Integrating Cost and Behavior in Type Theory

Robert Harper

Morgenstern Colloquium
October 17, 2023

Computer Science Department
Carnegie Mellon University

Acknowledgements

This talk represents joint work with

- Harrison Grodin (Carnegie Mellon)
- Runming Li (Carnegie Mellon)
- Yue Niu (Carnegie Mellon)
- Jon Sterling (Cambridge)

Sponsored by AFOSR awards A210038S0002 and A210038S0002 (Tristan Nguyen, PM) and by NSF award CCF1901381.

Thank You

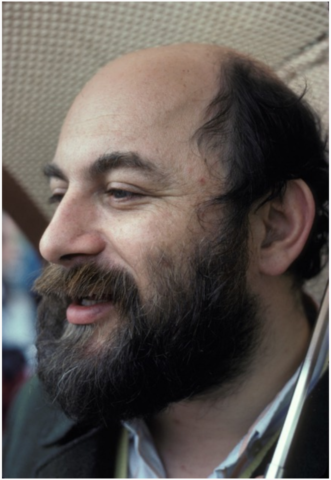
Thank you to the Morgenstern Colloquium organizers for the kind invitation!

Thank you to INRIA and to Université Côte d'Azur, Lab i3S for supporting my visit.

Thank you to Luigi Liquori for organizing and hosting.

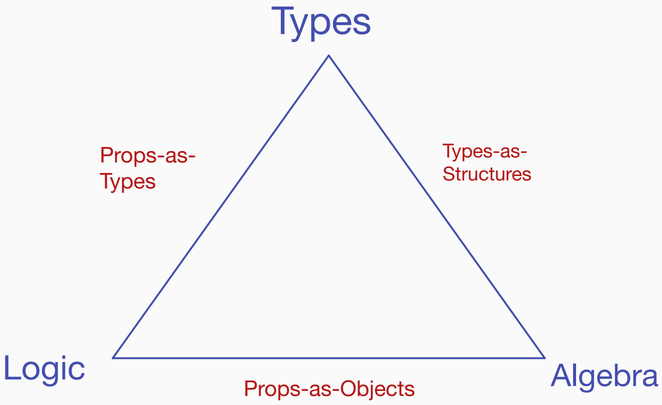
Dedication

With warm memories of Gilles Kahn, whose work inspires me to this day.



Motivation

Computational Trinitarianism



Computational Trinitarianism

Intuitionistic Type Theory has emerged as a unifying language for Mathematics, Logic, and CS.

Previously disparate notions …

- Logical consequence: $\phi_1 \,\mathsf{true}, \ldots, \phi_n \,\mathsf{true} \vdash \phi \,\mathsf{true}$.
- Maps between structures: $f : A_1 \times \cdots \times A_n \to B$.

…are unified as types of terms:

$$x_1 : A_1, \ldots, x_n : A_n \vdash e : A$$

Given proofs of $e_1 : A_1, \ldots, e_n : A_n$, the instance $[e_i/x_i]e$ is a proof of A.

Computational Trinitarianism

E.g., the proof of $\phi \supset \psi$ true because ϕ true $\vdash \psi$ true corresponds to the term

$$\lambda(x \,.\, e) : \phi \to \psi$$

with a scoped assumption of ϕ deducing ψ:

$$x : \phi \vdash e : \psi.$$

(cf exponential object in categories)

Equations specify the behavior of terms:

$$x_1 : A_1, \ldots, x_n : A_n \vdash e = e' : A$$

Intuitively, e and e' have the same behavior as programs with inputs $x_1, \ldots, x_n$ of specified type.

Equivalently, e and e' are the same proof of the proposition A.

Any usage of the assumption respects this equality!

If $x : A \vdash f : B$ *and* $e = e' : A$*, then* $[e/x]f = [e'/x]f : B$.

A new notion: when are two proofs the same?

Computational Trinitarianism

Type theory unifies propositions/proofs with the data on which they act.

E.g. Numbers, functions, algebras, spaces, paths.

Type theory is thus the grand unified theory of mathematics, logic, and computation!

Computational Trinitarianism

As per Brouwer's Intuitionism, types and their elements have computational meaning as programs.

Surprisingly, this is important even for "classical" mathematics!

