Technical notes

Those are a collection of technical notes about a specific topic in Lix.

- Pointer equality
- Darwin debugging for the Linux developer

Pointer equality

Introduction

This page dives into the concept of pointer equality, its role in Lix, and provides an outlook on how it is implemented and utilized.

For more in-depth information, refer to the resource on https://snix.dev/docs/reference/nix-language/value-pointer-equality/ from the Snix project.

Some background on pointer equality

This section provides a foundational overview of pointer equality, specifically in the context of nonstrict languages like Nix.

Why pointer equality is used?

Pointer equality is a "low-level" operation that checks whether two values are represented by the same memory reference, that is, whether they point to the same object in memory. In a lazy functional language implementation (such as Lix, Haskell or Lean), pointer equality is often much cheaper than structural equality, which may require traversing the entire data structure. Traversing terms can be slow (exponential) when doing structural comparisons, (recursive) pointer equality offer a linear time approach to structural comparison.

In Nixpkgs, `lib.systems.equals` implements the expensive structural comparison. In Lix, `==` relies on pointer equality in underspecified fashions.

The relation of pointer equality and call-by-need evaluation strategy

Pointer equality is closely tied to the evaluation strategy employed by an interpreter.

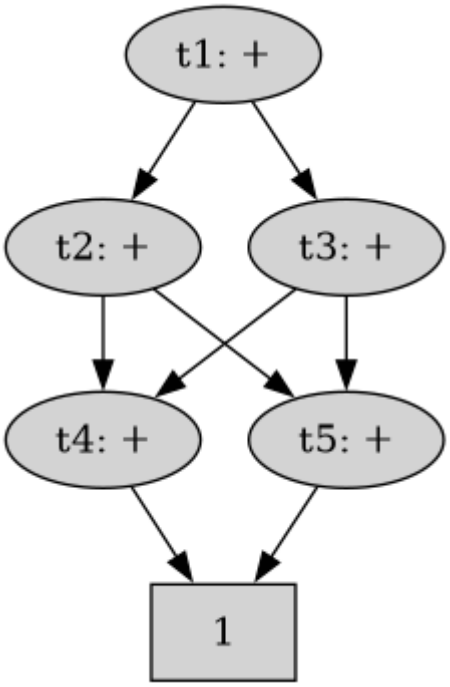
There are usually three main evaluation strategies:

- Call-by-value
- Call-by-name
- Call-by-need

A **call-by-need** strategy (also known as **lazy** or **nonstrict** evaluation) evaluates values **only when needed**, unlike **call-by-value**, which evaluates values eagerly, or **call-by-name**, where values are re-evaluated each time they are used.

In call-by-need, values are evaluated **at most once**, and the result is reused whenever the value is required again.

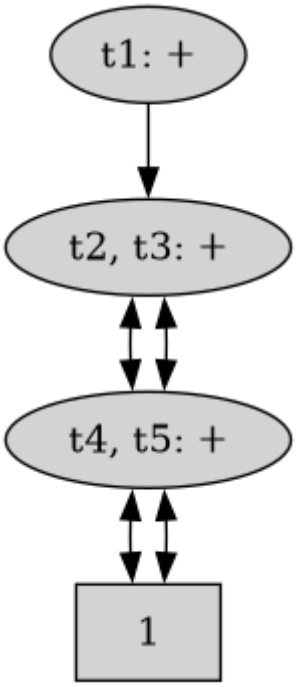
This has important implications for **pointer equality**. To have effective pointer equality, it's crucial to optimize when sharing occurs between structurally equal expressions. For example:



In this diagram, the terms represent a call to the binary addition operation `+` and one operand `1` is at the bottom. Notice that terms at the same level evaluate to the same result.

If we ask whether `t_2` is equal to `t_3` without considering sharing, pointer equality would initially return `false`, as it compares only the "tip" of the structure. However, through recursive pointer equality, the comparison can descend into the structure, testing further until it finds that the two terms ultimately share the same operand (`1`).

That being said, an interpreter can optimize this by merging shared substructures:



This maximally shared structure leads to more efficient pointer equality checks between terms that are actually equal, reducing redundant evaluations.

But, these representations are internal to the evaluator and should not lead to a language evaluating in a way or another, they are purely about optimization.

Unfortunately, pointer equality can be a primitive for detecting shared structures in structurally equal objects, causing the internal details of the interpreter to leak into the language.

