Module Tjr_pcache.Tjr_pcache_doc

Main documentation entry point for `tjr_pcache`

Introduction

This library implements a persistent cache (pcache), i.e., an on-disk cache. This should be used with `tjr_btree` to reduce the writes going to the B-tree.

The main modules are included below.

▼ `include Persistent_list`

A persistent-on-disk list

▼ `include Pl_types`

```
type ('a, 'ptr, 'i) pl_state_ops = {
  set_data : 'a -> 'i -> 'i;
  set_next : 'ptr -> 'i -> 'i;
  new_node : 'ptr -> 'a -> 'i -> 'i;
  }
```

The persistent list state. Each node consists of data and a possible next pointer (initially None, but may be set subsequently). For `new_node`, the ptr is the ptr of the new block, and the second argument is the data.

NOTE the type `'a` is the type of the data stored in each node.

Type variables:

- `'a` the type of data stored in pl nodes
- `'ptr` the type of pointers to nodes; each node has an optional next pointer
- `'i` the internal state of the persistent list FIXME rename?

```
type ('a, 'ptr, 't) pl_ops = {
  replace_last : 'a -> (unit, 't) Tjr_monad.Types.m;
  new_node : 'a -> ('ptr, 't) Tjr_monad.Types.m;
  }
```

The operations provided by the persistent list.

- `replace_last` replaces the contents of the last element of the list.
- `new_node` allocates a new node at the end of the list and makes it the "current" node.

```
val make_persistent_list : monad_ops:'t Tjr_monad.Types.monad_ops ->
  pl_state_ops:('a, 'ptr, 'i) pl_state_ops -> write_node:('i ->
  (unit, 't) Tjr_monad.Types.m) -> with_pl:('i, 't) Tjr_monad.Types.with_state
  -> alloc:(unit -> ('ptr, 't) Tjr_monad.Types.m) ->
  ('a, 'ptr, 't) Pl_types.pl_ops
```

```
val pl_to_nodes : read_node:('ptr -> 'blks -> 'a * 'ptr option) -> ptr:'ptr ->
  blks:'blks -> ('ptr * ('a * 'ptr option)) list
```

Unmarshal a persistent list to a list of nodes.

```
val pl_to_list : read_node:('ptr -> 'blks -> 'a * 'ptr option) -> ptr:'ptr ->
  blks:'blks -> 'a list
```

Convenience to unmarshal to a list of node contents

▼ **include** Persistent_chunked_list

Implement a "chunked" list (multiple items per node) using Persistent_list. Automatically create a new node when the current fills up.

NOTE not concurrent safe; access must be serialized.

▼ **include** Pcl_types

```
type ('pl_data, 'e, 'i) pcl_state_ops = {
  nil : unit -> 'i;
  snoc : 'i -> 'e -> [ `Error_too_large | `Ok of 'i ];
  pl_data : 'i -> 'pl_data;
  }
```

Pure interface for manipulating the `pcl_state`. Type vars:

- `'i` the internal pcl state type (kept abstract)
- `'e` the non-marshalled element type
- `'pl_data` the type of data stored in a persistent list node (there is also a pointer in the pl node)

Functions:

- `nil` the empty state corresponding to a new node created when the old node is full; use pl_data to get the underlying pl_node
- `snoc` to add an element
- `pl_data` to project from the pcl state to the actual data to be written

NOTE that the "next" pointer manipulation has been confined to the persistent list interface.

```
type 'ptr inserted_type =
| Inserted_in_current_node
| Inserted_in_new_node of 'ptr
```

A type that records whether an element was inserted in the current node, or whether a new node was allocated to hold the element.

```
type ('e, 'ptr, 't) pcl_ops = {
  insert : 'e -> ('ptr inserted_type, 't) Tjr_monad.Types.m;
  }
```

