priority.

3.2 Overloading

The same function symbol and name is sometimes used for several simple functions. An example is formed by the arithmetical functions. The arithmetical manipulation of `real`s and `num`s is different. Yet, when specifying an addition, one wants to use the symbol `+` irrespective of the kind of numbers added. This is done by *overloading*: attaching more signatures to the same function name, as shown in Figure 3.

Note that the name of the above functions differs from the symbol used. Also note the third definition of addition that manipulates sets of numerals. A more extensive list of numeric functions can be found elsewhere in the manual.

The priority in nested arithmetical terms such as a+b*c is like one is used to. Note the extra brackets that are needed in the term `(4/5)/(2/3)`. We encountered the division symbol earlier in the values belonging to type `num`.

3.3 Polymorphy

Polomorphy can be seen as infinite overloading. For instance set union does not regard the type of the contained elements. However, the union of sets of different type is illegal: the result cannot be typed. The signatures of the union thus contain `$num,$num` → `$num`, `$str,$str` → `$str`, but

name	signature	examples		
add	num,num → num	4/5 + 2/3	=	22/15
	real,real → real	0.8 + 0.67	∼	1.47
	num,$num → $num	4/5 + {0,2/3}	=	{4/5,22/15}
sub	num,num → num	4/5 - 2/3	=	2/15
	real,real → real	0.8 - 0.67	∼	0.13
mult	num,num → num	4/5 * 2/3	=	8/15
	real,real → real	0.8 * 0.67	∼	0.536
rdiv	num,num → num	(4/5)/(2/3)	=	6/5
	real,real → real	0.8 / 0.67	∼	1.194
gt	num,num → bool	5 > 7	=	false
gt	real,real → bool	5. > 4.	=	true

Figure 3: Overloaded functions

not $num,$str → ??. Hence $\$T,\$T \rightarrow \$T$ is a signature for all types T. This is the meaning of the signature $T,$T → $T, where T is a *type variable*. Functions with signatures containing type variables are called *polymorphic*.

In Figure 4, we give some standard polymorphic functions. The type variables used are T and S. The pick function takes one element from a set. The rest function gives the set with the picked element removed from it. Even polymorphic functions can be overloaded, as in the last examples. The functions on the string datatype partially coincide with the list functions. This makes sense, since a string can be thought of as a list of characters.

3.4 Other functions

The set, list, record and pair constructors can be applied to terms like in the following examples.

```
{1+1,3-1,5} = {2,5},
[name: 'J.' cat ' Doe', sal: 10*10*30] = [name: 'J. Doe', sal: 3000],
<<3 ins <|2,7-1|>, 1.1*1.1>> ∼ << <|3,2,6|>, 1.21>>.
```

The record projection (symbol @) and update (upd) functions are also treated here. The following examples illustrate their use; more details can be found in the reference manual.

```
[a:5, b:'q']@a = 5,
[a:5, b:'q'] upd [b:'r', c:true] = [a:5, b:'r', c:true]
```

Most of these functions do not posess a signature, although the allowed parameter types and the way they affect the result type are completely determined. The set constructor accepts one or more parameters of type T and yields a result of type $T. The record constructors [l_1:_, l_2:_ ...] accept terms of types T_1, T_2 ... and yield a term of type [l_1:T_1, l_2:T_2...]. The pair constructor is the only one with a normal signature: T,S → T><S.

