Illusyn

Version 6.4

Tero Hasu

April 11, 2016

Illusyn (Illusionary Syntax) is a Racket-based library. It supports declarative definition of *abstract syntax trees* (ASTs) for purposes of program transformation. Both the programming interface as well as the implementation of the interface gets generated from declarative specifications. As a form of abstraction, it is possible to declare multiple interfaces per AST node type, with each interface also exposing the "structure" of its data.

The library also includes operations for transforming and serializing AST nodes. Transformations may be implemented either using Racket's native functionality (e.g., `match` for pattern matching), or through Illusyn's support for *strategic pattern matching* (Visser 1999).

1 Concrete AST Node Types

```
(require magnolisp/ast-repr)
```
package: magnolisp

```
(define-ast id views fields maybe-opts)

               views = (cview ...)

               cview = view-id
                     | (view-id cview-spec)

          cview-spec = (field-spec ... cview-opt ...)
                     | (#:fields field-name ... cview-opt ...)

           cview-opt = #:copy copier-expr
                     | #:predicate predicate-expr

              fields = (field ...)

               field = (#:none field-name)
                     | (#:maybe field-name)
                     | (#:just field-name)
                     | (#:many field-name)

          maybe-opts = maybe-singleton
                       maybe-custom-write
                       maybe-struct-options

     maybe-singleton =
                     | #:singleton (arg-expr ...)

  maybe-custom-write =
                     | #:custom-write writer-expr

maybe-struct-options =
                     | #:struct-options (struct-opt-datum ...)
```

Defines a new AST node type named *id*.

Defining a node type involves also defining a new, underlying structure type named *id*. The defined type has no supertypes, but it is possible to specify a set of *views* that the new type should implement; each *view-id* should name a view defined using `define-view`.

The structure type will have a field for each specified *field-name*. Each *field* specification should also include one of the term quantity modifiers `#:none`, `#:maybe`, `#:just`, or

`#:many` to indicate the desired behavior during strategic term rewriting. Any field tagged with `#:none` is not traversed into during a rewrite. Any value of a `#:just` field is treated as a single term to be traversed. Any value of a `#:maybe` field is traversed and treated as a term unless it is `#f`. Any `#:many` field value is assumed to be a list whose elements are terms to be traversed.

The `#:singleton` option modifies the behavior of `define-ast` so that an *instance* of the structure gets bound under the name `the-id`, where the instance is created by calling the constructor using the arguments specified by `(arg-expr ...)`. This option does *not* in any way modify the way that `define-ast` defines the affected AST node type itself, or associated operations. In particular, all the usual operations for creating other copies of the "singleton" object are available.

By default, each AST node type will implement the `gen:custom-write` interface such that the `write-proc` method will *not* print the contents of any field named `annos`. To modify the default behavior, use the `#:custom-write` option to specify a different `write-proc` implementation. The *`writer-expr`* should be an expression that specifies a suitable function; see `gen:custom-write` documentation for further information about what said function must be like.

The defined structure type is always marked `#:transparent`, to allow for run-time introspection. Additional `struct` options may be specified; each *`struct-opt-datum`* given is passed directly onto `struct`.

The following interfaces and operations are implemented for AST node types:

- Each type has the operations usually associated with immutable structure types. For example, each type *`id`* gets: a membership predicate *`id?`*; a constructor *`id`*, and a getter *`id-field-name`* for each of its fields. Racket's `struct-copy` and `match` constructs also cooperate with the structure definition.
- Each type implements the `gen:custom-write` interface, as explained above.
- Each type implements the `gen:equal+hash` interface. The implementation defines the meaning of `equal?` comparisons when instances of the type are involved; hashing functions are also defined. The `equal?` comparison is implemented by comparing the values of all fields with `equal?`, except that any field named `annos` is ignored.
- Each type implements the `gen:syntactifiable` interface, thus defining how to serialize an instance of the type as Racket code.
- Each type implements the `gen:strategic` interface. This defines the node-type-specific functionality required to support the `magnolisp/strategy-stratego` API. The `magnolisp/strategy-stratego` module implements the primitive traversal combinators that it requires in terms of `gen:strategic`. Said combinators are required as a basis for generic traversals over AST nodes.