The situation in Lix

Lix is a **call-by-need** evaluator of a nebulous — underspecified — language, often called Nix or Nixlang.

General comparison theory in Lix

If we ignore pointer equality and focus on semantic equality (i.e. whether the contents of the values are equal), we can draw this summary table since Lix 2.90:

Type (`v1.type()`)	Structural comparison with same type?	Comparison semantics	Notes
`nInt`	Yes	Compare integer values (`==`)	Exact numeric equality
`nBool`	Yes	Compare boolean values (`==`)	True/false match
`nString`	Yes	Compare string contents (`==`)	Structural string equality
`nPath`	Yes	Compare underlying string contents (`content->str()`)	Path equality is based on string value

Type (`v1.type()`)	Structural comparison with same type?	Comparison semantics	Notes
`nNull`	Yes	Always `true`	All null values are equal
`nList`	Yes	Compare element-wise recursively	Compare elements one by one
`nAttrs`	Yes	Compare attribute names and values recursively	Twist: Derivations are compared by their `outPath` attribute
`nFunction`	**No**	Always `false`	Functions always compare not-equal
`nExternal`	Yes	Delegate to `*external == *external`	Uses external type's own equality definition
`nFloat`	Yes	Compare float values (`==`)	Standard floating-point equality (e.g., `NaN != NaN`)

The case of functions

As shown in the previous table, functions always compare as **not equal**. However, when using pointer equality, `f == g` can evaluate to **true**.

In Lix, pointer equality introduces an **irreflexive** binary relation for functions: that is, `(x: x) == (x: x)` is always **false**.

There are several reasons why functions consistently return **not equal**:

- When defining a function like `(x: x)`, it creates a **new** identity each time. Binding this function and reusing it elsewhere preserves this identity. Thus, pointer equality can cause comparisons to return **equal**.
- Comparing functions in a lambda calculus-inspired language can be complex and is not a feature we intend to offer to users (for a deep dive into the topic, see this paper if you're still interested). It has limited practical value.

Some members of the Lix core team currently believe that allowing function comparisons at all was a **grave mistake**, as it encouraged users to mix **code** with **data** by placing functions into data structures.

The team envisions a future where function comparisons are entirely removed from the language, making any function comparison an error. The next section explores why achieving this goal will be challenging.

Where is pointer equality used in the Lix ecosystem?

The Lix codebase refers to a mysterious `builderDefs` to explain why pointer equality between attribute sets (including those with functions) is supported. This is illustrated in the following code:

```
    /* !!! Hack to support some old broken code that relies on pointer
       equality tests between sets.  (Specifically, builderDefs calls
       uniqList on a list of sets.)  Will remove this eventually. */
    auto pointerEq = [&] { return v1.pointerEqProxy() == v2.pointerEqProxy(); };
```

In reality, `builderDefs` has long since disappeared, but this comment was never updated to reflect that.

The primary use case for this specific form of pointer equality comes from the cross-compilation and platform machinery in nixpkgs. For example, the expression `pkgs.stdenv.hostPlatform == pkgs.stdenv.buildPlatform` performs a **function** pointer equality comparison.

To elaborate: `pkgs.stdenv.hostPlatform.emulator` is a function, and there are other functions within the set.

As a result, any changes to this semantics could cause issues within the Nixpkgs platform machinery, potentially leading to unexpected behavior, such as entering cross-compilation mode when it shouldn't. More details on this will follow in the next sections.

The semantics of pointer equality in Nix 2.18

As of Nix 2.18, semantics for pointer equality were to apply it for any pair of values, no matter their types, i.e. in pseudo C++:

```
bool EvalState::eqValues(Value * v1, Value * v2, const PosIdx pos, std::string_view errorCtx)
{
    forceValue(v1, pos);
    forceValue(v2, pos);

    // This is where the pointer equality intervened by comparing the addresses of the Value
pointers.
    if (v1 == v2) return true;

    // Special case type-compatibility between float and int
    if (v1.type() == nInt && v2.type() == nFloat) {
        return v1.integer().value == v2.fpoint();
    }
    if (v1.type() == nFloat && v2.type() == nInt) {
        return v1.fpoint() == v2.integer().value;
    }

    // All other types are not compatible with each other.
    if (v1.type() != v2.type()) return false;
```

```
    // [snip]
}
```

The semantics of pointer equality in Lix `80654b84b610f4c0622dd10f0af78a8a2ce97048`

Commit link: `80654b84b610f4c0622dd10f0af78a8a2ce97048`

This commit introduced a higher level of sharing of the expressions in our internal implementation, while we were careful not to induce a change in visible semantics, we decided to reduce the eligibility of the pointer equality check above to only: list, attributes and external (plugin) objects.