The projection functions _@*l* accept a parameter of any type of the form [... l:T ...] and yield a result of type T. The record update function accepts two parameters of type [l_1:T_1...] and

name	signature	examples		
eq	T,T → bool	'a' = 'a'	=	true
		{1} = {}	=	false
cond	bool,T,T → T	if true then 6 else 7 fi	=	6
elt	T,$T → bool	2 elt {1,3,5}	=	false
pick	$T → T	pick({0})	=	0
rest	$T → $T	rest({0})	=	{}
union	$T,$T → $T	{1,2} union {3}	=	{1,2,3}
pi1	T><S → T	pi1(<<3,'a'>>)	=	3
pi2	T><S → S	pi2(<<3,'a'>>)	=	'a'
dom	T->S → $T	dom({<<1,5>>,<<2,7>>})	=	{1,2}
rng	T->S → $T	rng({<<1,5>>,<<2,7>>})	=	{5,7}
apply	T->S,T → S	{<<1,5>>,<<2,7>>}.2	=	7
	T->S,$T → $S	{<<1,5>>,<<2,7>>}.{1,2}	=	{5,7}
ins	T,*T → *T	4 ins <|4,2|>	=	<|4,4,2|>
ins	T,$T → $T	4 ins {2,4}	=	{2,4}
		4 ins {3,2}	=	{2,3,4}
head	*T → T	head(<|4,4,2|>)	=	4
	str → str	head('Gee')	=	'G'
tail	*T → *T	tail(<|4,4,2|>)	=	<|4,2|>
	str → str	tail('Gee')	=	'ee'
cat	*T,*T → *T	<|4,4|> cat <|2|>	=	<|4,4,2|>
	str,str → str	'Oh' cat ' boy'	=	'Oh boy'

Figure 4: Polymorphic functions

`[`$\texttt{m}_1$`:`$\texttt{S}_1\ldots$`]` respectively, such that common labels are matched to the same types, i.e. if $\texttt{l}_1 = \texttt{m}_1$ then $\texttt{T}_1 = \texttt{S}_1$. The result type is the record type that joins the parameter types.

3.5 Erroneous function applications

In writing terms, three kinds of errors may occur. The first ones are syntax errors due to ill-matched brackets, misspellings and the like. The second are errors against the signature, like `{1,2} union {5.0}` or `5 = '5'`. Both kinds of errors are discovered during translation and reported by a "syntax error" or "type unknown" error message. The third are runtime errors, which are detected during execution only. Examples are given in the table below.

runtime error	explanation
`1/0`	division by zero
`head('')`	head/tail of empty string
`pick({})`	pick on empty set
`tail(<||>)`	head/tail of empty list
`{<<1,2>>,<<3,4>>}.2`	invalid apply argument

In these cases the "non-value" `abort` is the evaluation result. Of course nobody will write the above erroneous terms, but they may occur while evaluating a term involving variables. In applications of the `cond` function, the branch that is not chosen is not evaluated and hence might have aborted. So

```
 if a=0 then 0 else b/a fi
```

and

```
 if a elt dom(M) then M.a else 0 fi
```

are not aborting terms. The same holds for the `and` and `or` functions; their evaluation depends on the order of the parameters (since they use the `cond` function). The second term below may abort whereas the first won't, like in many programming languages. `a elt dom(M) and M.a=0`

```
 M.a=0 and a elt dom(M)
```

3.6 Implicit mapping construction and quantors

We have constructed mappings by explicit enumeration. Mappings can also be implicitly constructed. The syntax is `[`x`:`A`|`H_x`]`. Here, x must be an identifier, A a term of type `$`D or D`->`E for certain type constructions D, E and H_x a term that may contain x as parameter. If A is of type `$`B, the parameter x is understood to have type B. The result type is B`->`C, where C is the type of H_x. The explicit value is obtained by constructing the set of pairs `<<`a`,`b`>>`, where a subsequently takes all values from A and b is obtained by replacing x by a in the evaluation of H_x. Some examples of this construction are given below.

mapping	type	explicit value
`[y:{'b','d'}|tail(y)]`	`str->str`	`{<<'b',''>>,<<'d',''>>}`
`[x:{1,3}|[a:x-1]]`	`num->[a:num]`	`{<<1,[a:0]>>,<<3,[a:2]>>}`
`[z:{{},{0}}|0 elt z]`	`$num->bool`	`{<<{},false>>,<<{0},true>>}`