- Each type includes the structure type property `prop:view-id` for each of the *views* it supports. The value of the property is set to contain type-specific implementations of the primitive view operations, to which the generic view functions dispatch dynamically as appropriate. If no overriding implementation is specified as *cview-spec*, then the default implementations of the operations (as specified for the view) are generated. See `magnolisp/ast-view` for information on defining views, the operations associated with views, and the *field-spec* and *field-name* grammar rules.

  The `#:copy` option is specific to `define-ast`, and may be used to specify a non-default `copy-view-id` operation, with *copier-expr* being an expression that specifies a suitable function.

  Similarly, the `#:predicate` option may be used to specify a *predicate-expr*, which must evaluate to a function accepting one argument. Any specified predicate is used for determining whether a given instance of *id* is a member of an associated *view-id*; in other words, the predicate determines the result of *view-id?*, when applied to instances of *id*. The default behavior is for *all* instances of *id* to be members of any listed *view-id*. A `#:predicate` may only be specified for views declared with the `#:partial` attribute.

- Each type *id* implements the comparison relation *id=?*, for compatibility with views. For AST node types, this operation is like `equal?`, but with the additional requirement that both arguments must be of type *id*; otherwise an argument error is raised.

- Each type *id* implements the operation `copy-id`. This operation is provided for compatibility with views, and has a signature consistent with view-based `copy-view-id` operations. The operation is implemented as a function. The first argument should be the instance being "copied," but it is actually ignored. The rest of the arguments are the positional field values for the new object. See `magnolisp/ast-view` for more details on this operation.

- Each type *id* also implements the operation `set-id-field-name` for each of its fields. These operations are functionally updating field setter operations, appropriate for immutable structure types. The signature of the operations is consistent with the operations that views have. Again, see `magnolisp/ast-view` for more details.

For example:

```
> (define-ast Var () ([#:none name]))

> (define-ast Inc () ([#:just exp]))

> (define-ast Add () ([#:many exp-lst]))

> (Add (list (Var 'x) (Inc (Var 'y))))
(Add ((Var x) (Inc (Var y))))
> (Var? (Var 'x))
#t
```

```
> (Var=? (Var 'x) (Var 'x))
#t
> (Var=? (Var 'x) (Var 'y))
#f
> (equal? (Var 'x) 42)
#f
> (Var-name (Var 'x))
'x
> (set-Var-name (Var 'x) 'y)
(Var y)
> (struct-copy Var (Var 'x) [name 'y])
(Var y)
> (copy-Var (Var 'x) 'y)
(Var y)
> (copy-Var 'whatever 'y)
(Var y)
> (match (Var 'x) [(Var n) n])
'x
> (syntax->datum (syntactifiable-mkstx (Var 'x)))
'(Var 'x)
> ((all-visitor displayln) (Add (list (Var 'x) (Inc (Var 'y)))))
(Var x)
(Inc (Var y))
```

```
(define-ast* id views fields maybe-opts)
```

A variant of `define-ast`, such that it also `provides` any bindings that it introduces.

1.1 View Structure

It is possible to enumerate view-based substructure of an AST node, with strategies instantiated using `make-view-all`, `make-view-some`, and `make-view-one`. Instantiation will only succeed for views that have been marked as `#:traversable`. Instantiated stragies can only be used on nodes that implement their specific view.

```
(make-view-all view-id)
```

Instantiates a *view-id*-specific variant of the `all` strategy combinator. See §2 “AST Node Abstract Interfaces” for information about views, and §4.3.2 “Primitive Strategy Combinators” for information about `all` and other structure-aware strategy combinators.

```
(make-view-some view-id)
```

Instantiates a *view-id*-specific variant of the some strategy combinator.

```
(make-view-one view-id)
```

Instantiates a *view-id*-specific variant of the one strategy combinator.

