The Virtues of Values

Chris Collazo
Data & Software Engineer, GeoSonics/VibraTech

for NimConf 2026

Or: How high-performance, high-level software is *simple* with Nim's default semantics.

Languages are a careful balance of systems-awareness and abstraction.

- We want high performance and memory efficiency…
- But also powerful abstraction tools that are memory safe.
- Usually considered hard trade-offs in the language design dialectic.

The Systems Side

Memory safety is the most enduring problem in software.

- Languages with manual allocation are fraught with peril.
- Semantic differences between values and references are often vague, perennial footguns.

Most native-compiled languages settled on either GC or single-ownership.

- GC forces you to accept indeterminate lazy deallocation and pervasive reference semantics.
- Single-ownership forces you to accept either tons of smart-pointer wrappers or overbearing static analyzer.

Or do they?

The Abstractions Side

Functional languages showed how robust pure abstractions could be.

- Immutable data, values, transformations, stateless expressions.
- Enforce formally correct program structure!
- Rich Hickey: The Value of Values (https://www.infoq.com/presentations/Value-Values/)

But functional abstractions:

- Force you to accept lots of allocations and copying, lots of GC churn.
- Code cleanliness, eliminating memory semantics requires deferring cleanup to the runtime.

Or do they?

Is there an ideal systems/abstraction synthesis that pays a minimal cost?

Thesis: Yes.

Nim’s value semantics, single-ownership by default, hidden unique pointers, copy mitigation strategies, and explicit trapdoors make it **easy** to write high-performance, high-level native-compiled software.

Nim’s Value Semantics

A Small Number of Tools

- Simple, compositional type system
- Functional, compositional, transformational `std`
- `let` and immutable parameters
- `var` and `var` parameters

These are all you need to build robust software projects.

Value Semantics & Simple Types

Nearly all interactions are based on data values.

- Identity/place are hidden by default.
- Types can easily be composed (no classes, no inheritance).
- No references, pointer decay, or windows/views by default.
- `std` library operates on data value, not identity/place.
- Even for complex composite types.

This enables functional-like program structure.

...Even for Dynamic Heap Data

Dynamically-allocated types are wrapped in a **hidden unique pointer (HUP)**.

- `seq`, `string` are HUP types.
- Data is allocated on the heap, referenced by a pointer on the stack.
- Lifetime is governed by the stack frame
- Compiler can statically determine pass-by-value or pass-by-reference, avoiding copies...
- Meanwhile, we logically operate **only on the data's value**.

Functional Standard Library

Much like a functional language…

- Most `std` functions take immutable inputs, produce new values.
- *Rarely* mutate `var` inputs in-place.
- Tradeoff: Usually allocating those return values.

The benefit is simpler program logic.

- Can guarantee the inputs are not being mutated.
- Can pipeline/chain stateless transformations easily.

This enables expression- and data-oriented programming.

- Referential transparency:
  - A function call can be replaced by the value it returns.

`let`

- Immutable value.
- Single-owner: Lifetime is the declaring stack frame.
- Dynamic heap types (`seq`, `string`) managed by HUPs.
  - Data values **cannot** be modified!
- When assigned or passed, compiler checks:
  - If origin is not used again, the hidden reference is passed.
  - Otherwise, copied.
- So, a mix of `const` and `unique_ptr` in **one keyword**.
- `object` type declared using `let`:
  - All members are **immutable**.

Immutable Parameters

ex: `proc myProc(inVal: InType): OutType =`

- Copied if less than 24 bytes, otherwise passed by immutable reference.
  - Does not take ownership: The original context still owns the value.
- Passed a mutable `var`?
  - Receives an immutable reference, cannot modify data.
  - Other contexts can still modify!

`var`

- Mutable value.
- Single-owner: Lifetime is the declaring stack frame.
- Dynamic heap types (`seq`, `string`) managed by HUPs.
  - Data values **can** be modified!
- Copy-on-assignment.
- `object` type declared using `var`:
  - All members are **mutable**.