The implicit mapping construction can be combined with the `rng` function to construct sets in an implicit way. The term `rng[`x`:`A`|`E_x`]` is equivalent to the mathematical notation for sets $\{E_x \mid x \in A\}$. The above examples combined with `rng` give the following results.

term	type	value
`rng[y:{'a','b','d'}|tail(y)]`	`$str`	`{''}`
`rng[x:{1,3}|[a:x-1]]`	`$[a:num]`	`{[a:0],[a:2]}`
`rng[z:{{},{0}}|0 elt z]`	`$bool`	`{false,true}`

There are other functions (quantors) that combine well with implicit mappings. We give a table.

name	signature	example		
`all`	`T->bool` $\rightarrow$ `bool`	`all[x:{1,2,3}|x>0]`	=	`true`
`any`	`T->bool` $\rightarrow$ `bool`	`any[x:{1,2,3}|x>3]`	=	`false`
`set`	`T->bool` $\rightarrow$ `$T`	`set[x:{1,2,3}|x>1]`	=	`{2,3}`
`sum`	`T->num` $\rightarrow$ `num`	`sum[x:{1,2,3}|x+1]`	=	`9`

The term `set[`x`:`A`|`E_x`]` can also be written `$[`x`:`A`|`E_x`]` because of the association of "set" with the `$` symbol. We tabulate some terms with their mathematical counterpart. Here x is a variable, E_x a term containing x, P_x a predicate containing x, N_x a numerical term containing x, S_x a set term containing x and A a (finite) set. We added the `max` and `min` quantors that have the same signature as `sum`. Note the last element where two quantors are combined.

ExSpect	math
`[`x`:`A`|`E_x`]`	$\{(x, E_x) \mid x \in A\}$
`rng[`x`:`A`|`E_x`]`	$\{E_x \mid x \in A\}$
`all[`x`:`A`|`P_x`]`	$\forall x \in A : P_x$
`any[`x`:`A`|`P_x`]`	$\exists x \in A : P_x$
`set[`x`:`A`|`P_x`]`	$\{x \mid x \in A \wedge P_x\}$
`sum[`x`:`A`|`N_x`]`	$\Sigma_{x \in A} N_x$
`max[`x`:`A`|`N_x`]`	$\mathrm{MAX}_{x \in A} N_x$
`min[`x`:`A`|`N_x`]`	$\mathrm{MIN}_{x \in A} N_x$
`rng[`x`:set[`x`:`A`|`P_x`]|`E_x`]`	$\{E_x \mid x \in A \wedge P_x\}$

3.7 Function definitions

Like type definitions, function definitions are made by attaching a name to a term. A function definition must be accompanied by its signature, so its parameter variables and their types and its result type must accompany the definition term (the "body" of the definition). Since the body is so important, definition bodies are called "Definition" in the ExSpect windows. A body term without parameters gives a value definition. In the following examples simple functions are defined.

Name:	`half`
Parameters:	
Resulttype:	`num`
Definition:	`1/2`
Name:	`triangle`
Parameters:	`x:num`
Resulttype:	`num`
Definition:	`x*(x-1)/2`
Name:	`headstogether`
Parameters:	`x:str`, `y:str`
Resulttype:	`str`
Definition:	`head(x) cat head(y)`
Name:	`mult`
Parameters:	`x:$num`, `y:num`
Resulttype:	`$num`
Definition:	`rng[t:x|t*y]`

The last function in fact is an extra overloading of the `mult` function that we saw earlier. The choice of the name `mult` means that we can use the infix symbol `*`; for instance in the term `{5,4,3} * 7` meaning `mult({5,4,3}, 7)` and yielding `{35,28,21}`.

Polymorphic functions are defined by using type variables, like in the following examples: the set intersection and delete functions, followed by mapping update.