2 AST Node Abstract Interfaces

`(require magnolisp/ast-view)` package: `magnolisp`

The `define-ast` form creates at least one interface for the AST node type being defined, namely the one corresponding to the concrete `structure` and its field names. It can also be useful to define additional *abstract interface*s (here called *view*s) that perhaps capture some common structure or semantics shared by a number of different concrete node types. Such interfaces can be defined using the `define-view` form. One then merely needs to list the views that each `define-ast` declared type should implement, and the necessary support code will be automatically added for the concrete node type in question.

```
(define-view id view-spec view-option ...)

  view-spec = (field-spec ...)
            | (#:fields field-name ...)

 field-spec = (#:field maybe-qty field-name)
            | (#:field maybe-qty field-name #:use c-field-name)
            | (#:access maybe-qty field-name get-expr set-expr)

  maybe-qty =
            | qty

        qty = #:none
            | #:maybe
            | #:just
            | #:many

view-option = #:also (other-view-id ...)
            | #:partial
            | #:traversable
```

Declares a view named *id*. Each *field-name* is an identifier naming one of the abstract fields of the view. The form with the `#:fields` keyword is shorthand for `(define-view` *id* `([#:field` *field-name*`] ...))`.

A field specification of the form `[#:field` *maybe-qty field-name*`]` means that the view should have a field named *field-name*, implemented so that the field maps directly to a field of the same name in a concrete structure type.

The `#:use` variant of the `#:field` specification may be used to specify a different concrete structure field name *c-field-name* for a by-name mapping of fields. This can be particularly useful for overrides when using `define-ast`, if some concrete structure has an unusually named field of compatible semantics.

A field specification of the form `[#:access maybe-qty field-name get-expr set-expr]` means that the view should have a field named `field-name`, with field access implemented as specified. The expression `get-expr` specifies a getter function (taking an AST node as an argument, and returning the value of the abstract field), and the expression `set-expr` specifies a setter function (taking an AST node as the first argument, an abstract field value as the second argument, and returning a modified AST node).

Where a view is marked as `#:traversable`, a term quantity value `qty` must be specified for all of its fields.

When `other-view-id`s are declared for a view `id`, then any concrete node type declared as having the view `id` will also have the `other-view-id` views, implemented as originally specified for those views. This is a facility that simulates "inheritance" between views.

The following interfaces and operations are implemented for views:

- Each view `id` gets its own corresponding structure type property `prop:id`. Each AST node type that has said "viewpoint" sets the property in order to implement the data-type-specific parts of the generic interface of the view.
- Each view `id` gets a membership predicate `id?`, which holds for all AST nodes that have the view. A `#:partial` view may include only a subset of the values of a given concrete node type that implements the view.
- Each field `field-name` of the view `id` will get accessors. The getter is named `id-field-name`, and the functional setter is named `set-id-field-name`.
- Each view `id` also gets a function for constructing a different instance of the object being accessed through the view. The function is named `copy-id`, and its signature is `(-> id? any/c ... id?)`; that is, the function takes an object to functionally modify, new values for the fields of the view (ordered as declared), and returns a new copy of the original object, with the passed abstract view values incorporated into the concrete object.
- Each view `id` also gets a comparison function named `id=?`, which compares only the parts of the parent object that correspond to the abstract field values that belong to the view. The comparison of said values is done with `equal?`. Note that only the immediate object comparison is view specific, and any sub-objects of the abstract field values are compared with `equal?`, even if they also implement the same view.
- Racket's `matching` construct automatically sees to it that any `structure` types can be matched against by name; this means that any AST node types defined with `define-ast` are also `matchable`. To make views behave in the same way for purposes of matching, every view `id` also gets a match pattern of the form `(id pat ...)`, where each `pat` is a pattern that matches against the corresponding abstract field value. This is implemented by using `define-match-expander` to define a view-specific *match expander* named `id`.

- Each `#:traversable` view additionally gets the operations *id*`-term-fields` and `set-`*id*`-term-fields`. Their semantics are whatever is required by `make-view-all`, `make-view-some`, and `make-view-one`. (In essence, they semantics are the same as for `term-fields` and `set-term-fields`, but for the view *id* rather than for concrete nodes.)