The intuitionistic framework provides axiomatic freedom that is not available in classical settings such as set theory.

- May always assume that a proposition is true or not.
- May always collapse proofs to "at most exist."
- Voevodsky's univalence principle equates isomorphic structures.

Type Theory for Programming

Type theory is a natural setting for specification and verification of programs.

- $\texttt{sort} : \texttt{seq} \to \texttt{seq}$ (transforms sequences).
- $\texttt{sort} : s : \texttt{seq} \to s' : \texttt{seq} \times \texttt{sorted}(s') \times \texttt{perm}(s, s')$ (puts in order).

Standard type theory emphasizes pure behavior.

- Advantage: compatible with any choice of implementation.
- Disadvantage: useful notions of cost depend on evaluation order.

Example: Two Sorting Algorithms

We may define

- $\texttt{isort} : \texttt{seq} \to \texttt{seq}$ (insertion sort)
- $\texttt{msort} : \texttt{seq} \to \texttt{seq}$ (merge sort)

Extensionally these are equal, because they both sort!

$$\texttt{ext}\,\mathsf{true} \vdash \texttt{isort} = \texttt{msort} : \texttt{seq} \to \texttt{seq}$$

Using, say, Agda it is not difficult to verify these properties.

Intensionally these programs are rather different algorithms!

When are two proofs the same? When is one proof better than another?

Type Theory for Programming

Levy's call-by-push-value theory constrains evaluation order.

- Positive types A classify values: "data is."
- Negative types X classify computations: "programs do."
- Modalities: computations, $F(A)$, and suspensions, $U(X)$.

Pedrot's and Tabareau's ∂CBPV for the dependent case:

- Type families are indexed by value types.
- Polarity imposes order on chaos to permit effects.

Allows for effects in programs/proofs, which are essential for what follows.

Acounting for Cost

These type theories capture the behavior of programs … but what about their cost?

Want to state and prove cost bounds such as number of comparisons.

- $\texttt{isort} : \texttt{seq} \xrightarrow{n^2} F(\texttt{seq})$ (quadratic).
- $\texttt{msort} : \texttt{seq} \xrightarrow{n \lg n} F(\texttt{seq})$ (polylogarithmic).

But how can equal functions have different properties?

And what does cost even mean?

- What are we counting?
- Sequential vs parallel?

Sense and Reference

Frege distinguished sense from reference.

- Reference: what is being described.
- Sense: how it is presented.

Famously, “the morning star” vs “the evening star”, both being the planet Venus.

A similar distinction is considered here:

- Reference: a function.
- Sense: an algorithm.

Thinking of proofs as programs, we obtain a precise qualitative distinction among proofs of the same theorem.

Cost Measures

The textbook story is machine-based.

- Cost = instruction steps (or memory cells).
- Higher-order programming is never considered.
- Parallelism? Specifying *p* is a non-starter.
- There is no theory of composition of programs.

Language-based approaches are compositional, and are concerned with the code you write (not how it is compiled).

Cost Measures

Cost is not absolute (per model), but relative (per algorithm).

- Sorting: number of comparisons.
- Graphs: edge inserts or removals, etc.
- Sequences: access, update, map-reduce.

Such measures are not definable at the machine level!

Abstract cost measures fit well with abstract types, a fundamentally linguistic notion.

How can this be expressed?

Method

Abstract Cost Accounting

Idea: introduce step counting aka profiling.

$$\mathtt{step}_X : \mathbb{C} \to X \to X$$

where $\mathbb{C}$ is a type of costs (numbers, for now).

But this allows profiling to influence behavior!

$$\mathtt{if}\ \mathsf{step\_count} > 1000\ \mathtt{then}\ \ldots\ \mathsf{else}\ \ldots.$$

Such programs ought to be ruled out, but how?

Moreover, profiling destroys behavioral equivalence:

$$\mathtt{isort} \neq \mathtt{msort} : \mathtt{seq} \to F(\mathtt{seq})$$

Phase Distinctions in Type Theory

Achieve full integration using a phase distinction.

1. Prototypically, compile-time vs run-time.
2. For metatheory, syntactic vs semantic.
3. For program modules, static vs dynamic.
4. For information flow, private vs public.

What do they have in common?

