How to build your own analysis: The FULA language

» Back to overview

The FULA language is a first order functional language with eager evaluation. It offers most of the constructs found in modern functional languages together with some extensions for sets and data flow analysis.

Data types

FULA supports the following datatypes:

- *Primitives* (snum, str, bool): `snum` denotes 32bit signed integers, `bool` consists of `true` and `false`, and `str` can be any character string in double quotes (like `"This is a string"`).
- *Sets* over any type can be defined in the TYPE section. Sets can be enumerated: `{` e_1`, ...,` e_n `}`, n>=0. `{}` denotes the empty set and `all` denotes the full set of all elements of the base type. Since sets are always complete lattices ordered by subset inclusion, `{}` can also be written as `bot` and `all` can be written as `top`.
- *Lattices* can be defined in the TYPE section by the `flat` operator or as `lift`ed lattices of a base type.
- *Lists* of any type can be defined in the TYPE section. Lists can be enumerated: `[` e_1`, ...,` e_n `]`, n>=0, or constructed with a cons operator e_1`:`e_2`:...:[]`. `[]` denotes the empty list.
- *Tuples* of any types can be defined as the crossproduct e_1`* ...*` e_n, n>=2. Tupel expressions are written as `(` e_1`, ...,` e_n `)`, n>=2.
- *Functions.* FULA knows two kinds of functions. On the one hand there are function types defined in the TYPE section. On the other hand there are functions defined in SUPPORT section. We will call functions defined in the SUPPORT section *static* functions because they are defined together with their body, which cannot be changed or updated. Thus they cannot change during the execution of your analyzer. Obviously we will call the other kind of function *dynamic*. Dynamic functions are very different from static functions; they do not even have a function body. You should rather think of dynamic functions as lists of argument-value pairs. Dynamic functions must always have a default value, i.e. for each x that is not explicitly specified the function takes this default value. A dynamic function may look like this: `h = [->3]\[1->1, 2->4, 5->6]`. This means that h(1)=1, h(2)=4, h(5)=6 and h(x)=3 for all other x. The type of h is `snum->snum`. While static functions may only be applied to some arguments, dynamic functions are first-class FULA values that can be used as arguments or results of functions, can be bound to variables, and used as generators in ZF expressions. In contrast to static functions, dynamic functions can only have one argument (because they are really just argument-value pairs). If you like to build dynamic functions with several arguments you will have to define a tuple type of the argument types and then build a dynamic function from that tuple to the result type.

Patterns

FULA defines static functions by pattern matching. Patterns may also be used in `let` and `case` expressions as well as local definitions in ZF expressions. The following kinds of patterns exist:

- A *variable* matches arbitrary values, and is bound to the value being matched.
- The *wild card* or universal matcher matches any value without introducing a name binding.
- A *constant*, that is a number, `true`, `false`, `top`, `bot`, `all`, `{}`, `[]` or a literal string, matches only one value, namely the value of the constant.
  (Variables, the wild card, and constants are *simple patterns*.)
- A *tuple pattern* `(` p_1`, ...,` p_n `)` matches tuples of length n if each pattern p_i matches the corresponding tuple entry e_i.
- A *list pattern* is a cons pattern `head:tail` that matches lists with at least one element if `head`

matches the first element in the list and `tail` the rest of the list. `head` and `tail` may again be arbitrarily complex patterns. For example `(x,y):a:b` will match the list of tuples `[(1,2), (4,5), (3,6), (2,7)]`. After the match, the pattern variables have values x=1, y=2, a=(4,5) and b=[(3,6), (2,7)].
- An *alias pattern* `p as x` consists of a pattern `p` and a variable `x`. It is equivalent to the pattern `p` except that it binds the variable `x` to the value that is being matched. For instance, `(1,b) as c` matches all pairs whose first component is 1, and binds `b` to the second component of the pair and `c` to the whole pair. This is useful if the pair is needed later since a reference to `c` (i.e. the already existing pair) is more efficient than `(1,b)` which builds a new pair.

Note: Beware of using the same variable twice in the same pattern: Each occurrence of a variable introduces a new instance of that variable, shadowing all other occurrences of the same variable. For example the pattern `(v,v)` will match all pair expressions, binding `v` to the second component. If you want to do something special for pairs consisting of two identical elements, use an if-expression with an equality test.