For example:

```
> (define-view Empty ())

> (match 5 [(Empty) 'yes] [_ 'no])
'no
> (define-view WithV (#:fields v))

> (WithV? 5)
#f
> (define-view WithW ([#:field w]))

> (match 5 [(WithW w) 'yes] [_ 'no])
'no
> (define-ast HasVW (WithV WithW) ([#:none v] [#:none w]))

> (define vw (HasVW 1 2))

> (WithV-v vw)
1
> (WithW-w vw)
2
> (match vw [(WithW w) w] [_ 'no])
2
> (match vw [(WithW (? number? w)) w] [_ 'no])
2
> (match vw [(and (WithV v) (WithW w)) (list v w)] [_ 'no])
'(1 2)
> (set-WithV-v vw 1/2)
(HasVW 1/2 2)
> (copy-WithW vw 2/3)
(HasVW 1 2/3)
> (WithW=? vw (set-WithV-v vw 1/2))
#t
> (define-view Ast ([#:field #:none annos]))

> (define (get-type ast)
    (hash-ref (Ast-annos ast) 'type))

> (define (set-type ast t)
```

```
    (set-Ast-annos ast (hash-set (Ast-annos ast) 'type t)))

> (define-view Expr ([#:access #:maybe type get-type set-type])
    #:also (Ast))

> (define-ast Lit (Expr) ([#:none annos] [#:none dat]))

> (define five (Lit #hasheq((type . int)) 5))

> (Lit-dat five)
5
> (Expr-type five)
'int
> (define str (set-Expr-type (set-Lit-dat five "str") 'string))

> (match str [(and (Expr 'string) (Lit _ d)) d] [_ 'no])
"str"
> (define-view TypeExpr ())

> (define-view DefType ([#:field id]) #:partial)

> (define-ast ForeignTypeName (TypeExpr) ([#:none id]))

> (define (DefVar-DefType? ast)
    (TypeExpr? (DefVar-val ast)))

> (define-ast DefVar ([DefType (#:predicate DefVar-DefType?)])
    ([#:none id] [#:just val]))

> (DefType? (DefVar 'x five))
#f
> (DefType? (DefVar 'Int (ForeignTypeName 'long)))
#t
> (DefType-id (DefVar 'Int (ForeignTypeName 'long)))
'Int
```

(define-view* ....)

A variant of `define-view`, with the same form and semantics, except that it also `provides` any top-level bindings that it introduces.

3 AST Serialization

```
(require magnolisp/ast-serialize)
```
package: magnolisp

It can be useful to be able to persist an AST by storing it in a Racket module as code. The `define-ast`-defined AST node types support serialization into code, and the same is true for most basic Racket data types that might appear in the `#:none` fields of nodes.

```
gen:syntactifiable : any/c
```

A *generic interface* for as-Racket-code serializable values. The following method must be implemented:

- `syntactifiable-mkstx`

```
(syntactifiable? v) → boolean?
  v : any/c
```

A membership predicate associated with the `gen:syntactifiable` interface. Returns `#t` if and only if `v` is a value whose type implements the interface.

```
(syntactifiable-mkstx v) → syntax?
  v : any/c
```

A generic method that returns syntax for code that reconstructs value `v`, although possibly in a somewhat information-losing way (in that the constructed value is the same to some extent, but not necessarily precisely so). By definition, any value `v` whose type implements `gen:syntactifiable` supports this operation. Other supported types are `syntax?`, `null?`, `pair?`, `box?`, `vector?`, prefab `structure` types, `hash?`, `path?`, `symbol?`, `number?`, `boolean?`, `string?`, `char?`, `keyword?`, `bytes?`, and `regexp?`.

Any mutability or weakness properties associated with basic collection types are not preserved. For syntax objects, location information and the `'origin` and `'paren-shape` syntax properties are preserved in the serialized form.

For example:

```
> (for/list ([ast (list five str)])
    (syntax->datum (syntactifiable-mkstx ast)))
'((Lit (make-immutable-hasheq (list (cons 'type 'int))) '5)
  (Lit (make-immutable-hasheq (list (cons 'type 'string)))
'"str"))
```

4 Traversing and Rewriting

```
(require magnolisp/strategy)
```
package: magnolisp

To support program analyses and transformations, Illusyn includes a library of operations for generic traversals over ASTs and strategic rewriting of ASTs.