1. Phase influences equational properties (behavior).
2. Non-interference between phases.

Phases Distinctions in Type Theory

A phase is specified by a proposition, ϕ.

- True only by assumption: $x : \phi \vdash e : A$.
- Proof-irrelevant: $\Gamma \vdash e = e' : \phi$.

Phase induces modalities [Rijke, Shulman, Spitters]:

Open	$\bigcirc(A) = \phi \supset A$.	"The ϕ part of A."
Closed	$\bullet(A) = \phi \vee A$.	"All of A, with no ϕ part."

Open and closed parts are exhaustive, but not exclusive!

eg, programs decompose into an dynamic part (run-time) and a static part (compile-time).

Calf = Cost-Aware Logical Framework

Niu, Sterling, Grodin & H PoPL '22

Calf enriches ∂CBPV with an extensional phase, `ext`.

- When `ext` is true, profiling is disregarded, isolating behavior.
- Otherwise, profiling distinguishes algorithms and permits cost verification.

Non-Interference Thm: If $f : \bullet(A) \to \circ(B)$, then f is constant.

In other words cost accounting cannot influence behavior!

Laws of Profiling

General laws for step counting itself:

- $\mathtt{step}^0(e) \simeq e$.
- $\mathtt{step}^c(\mathtt{step}^d(e)) \simeq \mathtt{step}^{c+d}(e)$.

Interaction between profiling and computation:

- $\mathtt{step}^c(\mathtt{bind}(e; x.f)) \simeq \mathtt{bind}(\mathtt{step}^c(e); x.f)$.
- $\mathtt{step}^c(\lambda(x \,.\, e)) \simeq \lambda(x \,.\, \mathtt{step}^c(e))$.
- $\mathtt{step}^c(\langle v_1, e_2 \rangle) \simeq \langle v_1, \mathtt{step}^c(e_2) \rangle$.

These laws enable equational reasoning about costs of programs.

Isolating Behavior

Extensional phase erases step counting:

$$\mathtt{ext}\ \mathsf{true} \vdash \mathtt{step}^{c}(e) \simeq e$$

Thus, the extensional phase isolates behavior:

$$\mathtt{ext}\ \mathsf{true} \vdash \mathsf{isort} \simeq \mathsf{msort} : \mathtt{seq} \to F(\mathtt{seq})$$

Thus, insertion sort and merge sort are equal, yet they have different cost!

Cost Analysis

Define $\mathtt{isBounded}_A(e, c)$ for $e : F(A)$ and $c : \mathbb{C}$ by

$$d : \mathbb{C} \times \bigcirc(d \leq_{\mathbb{N}} c) \times e \simeq \mathtt{step}^d(\mathtt{ret}(v))$$

(The open modality indicates that "costs do not have cost.")

Intensionally, one may specify costs of algorithms:

- $s : \mathtt{seq} \vdash \mathtt{isBounded}_{\mathtt{seq}}(\mathtt{isort}(s), |s|^2)$.
- $s : \mathtt{seq} \vdash \mathtt{isBounded}_{\mathtt{seq}}(\mathtt{msort}(s), |s| \lg |s|)$.

Thus, precise qualitative statements can be made about programs and proofs.

Analyses

Analyzing Algorithms in Calf

How are interesting algorithms defined in total type theory?

- Non-structural recursions are typical.
- Instrumentation determines "figure of merit".

How is their (behavior and) cost verified?

- Specify recurrence on cost of algorithm.
- Solve recurrence separately.

Example: Euclid's algorithm, counting modulus operations.

Patterns of Recursion

Add a "clock" parameter counting recursion depth.

- Define instrumented algorithm:

$$\mathtt{gcd}_{clocked} : \mathtt{nat} \to \mathtt{nat}^2 \to F(\mathtt{nat})$$

- Define upper bound on recursion depth:

$$\mathtt{gcd}_{depth} : \mathtt{nat}^2 \to \mathtt{nat}$$

- Define gcd itself:

$$\mathtt{gcd}(x, y) \triangleq \mathtt{gcd}_{clocked}(\mathtt{gcd}_{depth}(x, y))(x, y)$$

Clock permits formulation of algorithm, but does not determine its cost!

