slist

Release 2018.02

Jun 05, 2019

Contents

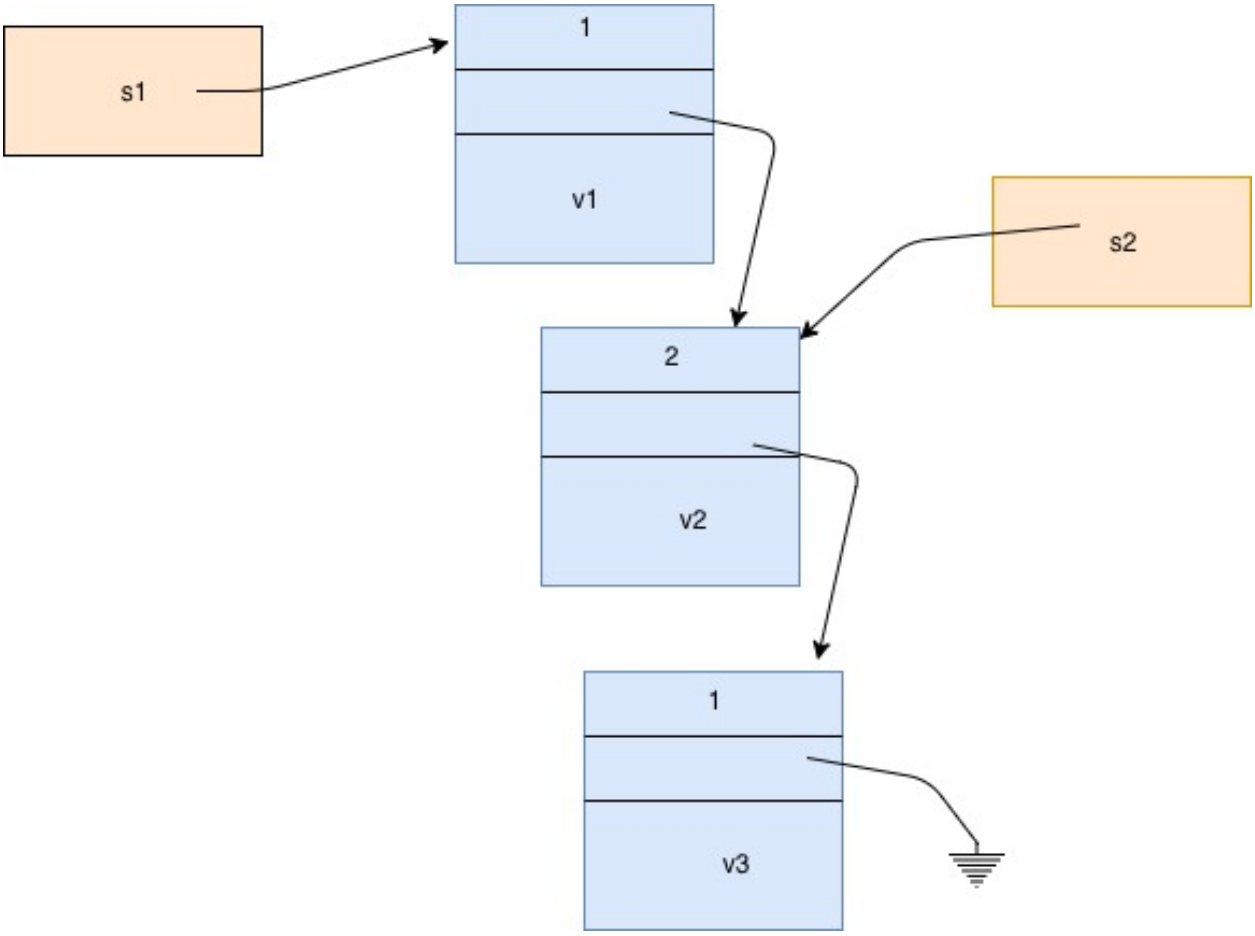
THIS IS A WORK IN PROGRESS!

Contents:

CHAPTER 1

Slist Design

An slist is a purely functional, abstractly immutable, singly linked list. A smart pointer is used to maintain the abstraction over a list of reference counted list nodes.



Dynamic uniqueness detection is used to optimise list operations by internally modifying lists when a function has exclusive ownership.

Slist has a strong property that a read iterator cannot be invalidated and always refers to the sequence of elements in the list at the time of the iterators construction.