4.1 Abstraction over Concrete Term Specifics

As we want to implement generic traversals over AST nodes, we want to mostly abstract over the specifics of individual AST node types. This abstraction is defined by the `gen:strategic` interface; each AST node being traversed or rewritten is expected to implement it.

```
gen:strategic : any/c
```

A *generic interface* that supplies operations for traversing and rewriting a structure type. The following methods must be implemented:

- `term-visit-all`
- `term-rewrite-all`
- `term-fields`
- `set-term-fields`

```
(strategic? v) → boolean?
  v : any/c
```

A membership predicate associated with the `gen:strategic` interface. Returns `#t` if and only if `v` is a value whose type implements the interface.

```
(term-visit-all s ast) → void?
  s : (-> strategic? any)
  ast : strategic?
```

A `gen:strategic` generic method. Applies function `s` for its side effects on all the immediate sub-terms of the term `ast`.

```
(term-rewrite-all s ast) → (or/c strategic? #f)
  s : (-> strategic? (or/c strategic? #f))
  ast : strategic?
```

A `gen:strategic` generic method. Applies rewrite rule `s` to all the immediate sub-terms of the term `ast`. Returns the rewritten term if the rule applies to all the sub-terms; otherwise returns `#f`.

Where `(eq? sub (s sub))` holds for all rewritten sub-terms `sub`, then this function returns its `ast` argument. This makes it possible to detect whether the term was changed.

```
(term-fields ast) → (listof (or/c strategic? #f list?))
  ast : strategic?
```

A `gen:strategic` generic method. Returns a list of the values of the potentially sub-term containing fields of term `ast` (i.e., the fields declared as `#:just`, `#:maybe`, or `#:many`). The fields' contents are returned in declaration order.

```
(set-term-fields ast lst) → strategic?
  ast : strategic?
  lst : (listof (or/c strategic? #f list?))
```

A `gen:strategic` generic method. Like `term-fields`, but functionally sets the field contents specified by list `lst` instead of extracting them from the term `ast`. Returns the modified term.

4.1.1 Lenses

A functional *lens* can serve as a composition-friendly abstraction for de- or re-construction of AST nodes (or views). A lens into the term content of a concrete AST node can be instantiated in terms of the `term-fields` and `set-term-fields` operations. Similarly named operations are also available for views, enabling lens instantiation for the term content of a given view. Illusyn itself includes no lens abstraction, but third-party APIs for lenses do exist.

4.2 Visits

Illusyn includes combinators that build visiting rather than rewriting strategies. In our terminology, a *visit* is a strategy that does not rewrite; it is hence more efficient, as terms do not need to be reconstructed. No return values are checked during a visit, as a visit is only done for its side effects.

```
(all-visitor s) → (-> strategic? any/c)
  s : (-> strategic? any/c)
```

A combinator that creates a visiting operation that applies the rule `s` to all the sub-terms of the term passed to it. No rewriting takes place; the rule is applied only for its side effects.

```
(topdown-visitor s) → (-> strategic? any/c)
  s : (-> strategic? any/c)
```

Constructs a strategy that applies the function `s` depth-first, once per node, to every node in the tree.

```
(bottomup-visitor s) → (-> strategic? any/c)
  s : (-> strategic? any/c)
```

Like `topdown-visitor`, but traverses the tree in bottom-up order.

4.3 Rewriting Strategies

```
(require magnolisp/strategy-stratego)
```
package: magnolisp

Illusyn includes a collection of combinators for building rewriting strategies; the API bears a close resemblance to that of Stratego, in part to facilitate fairly direct reuse of strategies appearing in literature.

A *strategy* is a function that either rewrites an AST (and returns the rewritten AST) or fails at doing so (returning a value indicating failure). The value `#f` is used to indicate strategy failure.

For related discussion on Scheme-based implementation of generic traversal strategies, see Chapter 5 of Pankaj Surana's dissertation (Surana 2006).

4.3.1 Primitive Strategies

The most primitive strategies are neither concerned with the structure of concrete terms, nor are they *strategy combinators* (i.e., higher-order strategies that compose more complex strategies out of simpler ones). Such strategies include `fail-rw` and `id-rw`.

```
(fail-rw ast) → #f
  ast : strategic?
```

A strategy that always fails.

```
(id-rw ast) → strategic?
  ast : strategic?
```

The identity strategy. I.e., a strategy that just returns its argument.