Name:	`isect`
Parameters:	`x:$T`, `y:$T`
Resulttype:	`$T`
Definition:	`set[t:x|t elt y]`
Name:	`del`
Parameters:	`x:T,y:$T`
Resulttype:	`$T`
Definition:	`set[t:y|not(t=x)]`
Name:	`upd`
Parameters:	`x:T->S,y:T->S`
Resulttype:	`T->S`
Definition:	`[t:dom(x) union dom(y)| if t elt dom(y) then y.t else x.t fi]`

The body of the mapping update function `upd` above could also have been

```
  y union set[t:x| not(pi1(t) elt dom(y))].
```

This example needs some study, since many functions are combined. An example of the mapping update: `{<<1,2>>,<<3,4>>} upd {<<3,5>>,<<4,7>>}` yields `{<<1,2>>,<<3,5>>,<<4,7>>}`.

The update of a mapping at a single point is used frequently; an extra (overloaded) definition has been added as follows.

Name:	`upd`
Parameters:	`x:T->S,y:T,z:S`
Resulttype:	`T->S`
Definition:	`x upd {<<y,z>>}`

The above definition is not accepted by the translator, since it cannot perceive that the term `{<<y,z>>}` is a mapping. Instead, it is typed with `$(T><S)` and a function `upd` with signature `T->S` $\rightarrow$ `$(T><S)` is not found. So, an error message "type of upd unknown" is issued. This problem can be solved by a *type cast*: the type is reinforced to `T->S` by applying the "dummy" function `tomap` with the following definition.

Name:	`tomap`
Parameters:	`x:$(T><S)`
Resulttype:	`T->S`
Definition:	`x`

The refused body of `upd` is replaced by `x upd tomap({<<y,z>>})` and this is accepted. The ultimate type cast is the function `tovoid` reinforcing any type to the "strongest" type `void`. In this way, the type checking of the translator is bypassed!

3.8 Recursive definitions

Unlike type definitions, function definitions may contain the function that is being defined. An example of such a *recursive* definition is the size of a set.

Name:	`size`
Parameters:	`x:$T`
Resulttype:	`num`
Definition:	`if x={} then 0 else 1+size(rest(x)) fi`

Recursive definitions must be used with care, since they may lead to nontermination. We give an example of a "dangerous" recursive definition.

Name:	`div`
Parameters:	`x:num`, `y:num`
Resulttype:	`num`
Definition:	`if x<y then x else 1+div(x-y,y) fi`

Evaluating the function `2 div -1` will not terminate. As an exercise, try to improve upon this definition.

In the majority of cases, recursion can be replaced by quantor definitions. The definition in `size` could have been e.g. `sum[t:x|1]`. Consider the function that extracts the adresses from a set of client records.

Name:	`addresses`
Parameters:	`x:$[name:str, addr:str]`
Resulttype:	`$str`

Definition:	`if x={} then {}`
	`else {pick(x)@addr} union addresses(rest(x)) fi`

Much more comprehensible, concise and efficient is the definition `rng[t:x|t@addr]`. Many "standard" functions encountered earlier are in fact defined by means of a small set of truly atomic functions. The `dom` and `apply` functions can be defined as follows.

Name:	`dom`
Parameters:	`x:T->S`
Resulttype:	`$T`
Definition:	`rng[t:x|pi1(t)]`
Name:	`apply`
Parameters:	`x:T->S`, `y:T`
Resulttype:	`S`
Definition:	`pick(rng(tomap(set[s:x|pi1(s)=y])))`

The function `rng(x)` must be defined recursively, its definition being

```
  if x = {} then {} else {pi2(pick(x))} union rng(tomap(rest(x))) fi
```

3.9 Types and sets; functions and mappings

ExSpect types and sets can mean the same, but are used in different context. The same hold for functions and mappings. A type construction means a certain set of values, that may be infinite. A set is a term that has a type (a set type, usually starting with a `$` symbol) and is always finite. When a type is needed, a set is not accepted and vice versa.