Patterns of Recursion

Explicitly, $\mathtt{gcd}_{clocked}$ is defined by recursion on the clock:

$$\begin{aligned}\mathtt{gcd}_{clocked}(\mathtt{zero})(x, y) &= \mathtt{ret}(x)\\ \mathtt{gcd}_{clocked}(\mathtt{succ}(k))(x, 0) &= \mathtt{ret}(x)\end{aligned}$$

and

$$\begin{aligned}&\mathtt{gcd}_{clocked}(\mathtt{succ}(k))(x, \mathtt{succ}(y)) =\\ &\quad \mathtt{bind}(\mathtt{mod}_{instr}(x, \mathtt{succ}(y))\ ;\ r\,.\,\mathtt{gcd}_{clocked}(k)(\mathtt{succ}(y), r))\end{aligned}$$

where $\mathtt{mod}_{instr}$ computes and counts moduli.

The function $\mathtt{gcd}_{depth}$ computes recursion depth for a given input (not the clock count).

Correctness

Algorithm $\texttt{gcd}$ is extensionally correct:

1. $\texttt{ext}\,\mathsf{true} \vdash \texttt{gcd}(x, \texttt{zero}) \simeq \texttt{ret}(x)$
2. $\texttt{ext}\,\mathsf{true} \vdash \texttt{gcd}(x, \texttt{suc}(y)) \simeq \texttt{gcd}(\texttt{suc}(y), \texttt{mod}(x, \texttt{suc}(y)))$

Intensionally cost is characterized by a recurrence:

$$\texttt{isBounded}(\texttt{gcd}(x, y), \texttt{gcd}_{depth}(x, y)).$$

Solve recurrence (purely mathematical):

$$\texttt{gcd}_{depth}(x, y) \leq \texttt{Fib}^{-1}(x) + 1.$$

Sorting, Revisited

Idea: count comparisons.

Define `isort` and `msort` as sketched.

- Clocked versions to manage recursion.
- Recursion bound for each algorithm.

Behavioral equivalence (extensional phase):

$$\mathtt{ext}\,\mathsf{true}, s : \mathtt{seq} \vdash \mathtt{isort}(s) \simeq \mathtt{msort}(s).$$

Cost discrepancy:

- $s : \mathtt{seq} \vdash \mathtt{isBounded}_{\mathtt{seq}}(\mathtt{isort}(s), |s|^2)$.
- $s : \mathtt{seq} \vdash \mathtt{isBounded}_{\mathtt{seq}}(\mathtt{msort}(s), |s| \lg |s|)$.

Parallel Cost Analysis

Blelloch & Greiner

Change cost monoid to $\mathbb{N}^2$:

- Work: sequential cost, as above.
- Span: idealized parallel cost.

Define parallel cost composition:

$$(w_1, s_1) \otimes (w_2, s_2) = (w_1 + w_2, \max(s_1, s_2))$$

Enrich language with parallel pairs, $e_1 \mathbin{\&} e_2$, such that

$$\mathtt{step}^{c_1}(\mathtt{ret}(v_1)) \mathbin{\&} \mathtt{step}^{c_2}(\mathtt{ret}(v_2)) = \mathtt{step}^{c_1 \otimes c_2}(\mathtt{ret}((v_1, v_2)))$$

Et voilà, may analyze parallel algorithms!

Parallel Cost Analysis of Sorting

Insertion sort remains quadratic in work and span.

Merge sort can be parallelized:

- Sequential merge:

$$s : \texttt{seq} \vdash \texttt{isBounded}(\texttt{msort}(s), |s| \lg |s|, 2\,|s| + \lg |s|)$$

- Parallel merge:

$$s : \texttt{seq} \vdash \texttt{isBounded}(\texttt{msort}(s), \lg^2(|s|+1), 2\,|s|\,(\lg^3(|s|+1))$$

NB: same algorithm, different cost analysis!

All verified using Calf in Agda.

Balanced Search Trees

Blelloch, et al '16; Li, Grodin & H '23

Verify cost analysis of red-black trees:

- Every node is red or black.
- Leaves are black; children of a red node are black.
- Balance: all branches have the same black height.

Formulate with join and singleton primitives.

- Combine two RBT's, form singletons for insert.
- Permits formulation of parallel algorithms, whereas insert does not!