4.3.2 Primitive Strategy Combinators

Some combinators in the library are concerned with the structure of concrete terms. These are implemented in terms of the `gen:strategic` generic interface.

```
(all s) → (-> strategic? (or/c strategic? #f))
  s : (-> strategic? (or/c strategic? #f))
```

A combinator that creates a strategy that applies the rule `s` to all the sub-terms of the term passed to it. The combined strategy fails unless the rule is applicable to all the sub-terms. (If there are no sub-terms, then it succeeds.)

```
(some s) → (-> strategic? (or/c strategic? #f))
  s : (-> strategic? (or/c strategic? #f))
```

A combinator that creates a strategy that applies the rule `s` to all the sub-terms of the term passed to it. The combined strategy fails unless the rule is applicable to at least one of the sub-terms.

```
(one s) → (-> strategic? (or/c strategic? #f))
  s : (-> strategic? (or/c strategic? #f))
```

A combinator that creates a strategy that tries to apply the rule `s` to at least one of the sub-terms of the term passed to it. Only the first sub-term to which `s` applies is rewritten. The combined strategy fails if `s` applies to none of the sub-terms.

4.3.3 High-Level Strategy Combinators

Higher-level combinators are no longer concerned with concrete data structures, as they leave such concerns to lower-level strategies that they are combining. Only some of the more commonly used ones are documented here.

```
(<* s ...)
```

Sequential composition. A macro that builds a strategy that applies the sub-strategies `s ...` in sequence, or fails. Defined as a macro so as to avoid evaluating unneeded argument expressions `s`.

```
(<+ s ...)
```

Deterministic choice. A macro that builds a strategy that applies the sub-strategies `s ...` in sequence until one of them succeeds. If none succeed, then the overall strategy fails.

```
(rec x s)
```

Recursive closure. A macro that builds a strategy as given by the expression `s`, but such that `x` is bound to the result of the expression `s` in the scope of the expression `s`. The expression `s` is evaluated once.

For example, the expression `(rec x (<* s (all x)))` specifies a top-down traversal strategy.

```
(try s) → (-> strategic? strategic?)
  s : (-> strategic? (or/c strategic? #f))
```

A combinator that constructs a strategy that tries the rewrite rule `s`, restoring the original term on failure.

```
(where s) → (-> strategic? (or/c strategic? #f))
  s : (-> strategic? (or/c strategic? #f))
```

A combinator that constructs a strategy that tries the rewrite rule `s`, restoring the original term on success, and failing otherwise.

```
(when-rw p s) → (-> strategic? (or/c strategic? #f))
  p : (-> strategic? any/c)
  s : (-> strategic? (or/c strategic? #f))
```

A combinator that constructs a strategy that applies the predicate `p`, and if it holds, then applies rewrite rule `s`. If `p` does not hold, then the original term is retained.

```
(repeat s) → (-> strategic? strategic?)
  s : (-> strategic? (or/c strategic? #f))
```

A combinator that constructs a strategy that applies rewrite rule `s` for as many times as it succeeds. Returns the resulting term, which will be the original term if `s` was not applicable even once.

```
(topdown s [an-all]) → (-> strategic? (or/c strategic? #f))
  s : (-> strategic? (or/c strategic? #f))
  an-all : (-> strategic? (or/c strategic? #f)) = all
```

Constructs a strategy that traverses its argument term top-down (from the root to the leaves), applying rewrite rule `s` once per node, either until it fails to apply to some node, or until it has been applied to every node in the tree. Traversal is performed as specified by the `an-all` strategy.

```
(bottomup s [an-all]) → (-> strategic? (or/c strategic? #f))
  s : (-> strategic? (or/c strategic? #f))
  an-all : (-> strategic? (or/c strategic? #f)) = all
```

Like `topdown`, but traverses the tree in bottom-up order (from the leaves to the root).