The interface exposed by the persistent chunked list, a single `insert` function. NOTE how 'pl_data and 'i have disappeared.

```
val make_pcl_ops : monad_ops:'t Tjr_monad.Types.monad_ops -> pl_ops:
  ('pl_data, 'ptr, 't) Pl_types.pl_ops -> pcl_state_ops:
  ('pl_data, 'e, 'i) pcl_state_ops -> with_pcl:
  ('i, 't) Tjr_monad.With_state.with_state -> ('e, 'ptr, 't) pcl_ops
```

Function to construct a persistent chunked list. Parameters:

- `pl_ops` The underlying persistent list operations.
- `pcl_state_ops, with_pcl` For the internal state of the pcl.

```
val pcl_to_nodes : read_node:('a -> 'b -> 'c * 'a option) -> ptr:'a -> blks:'b
  -> ('a * ('c * 'a option)) list
```

This is just `pl_to_nodes`

```
val pcl_to_es_node_list : read_node:('ptr -> 'blks -> 'e list * 'ptr option) ->
  ptr:'ptr -> blks:'blks -> ('ptr * ('e list * 'ptr option)) list
```

As `pcl_to_nodes`

```
val pcl_to_elt_list_list : read_node:('a -> 'b -> 'e list * 'a option) -> ptr:'a
  -> blks:'b -> 'e list list
```

Drop pointers from `pcl_to_es_node_list`

▼ **include** Detachable_chunked_list

A "detachable list", with an operation `detach` to drop everything but the current node.

NOTE this code is not concurrent safe. Access must be serialized.

▼ **include** Dcl_types

The DCL types

```
type ('op, 'abs) abs_ops = {
  empty : 'abs;
  add : 'op -> 'abs -> 'abs;
  merge : 'abs -> 'abs -> 'abs;
  }
```

Operations on the "abstract" state. The pcl state is something like a list of map operations. The 'abs type is something like the "map" view of these operations (needed because the list is redundant, or at the very least inefficient for map operations). Type vars:

- `'op` is eg insert(k,v), delete(k)
- 'abs is the "abstraction"

Operations:

- empty, the empty map

- add, to add an operation to the abstract state
- merge, to merge two maps (second takes precedence); used when a new node is created, and the old node is merged into the accumulated past nodes.

```
val abs_singleton : abs_ops:('a, 'b) abs_ops -> 'a -> 'b
```

```
type ('ptr, 'abs) dcl_state = {
  start_block : 'ptr;
  current_block : 'ptr;
  block_list_length : int;
  abs_past : 'abs;
  abs_current : 'abs;
  }
```

The state of the DCL. Fields are:

- `start_block` is the root of the log
- `current_block` is the current block being written to
- `abs_past` is the abstract view of ops from root to just before `current_block`
- `abs_current` is the abstract view of ops for the current block

NOTE unlike Pl and Pcl, we have a concrete type for the state, since we don't expect to have any extra info stored at this point. (FIXME what about dcl_dummy_implementation where we need to store all ptrs?) But perhaps we can avoid some of these extra type params if we keep dcl state abstract as `'dcl_state`. But this seems unlikely.

NOTE `block_list_length`: this is the number of blocks from the underlying chunked list, used to store the ops (not the abstract representation!)

NOTE upto this point, the pl and the pcl have not explicitly tracked the start of the list. FIXME perhaps they should? This has advantages in that the abstraction is self-contained.

```
type ('op, 'abs, 'ptr, 't) dcl_ops = {
  add : 'op -> (unit, 't) Tjr_monad.Types.m;
  peek : unit -> (('ptr, 'abs) dcl_state, 't) Tjr_monad.Types.m;
  detach : unit -> (('ptr, 'abs) dcl_state, 't) Tjr_monad.Types.m;
  block_list_length : unit -> (int, 't) Tjr_monad.Types.m;
  }
```

The DCL ops:

- `add` to add an op
- `peek` to reveal the dcl_state (FIXME why?)
- `detach` to issue a detach operation (eg prior to rolling the past entries into a B-tree)
- `block_list_length` to provide information to help determine when to roll up

The `detach` operation means that we should start a new cache from the current block.