Operators

Besides a set of predefined functions FULA provides the following infix operators:

- `=, !=` (equal, not equal). These operators work with every datatype including custom types you define in the TYPE section.
- `<, <=, >, >=.` These comparison operators work with all data types (including your own) that are complete lattices. In addition they work as expected with the data type `snum`.
- bool values b can be negated using !b and combined using && (logical and) and || (logical or). These operators are lazy, i.e. the arguments are evaluated from left to right only until the value of the whole expression is clear. So in `true || b` the expression b is not evaluated since the result is true anyway.
- The tuple selection operator # can be used to access elements of a tuple: `(`e_1`, ..., `e_n`)#i =` e_i, 1<=i<=n.
- `lub, glb` are infix operators that compute the least upper bound or the greatest lower bound of two elements of the same lattice.
- *Function application* is denoted by `().` `f(`e_1`, ..., `e_n`)` denotes the application of the static function f to its n arguments which must have appropriate types. ``e_1`(`e_2`)` denotes the dynamic application of e_1 to e_2. Here, e_1 must evaluate to a dynamic function of type *a->b* and e_2 must be of type *a*. The result is a value of type *b*.
- *List construction* `:`, e.g. `1:2:[3,4] = [1, 2, 3, 4]`. Note that `:` is right associative! In `a:b` if `a` is of type *a*, then `b` must be of type list(*a*).
- *Member test* `?` for checking membership in a set. `a?b` returns `true` if `a` is a member of the set `b`, and `false` otherwise. If `a` is of type *a*, `b` must have type set(*a*). (This operator does not work for lists.)
- The + operator is heavily overloaded in PAG/WWW. As expected it denotes the addition for snums. For other data types + means whatever is appropriate:
  - For `str` it means string concatenation: `"Hello" + "World" = "HelloWorld"`.
  - For sets it means set union: `{ x, y, z } + { x, a, b } = { x, y, z, a, b }`. You can also use it to add single elements to a set: `{ x, y } + a = a + { x, y } = { x, y, a }`.
    BE CAREFUL: + is left associative, so `1+2+{3,4} = (1+2) + {3,4} = 3 + {3,4} = {3,4}` BUT `1+(2+{3,4}) = 1 + {2,3,4} = {1,2,3,4},` and the same for `{3,4}+1+2`.
  - For lists it means list append. `[x,y] + [x,a,b] = [x,y,x,a,b]`. You can use + to append single elements to a list just like sets, but the same problems may arise because + is left associative. It is recommended to keep the brackets `[...]`, or to use the list construction operator `:`.
- The operator – denotes subtraction of snums. For sets you can use the - operator to remove elements from a set: `{x,y,z} - z = {x,y}` and `{x,y,z} - {x,y,a,b} = {z}`
- The remaining arithmetic operators `*, /, %, ^` only apply to two integers (snum). % denotes the remainder of a division (11%3 = 2), and ^ denotes exponentiation (2^3 = 8).

Control structures

FULA offers two control constructs: the conditional evaluation `if ... then ... else ... endif` and the `case` statement which offers a pattern-controlled choice between many alternatives.
An expression of the form `if` e_1 `then` e_2 `else` e_3 `endif` evaluates to e_2 if e_1 evaluates to `true`, and to e_3 otherwise. e_2 and e_3 must have the same type so that the whole if expression has a unique type no matter which value e_1 evaluates to.

The `case` construct allows a multi-pattern choice. An expression tuple is in turn matched against a pattern tuple. As soon as a match succeeds for all expressions, the expression associated with that match is evaluated and returned. If no alternative matches a run time error occurs. For example:

```
case 1, [4,5] of
x, [] => x;
x, y:z => x+y;
endcase
```

evaluates to 5, because `x` matches `1` and the cons pattern `y:z` matches the list `[4,5]`. So `x` is bound to 1 and `y` is bound to 4, and the whole expression evaluates to `x+y` which is 5.
Naturally the right hand sides of all alternatives must have the same type, so that the whole expression has a unique type no matter which alternative matches. In addition all pattern tuples must have the same type (in the example above it is `snum, list(snum)`), and this type must be the same as the type of the expression tuple to match. (This is not a restriction because patterns with wrong types would be useless anyway since they would never match). Variables introduced in patterns are local to their alternative, i.e. they are only visible in the corresponding right hand side of the alternative.