Verifed logarithmic bound on join using Calf in Agda.

Amortized Analysis

Grodin & H, CALCO ’23

A coinductive formulation of queues:

$$\texttt{Queue} \triangleq \nu Q.\, \texttt{quit} : F(1) \times \texttt{enq} : (E \to Q) \times \texttt{deq} : (E \ltimes Q)$$

Operations have negative types, for which costs are forwarded to the `quit`.

The internal state of the queue is hidden by the coinductive formulation.

- Specification: a list, charge on enqueue.
- Batched: a pair of lists, charge on reversal.

Amortized Analysis

Inherent deferral of costs in coinductive setting suggests amortized analysis.

Define potential of a pair of lists $\phi(b, f) \triangleq |b|$.

Thm: $\texttt{batchedQ}(b, f) \simeq \texttt{step}^{\phi(b,f)}(\texttt{specQ}(f \bowtie \texttt{rev}(b)))$

Accounts for both correctness and cost of efficient implementation.

Equivalent to textbook account based on sequences of instructions.

Richer Languages

Higher-Order Programming

Standard analysis methods are first-order.

- Programs are separate from the data they act on.
- Simplifies cost analysis, at the expense of expressiveness.

But higher-order methods are essential for parallelism.

- $\texttt{map} : (A \rightharpoonup B) \rightharpoonup \texttt{seq}_A \rightharpoonup \texttt{seq}_B$
  (apply $f : A \rightharpoonup B$ to each element)
- $\texttt{reduce} : (A \times A \rightharpoonup A) \rightharpoonup \texttt{seq}_A \rightharpoonup \texttt{seq}_A \rightharpoonup \texttt{seq}_A$
  (combine elements on a tree)

(cf Google map-reduce in a distributed setting)

Higher-Order Programming

Textbook analysis methods do not scale to higher-order!

- `map` f requires fixed f with fixed cost for any argument.
- `reduce` f requires full understanding of f, not just an abstraction of it.

Cannot separate cost analysis from behavior in h.o. setting.

- Behavior of f may depend on its actual argument, not just its "size."
- Delicate interplay between abstraction and generality.

Only a linguistic framework can address these issues!

Effectful Programming

The purely functional Calf framework is equational!

$$e \simeq \mathtt{step}^c(\mathtt{ret}(v))$$

Programs have a result, v, and a cost, c, and nothing else.

But what if e has effects other than cost accounting?

- Exceptions?
- Jumps?
- State?
- Randomization?

Many algorithms rely on randomization for efficiency!

Probabilistic Programming

Example: coin-flipping.

$$\mathtt{flip}^{1/2}(\mathtt{step}^5(\mathtt{ret}(x)); \mathtt{step}^7(\mathtt{ret}(y)))$$

Both result and cost depend on the outcome, and no Calf-like bound is derivable.

In many cases (eg, Quicksort) only the cost is affected, not the behavior!

$$\mathtt{flip}^{1/2}(\mathtt{step}^5(\mathtt{ret}(x)); \mathtt{step}^7(\mathtt{ret}(x))) \leq \mathtt{step}^7(\mathtt{ret}(x))$$

This suggests an inequational approach to cost analysis.

DeCalf = Directed, Effectful Calf

[Grodin, Niu, Sterling & H, PoPL '24]

DeCalf offers a more expressive framework:

Directed: $e \leq e'$ is fundamental.

$$\mathtt{step}^c(m) \leq \mathtt{step}^d(m) \quad (c \leq d)$$

Effectful: neglect effects when possible.

$$\mathtt{flip}^p(\mathtt{ret}(v); \mathtt{ret}(v)) \leq \mathtt{ret}(v)$$

Cost-Aware: intensional and extensional phases.

$$\mathtt{ext}\,\mathsf{true} \vdash e \leq e' \quad \text{iff} \quad \mathtt{ext}\,\mathsf{true} \vdash e \simeq e'$$

DeCalf Methodology

Calf within DeCalf:

- $s : \mathtt{seq} \vdash \mathtt{isort}(s) \leq \mathtt{step}^{|s|^2}(\mathtt{ret}(\mathtt{sort}(s)))$.
- $s : \mathtt{seq} \vdash \mathtt{msort}(s) \leq \mathtt{step}^{|s| \lg |s|}(\mathtt{ret}(\mathtt{sort}(s)))$.