For example:

```
> (let ()
    (define rw
      (bottomup
        (lambda (ast)
          (match ast
           [(Inc (Lit a (? number? n)))
            (Lit a (+ n 1))]
           [else ast]))))
    (rw (Inc (Add `(,(Inc (Inc five))
                    ,(Inc five))))))
(Inc (Add ((Lit 7) (Lit 6))))
```

```
(downup s [an-all]) → (-> strategic? (or/c strategic? #f))
  s : (-> strategic? (or/c strategic? #f))
  an-all : (-> strategic? (or/c strategic? #f)) = all
```

A combination of `topdown` and `bottomup` such that it applies `s` both ways of the traversal.

```
(oncetd s [a-one]) → (-> strategic? (or/c strategic? #f))
  s : (-> strategic? (or/c strategic? #f))
  a-one : (-> strategic? (or/c strategic? #f)) = one
```

Traverses top-down until `s` successfully applies to a term, or fails if it does not apply to anything. Traversal is performed as specified by the `a-one` strategy.

4.4 Sub-Term Pruning Strategies

`(require magnolisp/strategy-term)` package: `magnolisp`

Some traversals are such that it is unnecessary to proceed further inside certain sub-terms once they have been encountered, as would happen if using `topdown` or `topdown-visitor`. For this reason the library includes `topdown-break` and `topdown-visit-break` variants of the above (as inspired by Rascal's `top-down-break visit` expression), during which one can prune sub-term traversals by invoking `break`.

```
(topdown-break s) → (-> strategic? any/c)
  s : (-> strategic? any/c)
```

Similar to `topdown`, but allows `(break expr)` to appear in the node-rewriting rule `s`, where `expr` must evaluate to a value of type `(or/c strategic? #f)`. The evaluation of `s` stops immediately after any `(break expr)`, with an undefined result; if no `break` gets invoked, then `s` must evaluate to a value of type `(or/c strategic? #f)` as usual.

```
(topdown-visit-break s) → (-> strategic? any/c)
  s : (-> strategic? any/c)
```

Similar to `topdown-visitor`, but allows `(break)` to appear in the node-visiting rule `s`. The evaluation of `s` stops immediately after any `(break)`, with an undefined result.

For example:

```
> (define-ast Dec () ([#:just exp]))

> (let ()
    (define visit
      (topdown-visit-break
        (lambda (ast)
          (printf "visited ~s\n" ast)
          (when (Inc? ast)
            (break)))))
    (visit (Dec (Add `(,(Inc (Add `(,five ,five)))
                       ,(Inc five))))))
visited (Dec (Add ((Inc (Add ((Lit 5) (Lit 5)))) (Inc (Lit 5)))))
visited (Add ((Inc (Add ((Lit 5) (Lit 5)))) (Inc (Lit 5))))
visited (Inc (Add ((Lit 5) (Lit 5))))
visited (Inc (Lit 5))
```

```
(break maybe-expr)

maybe-expr =
           | expr
```

Breaks out of a rewrite/visit rule with the result given by `expr` as appropriate, without failing the overall strategy (unless `expr` is given and evaluates to `#f`). Afterwards the traversal continues so that it does not further descend into the sub-term, but rather continues with any remaining sibling. An `expr` must be provided when traversing with `topdown-break`, and must not be provided when traversing with `topdown-visit-break`.

5 Examples

Example code demonstrating the use of the Illusyn library can be found in the Magnolisp source code distribution. Notable files in the distribution include:

- The `"ir-ast.rkt"` file shows how to define AST node types. In addition to your typical definitions, it also includes examples of the use of the `#:custom-write` and `#:singleton` options for `define-ast`.
- The `"parse.rkt"` file provides a multitude of examples of the construction of AST nodes, using constructors defined by `define-ast`. Note that the code uses some helper functions defined in the `"ir-ast.rkt"` file in order to preserve location information found in Racket's native syntax objects.
- The `"backend-cxx-main.rkt"`, `"compiler-api.rkt"`, `"ir-transform.rkt"`, and `"type-infer.rkt"` files in particular provide examples of program transformations implemented in terms of strategic term rewriting, as supported by the `magnolisp/strategy` and `magnolisp/strategy-stratego` modules' API.
- The `"modbeg.rkt"` file demonstrates the use of the `syntactifiable-mkstx` generic method, for purposes of exporting information about Magnolisp definitions via a Racket submodule.

Bibliography

Pankaj Surana. Meta-Compilation of Language Abstractions. PhD dissertation, Northwestern University, 2006.

Eelco Visser. Strategic Pattern Matching. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pp. 30–44, 1999.