A function has a signature and means a set of pairs that may be infinite. A function is always implicitly given by its parameters and a defining term. A mapping has a type (with a `->` symbol in it) and is always finite. It may be explicitly or implicitly given. In the last case, a parameter, a set (term) and a defining term must be given.

The following examples are incorrect terms for this reason.

string	term to use
`[x:bool|not(x)]`	`[x:{false,true}|not(x)]`
`all[x:num|not(x*x < 0)]`	`true`
`set[x:num|5<x<13]`	`stretch(6,12)`

What is meant by the last non-term can be correctly formulated by the `stretch` function, giving the integers in a certain range. The following examples are incorrect types.

string	type to use
`{false,true}`	`bool`
`{5,6,7}`	`num` (cannot be strengthened)
`set[x:num|5<x<13]`	`num` (cannot be strengthened)

4 The dynamic part

4.1 Introduction

The dynamic part of the language is about modeling processes and their interaction. The theory behind the dynamic part of ExSpect is Petri net theory. We will give a brief introduction into the aspects that are relevant for modeling and specification.

A Petri net is a network of active objects called *processors* and passive objects called *channels*. The channels may contain any number of *tokens*. Depending on the kind of channels involved, the tokens in it may represent data, control or even physical objects. A special channel called *store* contains a single token at all times.

Processors can be connected to channels in three ways: for input, output and both. This is represented graphically by arrowheads. If the tokens in the input channels of a processor satisfy certain conditions, the processor may become activated. It then consumes certain tokens from its input channels and produces tokens for its output channels. The production of tokens may be subject to a *delay*. Delayed tokens become available only when the simulation clock has advanced the amount of time indicated by the delay.

Under which conditions a processor becomes activated and which tokens it may consume and produce follows from the definition of the processor. How a processor is defined is described in the next subsection.

A certain set of processors and channels can be grouped together in a subnet or *system*. Such a system can be used to build larger systems, by connecting some special channels or *pins* within the subsystem to the channels of the larger system. Connecting a processor or subnet into a larger net is called *installing*.

The final model is a system without pins. This way of hierarchical modeling is comparable to the well-known and intuitively clear DFD (data flow diagram) modeling technique.

In standard Petri net theory, processors are called *transitions* and channels are called *places*.

4.2 Processors

We now can explain how processors are defined and installed. Like a function, a processor has a name, parameters and a body. It also has a precondition and priority. The parameters are divided into input pin, output pin, store pin, value, and function parameters. The parameters consist of a name and a type (for function parameters a signature). The precondition contains a predicate: a term of type `bool`. The priority contains a term of type `real`. The body consists of a statement list. A statement is a conditional statement, the skip statement or an assignment. The statements are separated by commas (`,`) instead of semicolons, indicating that they can be executed concurrently: their order has no significance. A conditional statement consists of an if-predicate, a then-part and an optional else-part. The then- and else-parts are statement lists. An assignment consists of an output channel or store name, the assignment symbol (`<-`) and a term of the same type as the assigned channel or store. It may be followed by a delay term (of type `real`) preceded by the keyword `delay`. The terms and predicates may only contain variables amongst the non-output parameters.

For the definition of a processor, there is a default window that only contains in, out and store pin parameters and the body. The other parameters, precondition and priority are accessible by expanding it.

Processors can be *installed* in systems. The systems may have pins and parameters like a processor, but also channels and stores. If it has parameters, it can be installed in a higher-order system. Upon installing a processor, its pins are linked to either channels, stores or pins of the system, terms are attached to value parameters and functions to function parameters. Their types must match of course. Terms attached to a value parameter may only contain value parameter names of the system as variables; these must be filled in when the system itself is installed in a higher-order system. The topmost system has no value or function parameters, nor pins. Value and function parameters are static: their values/definitions are bound to the variables when they are installed. In the topmost system they are bound to fixed values/functions. Pins are dynamic: the values they contain may vary during the execution of the topmost system. When installing a processor, channels can also be indicated as *inhibitors* for that processor. This feature should be used with care; by using priorities instead of inhibitors one is able to obtain equal modeling power and express much clearer the modeler's intentions.