`var` Parameters

ex: `proc myProc(inVar: var InType): OutType =`

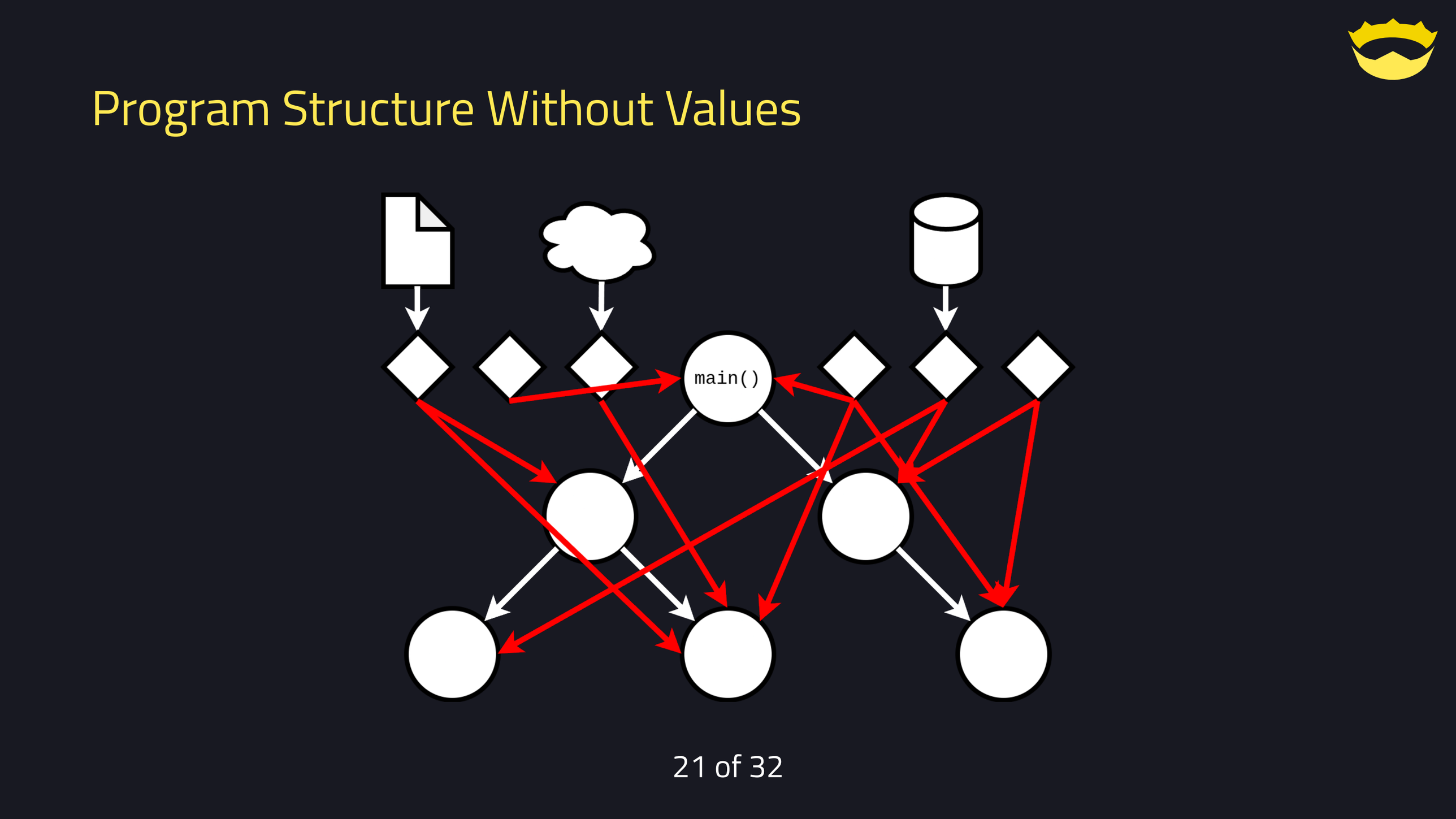
- Inputs are passed by hidden reference…
  - Only if they don’t outlive origin!
  - Does not take ownership: The original context still owns the value.
- Like a `&mut InType` in Rust or decaying a `unique_ptr<InType>` ⟶ `InType&` in C++.
- Passed an immutable `let`?
  - Compile error. Immutable values cannot be modified.

