How to build your own analysis: Built-in PAG/WWW functions

» Back to overview

PAG/WWW has some built-in support functions that make it easy for you to work with the built-in datatypes. You will need these functions to get data from WHILE-specific datatypes like `Expression`.

The following table gives you a list of all built-in functions. Note that you could not write most of these functions yourself because they use polymorphic types (denoted by *a*, *b*...), but you must always specify monomorphic types in your own SUPPORT functions.

`sgn :: snum -> snum`	The signum function which could be defined as `sgn(n) = if n>0 then 1 else if n=0 then 0 else -1 endif endif`
`abs :: snum -> snum`	Returns the absolute value of its argument.
`reverse :: list(a) -> list(a)`	Takes a list and reverses the order of its elements. `reverse( [1, 2, 3, 4] ) = [4, 3, 2, 1]`
`hd :: list(a) -> a`	Takes a list and returns its first element. `hd( [1, 2, 3, 4] ) = 1` Applying `hd` to an empty list will generate a run time error.
`tl :: list(a) -> list(a)`	Takes a list and removes its first element. `tl( [1, 2, 3, 4] ) = [2, 3, 4]` Applying `tl` to an empty list will generate a run time error.
`union :: set(a) * set(a) -> set(a)`	Takes two sets and returns their union. You will see easily that `union(x,y)` is equivalent to `(x lub y)` and `(x+y)` if `x` and `y` have the same type `set(T)`. This is because power sets are lattices ordered by subset inclusion, so the union of two sets is just their least upper bound (lub) in the lattice. Sometimes the use of `union` makes your code easier to read.
`intersect :: set(a) * set(a) -> set(a)`	Takes two sets and returns their intersection. Instead of `intersect(x,y)`, you could write `(x glb y)` just as well, but note that `x-y` means something very different.
`lift :: a -> lift(a)` `lift :: a -> flat(a)`	If `x` is a value of type `T`, then `lift(x)` is the corresponding value in the

	lifted type `lift(T)` or `flat(T)`. Since `lift(T)` and `flat(T)` cannot coexist in the same analysis (as noted in the description of the TYPE section), it is always clear from the context whether `lift(x)` has type `lift(T)` or `flat(T)`.
`drop :: lift(a) -> a` `drop :: flat(a) -> a`	This is the inverse operation to `lift`. The argument must be of type `T'=lift(T)` or `T'=flat(T)` and then the result is of type `T`. The result of a correct application of `drop` is given by `drop(lift(x))=x`. If `drop` is applied to the top or bottom element of `T'` (so that there is no corresponding value of type `T`), then it produces a run time error.
`error :: str -> a`	The error function prints its argument to `stderr` and then exits the analyzer with an error. Since the return type is polymorphic, you can use the error function at any position in an expression. Use this function to generate your own run time errors.
`crunch :: (a ->d b) * (a ->d b) * (b * b ->s b) -> (a ->d b)`	The `crunch` function takes two dynamic and one static function and creates a new dynamic function by combining the results of the two dynamic functions by means of the static function. The call `crunch(f, g, h)` builds a new dynamic function `k` such that `k(x) = h(f(x), g(x))` for every `x` of type *a*. Example: Consider the two dynamic functions `f = [->1]\[1->1, 2->2, 3->3]` and `g = [->2]\[1->3, 3->5, 5->7]` (where `f` and `g` are both of type `snum->snum`) and the static function `h :: snum * snum -> snum; h(x,y) = x+y`. Then `crunch(f,g,h)` builds a new dynamic function `k` by applying `h` to the corresponding results of `f` and `g`, e.g. `k(1) = h(f(1),g(1)) = h(1,3) = 4`. Thus the resulting function will be `k = [->3]\[1->4, 2->4, 3->8, 5->8]`.
`expType :: Expression -> str`	Returns the type of an

	expression as a string. It is one of: `ARITH_BINARY` A binary aexpression (`a+b, a-b, a*b or a/b`). Call `expOp` to get the operator. `BOOL_BINARY` A binary bexpression (`a<b, a<=b, a=b, a>b, a>=b, a<>b`). Call `expOp` to get the operator. `ARITH_UNARY` The unary aexpression `-a`. `BOOL_UNARY` The unary bexpression `not a`. `VAR` A single variable. `CONST` An integer constant. `TRUE` The boolean constant `true`. `FALSE` The boolean constant `false`.
`expOp :: Expression -> str`	Returns the operator of an expression as a string. `expOp(e)` returns one of +, -, *, /, <, <=, =, >, >=, <>. It is only defined if `expType(e)` is either `ARITH_BINARY` or `BOOL_BINARY`. Otherwise it creates a run-time error.
`expSub :: Expression -> Expression`	Returns the subexpression for unary operators. If `e` is of the form `(-e1)` then `expSub(e)=e1`, if `e` is of the form `(not e1)` then `expSub(e)=e1`. If `expType(e)` is not `ARITH_UNARY` or `BOOL_UNARY`, then `expSub` produces a run-time error.
`expSubLeft :: Expression -> Expression`	Returns the left subexpression for binary operators. If `e` is of the form `(e1+e2)`, `(e1-e2)`, .., `(e1<e2)`, `(e1=e2)`... then

	`expSubLeft(e)=e1`. If `expType(e)` is not `ARITH_BINARY` or `BOOL_BINARY`, then `expSubLeft` produces a run-time error.
`expSubRight :: Expression -> Expression`	Returns the right subexpression for binary operators. If `e` is of the form `(e1+e2)`, `(e1-e2)`, .., `(e1<e2)`, `(e1=e2)`... then `expSubLeft(e)=e2`. If `expType(e)` is not `ARITH_BINARY` or `BOOL_BINARY`, then `expSubRight` produces a run-time error.
`expVar :: Expression -> Var`	`expVar(e)` returns the variable in `e` if `expType(e)` is `VAR`. Otherwise `expVar` produces a run-time error.
`expVal :: Expression -> snum`	Returns the value of the constant if `expType(e)` is `CONST`. Otherwise `expVal` produces a run-time error.

Next step

- A formal description of the analysis specification language

Contents

Search