This resulted in various evaluation regressions, e.g. `nix eval "github:nixos/nixpkgs?rev=a999c1cc0c9eb2095729d5aa03e0d8f7ed256780#pkgsCross.gnu64.bitwarden" --no-eval-cache`.

The root cause lies in the following and is related to function comparisons:

```
with rec {
  a = {
    f = x: x;
    meow = true;
  };
  b = a // {
    meow = true;
  };
};
a == b
```

returned false.

This pattern occurs more generally in the Nixpkgs module system when deciding whether a package set is in cross mode or not:

```
let
  inherit (import <nixpkgs> { }) lib;
in
(lib.evalModules {
  modules = [
    (
      { config, ... }:
      {
        options = {
```

```
          buildPlatform = lib.mkOption {
            type = lib.types.either lib.types.str lib.types.attrs;
            apply = lib.systems.elaborate;
            default = config.hostPlatform;
          };
          hostPlatform = lib.mkOption {
            type = lib.types.either lib.types.str lib.types.attrs;
            apply = lib.systems.elaborate;
            default = "x86_64-linux";
          };
          isCross = lib.mkOption { type = lib.types.bool; };
        };
        config = {
          isCross = config.buildPlatform == config.hostPlatform;
        };
      }
    )
  ];
}).config.isCross
```

This can be tested this way as well:

```
nix-repl> nixosConfigurations.nixos.config.nixpkgs.hostPlatform ==
nixosConfigurations.nixos.config.nixpkgs.buildPlatform
```

(Credits to aloisw for the code snippets.)

Circling back to "where is pointer equality used?", the answer is that pointer equality is very used with functions in attribute sets.

The semantics of pointer equality in Lix 2.94.0

Our plan is to implement https://gerrit.lix.systems/c/lix/+/4556.

This makes again many types eligible to pointer equality checks, repairing the previous evaluation issue. A new behavior occurs now due to sharing changes:

```
let a = { f = x: x; }; in a.f == a.f
```

is now true.

At the time of writing, we believe this is the right thing to do as this object `f` has the correct identity now.

Open questions

- How does making more objects have an identity is going to hold in the future of the Nixlang?
- What about the goal of evaluating old Nixpkgs and providing stability?

Darwin debugging for the Linux developer

These technical notes are meant to explain to a Linux developer what are the equivalents to debug certain things over Darwin.

Key differences from Linux

Linux tool	macOS equivalent	Purpose
`strace`	`dtruss` (1), `strace-macos`	System call tracing
`perf`	`sample`, `spindump`, `Instruments` (2)	Performance profiling
`ss`	`netstat -nv`	Socket statistics
`iostat`	`iostat -Id`	Disk I/O statistics
`lsof`	`lsof`	Open file listings
`/proc`	`vmmap`, `heap`, `lsof`	Process/memory information
`blktrace`	`fs_usage -wf filesys`	Trace real-time IO activity

(1): `dtruss` is unavailable on systems with System Integrity Protection enabled (SIP). Lix CI machines have SIP enabled.

(2): `Instruments` is a graphical application that requires a remote desktop and cannot be used over SSH.

Getting flamegraphs out of `sample`

Combine https://github.com/brc7/flamegraph-sample/blob/master/stackcollapse-sample.py with https://speedscope.app/.

Inspecting memory allocations

`heap` can give a list of allocations sorted by sizes when it can recognize the type in-memory (doesn't always work for everything).

`iostat` tips

`iostat -Id` on macOS will return as the first row the cumulative data *SINCE* the first boot.

`xfrs` := IO operations per second.

Listing file descriptors

`/proc` doesn't exist on macOS, therefore, `lsof` is the only thing that can list file descriptors and let you understand the relationships.

Tracing network activity

`fs_usage -wf network` can trace real-time network activity on file descriptors.

Checking Internet quality

`networkQuality -v` will dump an Apple-infrastructure Internet connection check for bandwidth and latency.