An installed processor is *enabled* when all channels marked as inhibitors for the processor are empty (i.e. contain no tokens) and the channels linked to its input pins contain a token combination that satisfies the precondition. Such a token combination is called a *binding*.

A processor that is enabled may *fire*. There may be several enabled processors and several bindings of a same processor. By setting *priorities* to a binding, one can influence the selection of a binding for firing the processor. For each binding, the priority of that binding is evaluated; of the bindings to processors with the highest priority one is selected. The default priority is zero, so it is possible to specifiy bindings with a higher as well as a lower priority than the default.

When firing, the tokens from the binding are consumed. Their values are bound to the corresponding parameter variables as are the store values. According to the specified statements, tokens are produced for the channels linked to the output pins, and the store values are updated. In a conditional statement, the then-part is executed if the condition predicate evaluates to `true`; the else-part if it evaluates to `false`. A missing else-part means that the statement is skipped in this last case. The skip statement performs no actions. An assignment statement causes the creation of an output token (for output pins) or an update of the store value (for store pins). The created token becomes available after d time units, where d is the corresponding value of the delay term. If the delay term is absent, $d = 0$. More assignments for the same output pin are allowed; in this case several tokens are produced for the same channel. More assignments for the same store pin are not allowed. Delayed assignments for store pins are not allowed either.

As an example, we use the finance system defined earlier. The processor `book` has the following definition.

Connections:

Name	Kind	Type
`amount`	`in`	`num`
`total`	`store`	`num`

Pre: `amount > 0`

Val/Fun parameters:

Name	Kind	Type	Value
`kind`	`val`	`str`	

Definition

```
if kind='credit' then total <- total - amount
                  else total <- total + amount fi
```

When it is installed with name `cr`, value parameter `kind` bound to `'credit'`, input pin `amount` bound to channel `a` containing 3 tokens with value `5`, `-3` and `7` respectively, store pin `total` bound to `t` containing the value `1000`, the following can happen.

The tokens with values `5`, `7` both satisfy the precondition, so any of them may activate `cr`. Suppose `5` is selected. The body is evaluated; because of the installation, the then-part is executed. So `total` is updated with `1000-5`. In the new situation, `a` contains tokens with values `-3`, `7`, and `t` contains `995`. If the token with value `7` has not disappeared in the meantime, it can again activate `cr` and result in `t` containing `988`. The token with value `-3` does not satisfy the precondition; it will not be consumed by `cr`. For more activations of `cr`, other processors, or the end user, must insert positive-valued tokens in `a`.

Precondition predicates determine whether a token combination is selected; the if-predicate determines what to do with the tokens once they are selected. So a processor without precondition (i.e. `true` as precondition) and body “`if` P `then` S `fi`” differs from the processor with precondition P and body S.

The first is activated by any input token combination; if this combination does not satisfy P, it is consumed without causing any output or store update. The second does not consume token combinations that do not satisfy P, so they may be left for another round or for other processors. For token combinations that do satisfy P, both act the same.

Like functions, processors can be polymorphic, i.e. their signature types may contain type variables. When a polymorphic processor is installed, the installed types must match the types in the definition. More explicitly, the same type variable `T` must correspond to the same installed type. Consider the following polymorphic processor `copy`.

Connections:

Name	Kind	Type
`inpt`	`in`	`T`
`outpt`	`out`	`T`

Pre:

Val/Fun parameters:

Name	Kind	Type	Value
`t`	`val`	`real`	

Definition

```
outpt <- inpt delay t
```

This processor cannot be installed by connecting the input pin to a channel of type `num` and the output pin to a `str` channel. Nor can it be installed by producing e.g. a `num` term for the value parameter `t`.