Local definitions

Local definitions are introduced by the `let` construct:
`let` p_1`=`e_1`, ..., `p_n`=`e_n `in e`
This matches expressions e_1`..`e_n against patterns p_1`..`p_n, introducing new variable bindings in the patterns. These variables are visible in expression `e` and the variables introduced in p_i are visible in the following expressions e_j with `j>i`, and the value of `e` becomes the value of the whole expression. Note that all expressions e_i are evaluated before the evaluation of `e`, no matter whether the variables in pattern p_i are needed in `e`.
As a special case FULA offers *strict binding* of local names, by using p_i `<=` e_i instead of p_i`=`e_i. You can use p_i `<=` e_i if e_i is of a complete lattice type of the form `lift(`*a*`)` or `flat(`*a*`)`. If e_i evaluates to `top` (`bot`), the whole `let` expression evaluates to `top` (`bot`). Otherwise p_i is matched against `drop(`e_i`)`, i.e. the value in *a* which corresponds to the value of e_i in `lift(`*a*`)` or `flat(`*a*`)`. So the expression `let x<=y in z` is really a nice form to write `if y=top then top else if y=bot then bot else let x=drop(y) in z endif endif`. Note that the expressions e_i are evaluated from left to right so that `let x<=top, y<=bot in z` evaluates to `top` since `x<=top` is evaluated first and already gives the value of the whole `let` expression.

ZF expressions

ZF expressions (sometimes called list comprehensions) give you the ability to dynamically generate new sets, lists and dynamic functions from other expressions. FULA knows three kinds of ZF expressions:

- `{ e |` zf_1`; .. ;` zf_n `}` with n>=1 generates a new set.
- `[ e |` zf_1`; .. ;` zf_n `]` with n>=1 generates a new list.
- `[ [->`e_0`] \ e |` zf_1`; .. ;` zf_n `]` with n>=1 generates a new dynamic function. e_0 is the default value of the new function. (This expression must not depend on the variables bound in the zf_i.) `e` must either be a list of function updates or evaluate to a argument-value pair. (We admit that this syntax is not optimal.)

The syntactic entities zf_i may be generators, local definitions, or filters.

- `p in e` is a generator which consists of a pattern `p` and an expression `e` which must be a set, a list, or a dynamic function. Each element in this set or list (each argument-value pair in this function) is successively matched against the pattern `p` binding variables in `p` to new values any time. (For functions, the default value must be excluded to prevent the generation of infinitely many argument-

value pairs; see below.)
- `let` p_1=e_1`, .., ` p_n=e_n is a local definition. (Definitions of the form `p <= e` are not allowed here.)
- A *filter* can be any expression of type `bool`.

In contrast to `let` expressions, variables bound in some zf_i are already visible in all zf_j with j>i.

To understand these rather complex constructs it is best to look at some examples:

```
{ e | i in [1, 2, 3, 4]; let e = (i,5); i>2 }
```
Now this means: Make a set from all expressions `e` that fit the definition of the zf_i. First there is a generator. It successively binds `i` to all the values in the list (i.e. 1, 2, 3, and 4). For each such `i`, the local definition binds `e` to the tuple `(i,5)`, but only when the filter is true (i.e. if i>2). So `e` is bound to (1,5), (2,5), (3,5) and (4,5) but only the last two pairs pass the filter, so the resulting set is {(3,5), (4,5)}. Another way to obtain the same result is
`{ (i,5) | i in [1, 2, 3, 4]; i>2 }`.

```
[ [->1]\(x,y) | (a,b) in { (1,2), (3,4) }; let x=a+b; let y=x+2 ]
```
This ZF expression will evaluate to a dynamic function. Here the tuple `(a,b)` is generated from the tuples in a set. Then `x` is bound to the sum of `a` and `b`. So `x` is successively bound to 3 and 7. `y` is always bound to `x+2`. So the whole expression becomes the dynamic function `[->1]\[3->5, 7->9]`.

If you want to generate values from a dynamic function, you have to exclude the default argument to prevent PAG/WWW from generating infinite lists. Think of the following dynamic function: `f = [->1]\[1->2, 2->3]`. A simple generator would generate the infinite list of argument-value pairs (1,2), (2,3), (3,1), (4,1), (5,1)..... To prevent this you have to specify the default argument in the generator, so that PAG/WWW only generates a finite set of pairs. `x in f\1` will generate all pairs from the dynamic function `f` except those with value 1, so in our example `x` would be bound to (1,2) and (2,3). Note that nothing will keep you from writing `x in f\2` but then PAG/WWW will generate a run time error because this would cause an infinite list of pairs to be generated.

One last example:
```
let f = [->1]\[2->3, 3->4, 5->4] in
    [ [->2]\[y->z] | (a,b) in f\1; let y=a+b, z=b-a; b>a ]
```
will generate the dynamic function `[->2]\[5->1, 7->1]`

For further information see the complete FULA grammar.

Next step

- Global FULA variables

Contents

Search