CHAPTER 2

Interface

2.1 empty

```
empty(x)
x.empty()
```

Returns true if the list x is empty, false otherwise.

2.2 Default Constructor

```
slist<T>()
```

Is an empty list to which can be added elements of type T.

2.3 singleton

```
slist(v)
```

Returns a unique list of one element v.

2.4 slist from initializer list

```
slist({1,2,3,4})
```

Precondition: All the elements in the list must have the same type.

2.5 slist_from_container

```
slist_from_container (C)
```

Requires C be an STL container with a begin() method returning an input iterator and an end() method returning an end iterator. Returns a list of all the elements in the container in the sequence found from the iterator.

2.6 slist_from_iterators

```
slist_from_iterators<T> (begin,end)
```

Requires begin, end be valid iterators for a sequence. Returns all the values in the range of the begin iterator up to but excluding the end iterator. T::value_type must specify the container value type.

2.7 size

```
size(x)
x.size()
```

Returns the length of the list x.

2.8 uniq

```
uniq(x)
x.uniq()
```

Returns true if x s empty or is the only reference to the underlying list and all tails thereof. Implies the reference counts of all nodes of the underlying list are 1.

2.9 cons

```
cons (h,t)
t.cons(h)
```

returns list t with value h added to front. Unique if and only if t is unique.

2.10 head

```
head (x)
x.head()
```

Precondition non-empty list. Returns first value on the list.

2.11 tail

```
tail (x)
x.tail()
```

Precondition non-empty list. Returns list with first value removed. Unique if x is unique, may be unique even if x is not.

2.12 join

```
join (x,y)
x + y
```

Returns the list which is the concatenation of lists x and y. Unique if y is unique.

2.13 rev

```
rev (x)
```

Returns the list reversed. always unique.

2.14 copy

```
copy (x)
```

Makes a copy of the list. Always unique.

2.15 make_unique

TODO. Returns the list if it is unique, or a copy otherwise. Result is always unique.

2.16 map

```
map<U> (f,x)
x.map(f)
```

Returns a list with elements of type U, the result of applying f to each element of x. Always unique. Cost N allocations.

2.17 filter

```
filter (f,x)
x.filter(f)
```

Returns a sublist of elements of x satifying predicate f(v). Always unique.

2.18 fold_left

```
fold_left (f,init,x)
```

TODO. Uses f to fold each value of x starting at the front into init. Returns final result. f must accept two arguments, the first of type U, the type of init, and the second of type T, the type of the elements of x.

2.19 zip

```
zip(x,y)
```

TODO. Precondition, x and y have the same length. Returns a list of std::pair of corresponding element from x and y.

2.20 unzip

```
unzip(x)
```

TODO. Splits a list of pairs into a pair of lists. Precondition, the value type of x must be a std::pair.

2.21 begin

```
x.begin()
```

Returns forward list iterator starting at head of list. This iterator uses a strong pointer to the head of the list but scans the list using a weak pointer, avoiding the overhead of managing the reference count at the expense of retaining the whole list during the scan.

2.22 end

```
x.end()
```

Returns terminal fast list iterator.

2.23 begin_input

```
x.begin_input()
```

Returns input list iterator starting at the head of the list. This iterator uses a strong pointer to scan the list. Reference counts are adjusted during the scan. If the list is unique, then a scan will consume the list, freeing memory during the scan.

2.24 end_input

```
x.end_input()
```

Returns terminal input list iterator.

2.25 get_back_inserter

```
x.get_back_inserter()
```

Fetches an output iterator for the list x. The list should be unique. If not, the handle x detaches its current list and is set to a copy, making it unique. A prefix of the previous list may be lost because the head node's refcount is decremented. Note the whole list cannot be lost because it would have to be unique for that to happen, there must be at least one other reference to some non-empty suffix of the list.

Writes to the output iterator append elements to the end of the list. The pre-increment and post-increment operators have no effect. The dereference operator returns a proxy which accepts an assignment from the list value type, which causes a new node to be appended to the end of the list and the output iterator then advances.

Note, the iterator returned by this method is faster than

```
std::back_inserter(x)
```

however at present it is not fully safe. If the iterator is retained and the list shared, subsequent insertions will also be shared.

2.26 push_front

```
x.push_front(v);
```

Sematically equivalent to

```
x = x.cons(v);
```

Returns a reference to x.

2.27 push_back

```
x.push_back(v);
```

Sematically equivalent to

```
x = x + v;
```

Returns a reference to x.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search