4.3 Systems

A system definition may be parametrized like a processor. When installing subsystems, their parameters must be bound. Systems can also be polymorphic. In the graphical part of the editor, pins are drawn, channels and stores defined and processors and subsystems installed. Value and function parameters can be added after opening a ’signature’ window. Defining channels and stores is done by drawing them and then giving a name and type construction and (for channels optionally) an initialization. A store initialization is a term having the type of the store. It may contain variables from the value and function parameters. A channel initialization may contain one or more terms of the channel type.

There are two special stores that are maintained by the ExSpect interpreter and that need not be initialized. One is the store `random`, containing a random number of type `real` between `0.` and `1.` at all times. Each time a random number is drawn from this store, the interpreter replaces its contents by another number that is totally unrelated to the old one. The other is the store `time` of type `real` containing the simulation clock. Special symbols (die and clock) are used to select them. It is very unwise to involve the values of these system stores in a precondition term (try it, and you’ll see why).

5 Modules and scoping

ExSpect has a number of libraries containing general-purpose types, functions, processors and systems. It is possible to add one’s own libraries. A library is called a “module” in ExSpect; modules can be imported by “including” them. The order in which modules are included is important.

When creating a module, definitions that are to be used outside the module must be *exported*. Types can be exported with or without their defining type construction. In the last case, all possible access functions of the type must be defined in the module, i.e. the type is abstract. In this case, it is possible to implement the type and its access functions differently without the users of the type noticing it. In the other case this is impossible. Also definitions that are accessible by users must be exported.

Modules are one way of limiting the scope of definitions. Another way is by a *where part*, that may accompany a single function or processor definition. The definitions in the where part are not known outside the definition that they accompany. Where parts may be used for creating more efficient definitions, when recursion is involved. Consider the following definition of the fibonacci function.

Name:	`fib`
Parameters:	`x:num`
Resulttype:	`num`
Definition:	`if x<2 then 1 else fib(x-1) + fib(x-2) fi`

Evaluating e.g. `fib(5)` could mean going through the following steps.

`fib(5)` → `fib(4)+fib(3)` → `fib(3)+fib(2)+fib(3)` → `2*fib(2)+2*fib(1)+fib(2)` → `3*fib(1)+3*fib(0)+2*fib(1)` → `5*1+3*1` → `8`

We see that in the above process, `fib(3)` is combined twice and `fib(2)` three times. When evaluating `fib` for higher values, this gets worse and worse, thus causing bad performance. It would be less inefficient to evaluate `fib(5)` as follows

`fib(5)` → `1*fib(5)+0*fib(4)` → `1*fib(4)+1*fib(3)` → `2*fib(3)+1*fib(2)` → `3*fib(2)+2*fib(1)` → `5*fib(1)+3*fib(0)` → `8`.

By creating a subordinate function with more parameters, this can be achieved The definition of fib becomes `fib2(x,1,0)`, where `fib2` with `num` parameters `x,y,z` is defined with the body

```
  if x<2 then y+z else fib2(x-1,y+z,y) fi
```

The function `fib2` has no use except in the context of the `fib` function, so it is natural to define it in the where part of `fib`. Where parts can be nested, but it is not advised to use this feature. It is not pleasant to hunt for definitions that are too deeply nested. It is better to put definitions on the same level.

We conclude by treating the scope of variables. The smallest and strongest scope is the variable in an implicit mapping term. In the term `[x:`A`|`E`]`, any variable `x` within term E is bound by the mapping, and thus is interpreted as an element of the set A. An occurence of `x` in A is not bound by it, so it must occur in some wider scope. An example is `[x: stretch(0,x)| x+1]` in a function definition where `x` is a parameter. Here, the `x` in `x+1` is the mapping variable, whereas the one in `stretch(0,x)` is the function parameter.

The second scope is formed by parameters and where parts in a definition. The variables there take priority over those in the third scope, the global and imported definition names.