The return result is the ptr and map corresponding to the contents of everything up to the current block, and the ptr and map for the current block. The intention is that the detached part is then rolled into the B-tree. If we only have 1 block, then nothing is rolled up. This occurs when `old_ptr` is the same as `new_ptr` FIXME use a new type

NOTE detach returns the dcl_state since this includes at least all the fields we need.

```
val make_dcl_ops : monad_ops:'t Tjr_monad.Types.monad_ops -> pcl_ops:
  ('op, 'ptr, 't) Pcl_types.pcl_ops -> with_dcl:
  (('ptr, 'abs) dcl_state, 't) Tjr_monad.With_state.with_state -> abs_ops:
  ('op, 'abs) abs_ops -> ('op, 'abs, 'ptr, 't) dcl_ops
```

Construct the dcl operations. Parameters:

- `monad_ops`, the monadic operations
- `pcl_ops`, the persistent chunked list ops
- `with_dcl`, access the dcl state
- `abs_ops`, operations on the abstract data (see Detachable_map for an example where the abstract data is a map)

▼ **`include`** `Detachable_map`

As Detachable_chunked_list, but with the operations insert and delete; and the abstract view a map.

▼ **`include`** `Dmap_types`

A dmap is effectively just a DCL with a refined 'op type and 'abs type. However, we also include functionality to convert to a standard map interface (not one based on ops).

```
type ('k, 'v) op = ('k, 'v) Ins_del_op_type.op
```

```
type ('k, 'v) op_map = ('k, ('k, 'v) op) Tjr_polymap.t
```

Abbreviation; FIXME move to Ins_del_op_type

```
type ('ptr, 'k, 'v) dmap_state = ('ptr, ('k, 'v) op_map) Dcl_types.dcl_state
```

NOTE dmap_state is just an abbreviation for dcl_state

```
val internal_ : ptr:'ptr -> abs:('k, ('k, 'v) op) Tjr_polymap.t -> unit
```

```
type ('ptr, 'k, 'v, 't) dmap_dcl_ops =
  (('k, 'v) op, ('k, 'v) op_map, 'ptr, 't) Dcl_types.dcl_ops
```

NOTE dmap_dcl_ops is just an abbreviation for dcl_ops with:

- 'op the type of kv op
- 'abs the type of kv op_map

```
type ('k, 'v, 'ptr) detach_info = {
  past_map : ('k, 'v) op_map;
  current_map : ('k, 'v) op_map;
  current_ptr : 'ptr;
  }
```

The result of "detaching" the map. We get the abstract map for all but the current node, and information about the current node.

```
type ('k, 'v, 'ptr, 't) dmap_ops = {
  find : 'k -> ('v option, 't) Tjr_monad.Types.m;
  insert : 'k -> 'v -> (unit, 't) Tjr_monad.Types.m;
  delete : 'k -> (unit, 't) Tjr_monad.Types.m;
  detach : unit -> (('k, 'v, 'ptr) detach_info, 't) Tjr_monad.Types.m;
  block_list_length : unit -> (int, 't) Tjr_monad.Types.m;
  }
```

For the detach operation, we get the map upto the current node, and the map for the current node

```
val make_dmap_dcl_ops : monad_ops:'t Tjr_monad.Types.monad_ops -> pcl_ops:
  (('k, 'v) Ins_del_op_type.op, 'ptr, 't) Pcl_types.pcl_ops -> with_dmap:
  (('ptr, 'k, 'v) dmap_state, 't) Tjr_monad.With_state.with_state ->
  ('ptr, 'k, 'v, 't) dmap_dcl_ops

val convert_dcl_to_dmap : monad_ops:'a Tjr_monad.Types.monad_ops ->
  dmap_dcl_ops:
  (('b, 'c) Ins_del_op_type.op, ('b, 'c) op_map, 'd, 'a) Detachable_chunked_list.dcl_ops
  -> ('b, 'c, 'd, 'a) Dmap_types.dmap_ops
```