Using only these tools…

You can build native-compiled programs composed of *all* business logic.

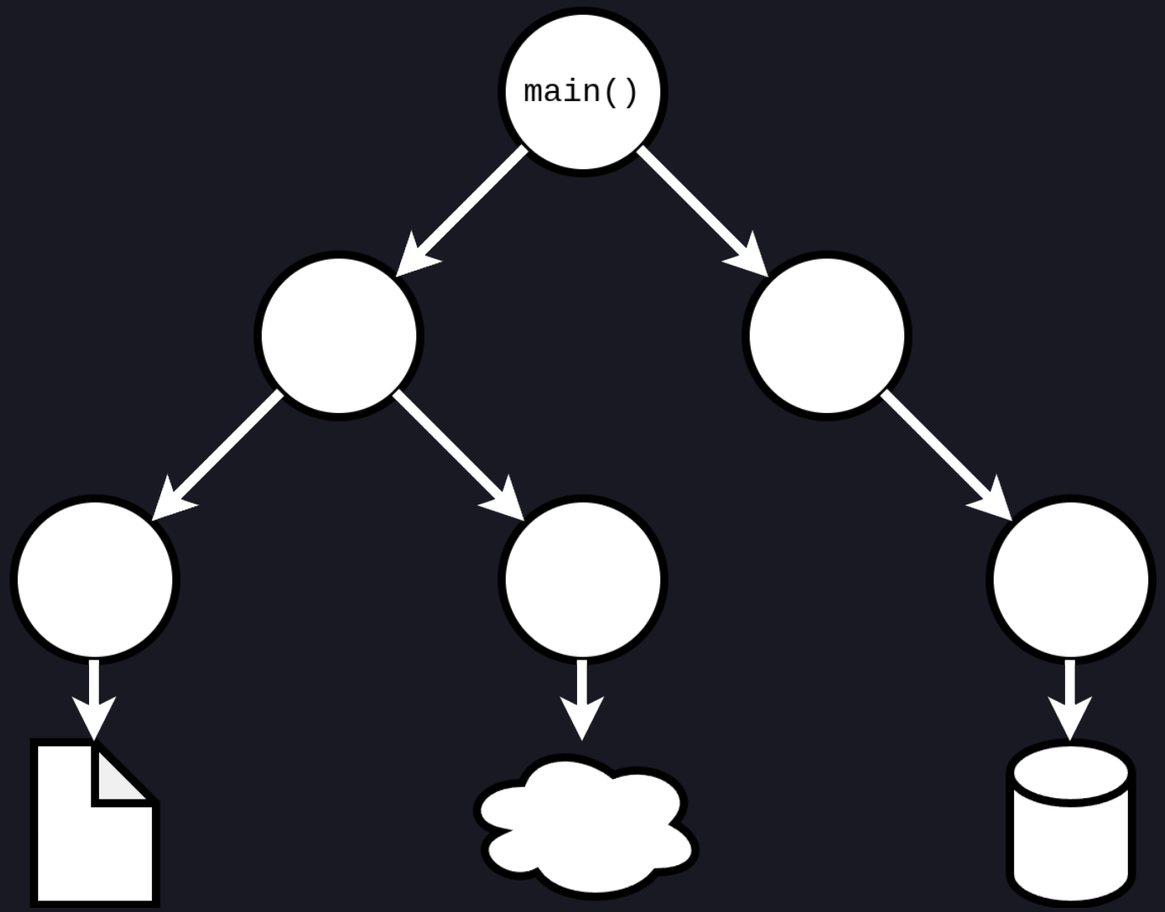
Programs that are…

- As simple to read as a scripting language, plus types.
- As cleanly structured as a functional language, plus local mutations where appropriate.
- As fast as an equivalent which would require much more effort in C++ or Rust.