Extensionally, these become pure equations!

- $\mathtt{ext}\,\mathsf{true}, s : \mathtt{seq} \vdash \mathtt{isort}(s) = \mathtt{ret}(\mathtt{sort}(s))$.
- $\mathtt{ext}\,\mathsf{true}, s : \mathtt{seq} \vdash \mathtt{msort}(s) = \mathtt{ret}(\mathtt{sort}(s))$.

How far can this be pushed?

Ongoing and Future Work

Scaling Up

Mechanization of 15-210 Introduction to Parallel Algorithms.

- FP-based course on parallel algorithms.
- Inductive data structures.
- Unbounded length sequences with map-reduce API.

So far: use Calf in Agda for deterministic algorithms.

Planned: use DeCalf in Agda for the deterministic and probabilistic case.

Computational Adequacy in Calf

Niu & H, LICS ’23

Computational adequacy relates denotational to operational semantics for programs.

- Plotkin’s LCF Considered as a P.L. is paradigmatic.
- Relates Calf axiomatics to execution model.

Can Plotkin’s results be extended to account for cost as well as behavior?

- For Gödel’s T, a total language, and, yes, for a first-order “while” language.
- For PCF using synthetic domain theory in Calf?

Summary

Calf = Cost-Aware Logical Framework

- Intensional and extension phases.
- Extensional phase ignores cost, isolating behavior.
- Intensional phase accounts for cost.

DeCalf = Directed, Effectful Logical Framework

- Extensional phase isolates behavior.
- Pre-order weakens cost and consolidates behavior.
- Natural setting for algorithms that use effects for efficiency.

Thank you for your attention!

Questions?

References

H. Grodin and R. Harper.
Amortized analysis via coinduction.
CoRR, abs/2303.16048, 2023.

H. Grodin, Y. Niu, J. Sterling, and R. Harper.
Decalf: A directed effectful cost-aware logical framework.
CoRR, abs/2307.05938, 2023.

R. Li, H. Grodin, and R. Harper.
A verified cost analysis of joinable red-black trees.
CoRR, abs/2309.11056, 2023.

Y. Niu, J. Sterling, H. Grodin, and R. Harper.
A cost-aware logical framework.
Proc. ACM Program. Lang., 6(POPL):1–31, 2022.

Phases Abound

Phase Distinction in STC

Sterling Ph.D.,'21

Synthetic Tait Computability has two characteristic features:

- Proof-relevant: generalize relations to families.
- Synthetic: all types express computability properties.

Developed to study Cartesian cubical type theory with a full univalent universe hierarchy.

Computability ensures completeness of a generalization of normalization by evaluation, crucial for implementation.

Phase Distinction in STC

Analytically, a computability structure has two parts:

- A syntactic part, a definitional equivalence class of terms of a type.
- A semantic part, a proof of that the relevant computability property holds of the syntax.

Synthetically, all types are computability structures.

- Dependent type structure lifts to computability structures.
- Syntactic part is isolated by a phase, which collapses semantic part.

Information Flow

Sterling & H, FSCD ’22

Cost accounting may be understood in terms of information-flow security:

- Profiling is private (implementor).
- Behavior is public (client).
- Non-interference: Behavior is independent of profiling.

Generalizes to a lattice of phases for security levels.

Non-interference ensures that “downward” flows are trivial, eg any map from private to public is constant.

Program Modules

Sterling & H, J.ACM ’21

Program modules form a dependent type theory enriched with

- Static phase, `stat`, for “compile-time” aspects of a module (types, static indices.)
- Dynamic phase for “run-time” aspects (including static).
- Extension types to express sharing:

$$\{ A \mid \mathtt{stat} \hookrightarrow m \}$$

“Any module whose static part is *m*.”

Program Modules

The theory of parametricity structures has two phases:

- Syntactic, the subjects of the relations, with left and right parts.
- Semantic, the proofs of computability.

Extension types specify syntactic aspect of a comp. str.:

$$\{ S \mid \mathsf{syn} \hookrightarrow \ulcorner x : A \to B \urcorner \}$$

Representation independence for abstract types is easily obtained from this interpretation.

(First result of its kind for modules.)