Program Structure Without Values

Program Structure With Values



The Abstraction Benefits

We no longer need to worry about entire classes of semantics. Without needing any annotations or extra baggage:

- Compiler helps us enforce a simple, clean program structure.
- Lifetimes are always scoped and safe.
- Compose types to our heart's content; focus on data rather than memory.
- Rest assured that the performance and memory behavior will be *more* than "good enough".
- Trapdoors are available if absolutely necessary.

The Trapdoors

- `ref` enables reference semantics and RC.
  - Operations can act upon identity/place.
  - `ref` enables RC cleanup.
- `ptr` enables raw pointers and manual allocation.
  - Not check by the RC.
  - Tools are `UncheckedArray`, `alloc0`, and `dealloc`.
- Trivial C interop.
  - `cstring` gives us C's null-terminated strings (for C compile targets).
  - Easily pull in C's stdlib.
- All of these short-circuit implicit copying.

Values for Vibrations

How I used Nim and value semantics

- Vibration analysis engine for construction and blasting.
  - One-shot stateless executable: Record files in, JSON data or HTML reports out.
  - Clean call tree structure: Leans on immutability, referential transparency.
  - No memory semantics: No `ref` or `ptr` anywhere, only 4 (!) manual deep copies.
  - All vibration calculations, or transforming data for calculation.
  - Single-threaded: We get parallelism from parallel invocations.
  - Lots of Datamancer `DataFrame` use:
    - Even with complex internals, still immutable values by default! Stateless API.
  - Failure states use `assert` to return errors ASAP.
- Built in record time, easy to maintain, easy to add features.

A Fully Chained Example Dataflow

```
let composite_leq_interval_utc_df =
  filename.read_record_archive().sort_by_event_time().map(
    r => r.sound_meter_record
  ).filter(
    r => r.header.sample_count > 0
  ).map(
    r => r.get_scaled_samples_df().ngr_to_sound_df(r)
  ).to_local_with_details_dfs(headers).assign_stack().get_day_dfs().map(
    df => sound_level_interval_from_day_df(df, interval_minutes)
  ).assign_stack().mutate(
    fn {`time` ~ local_str_to_utc_str(`time`, timezone)}
  ).to_csv_string()
```

Nim Bonuses

- Heterogenous data structures: `std/json` and `JsonNode`
  - Despite its composite, dynamically typed variant structure…
  - Still just an immutable value by default.
  - I use it for our numerous and highly variable reporting options.
  - Can be passed down the entire call stack and queried without worry.
- Trivial compatibility thanks to C compile target.
  - To build for Arm or RISC-V:
    - Compile the Nim compiler on the target, or
    - Install a cross-compile toolchain.

Conclusion

Nim’s semantics give us safe, performant software…

- » **WHILE** preserving abstraction power.
- » **WHILE** gifting us clean syntax free of noise.
- » **WHILE** disciplining clean program structure.
- » **WHILE** explicitly marking unsafe code.

We don’t have to fight a borrow checker, learn thousands of pages of unintuitive language and library behavior, or litter our code with a mess of smart pointer noise.

Just keep it simple.

Write the most obvious, least clever code. It’ll work, and it’ll be fast.

Thanks to:

- Peter Munch-Ellingsen (PMunch)
- Mamy Ratsimbazafy (mratsim)
- Jacek Sieka (arnetheduck)
- Ico Doornekamp (zevv)

Contact me:

- @Nerve in the Nim Discord and forums
- github.com/nervecenter
- nervecenter7@gmail.com