Integrating Cost and Behavior in Type Theory

Robert Harper

June 21, 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)
- Parth Shastri (Carnegie Mellon)
- Jon Sterling (Cambridge)

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

Motivation

Type Theory for Programming

Dependent type theory is a natural setting for specification and verification of functional programs.

- Essentially, the propositions-as-types principle in action, formulating Brouwer's intuitionism.
- cf Martin-Löf's Constructive Mathematics and Computer Programming, and Constable, et al's NuPRL System.
- cf Agda viewed as a programming language.

However, as a logic of programs it leaves evaluation order undetermined!

- Advantage: compatible with "any"choice.
- Disadvantage: completely unspecified.

Example: Sorting

Informally, we may define

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

Extensionally these are equal as functions, because they both sort their inputs:

$$\texttt{isort} \doteq \texttt{msort} : s : \mathsf{seq} \to (s' : \mathsf{seq} \times \texttt{sorted}(s) \times \texttt{perm}(s, s'))$$

The choice of types and their associated induction principles complicates matters, but these issues have been well-developed.

Type Theory for Programming

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

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

Pedrot's and Tabareau's ∂CBPV extends Levy's framework to the dependent case.

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

Calf also includes mixed-polarity dependent sums/products (value-value and value-computation forms).

Dependent Call-by-Push Value

Syntactically,

$$v : A ::= \mathsf{nat} \mid \mathsf{seq} \mid v_1 \doteq_A v_2 \mid x : A_1 \times A_2 \mid x : A_1 \to A_2 \mid U(X)$$
$$e : X ::= F(A) \mid x : A_1 \times X_2 \mid x : A_1 \to X_2$$

Computations are sequenced, using $\mathtt{bind}(e_1 ; x \, . \, e_2)$ and $\mathtt{ret}(v)$, in anticipation of effects.

Define $e_1 \simeq_{F(A)} e_2$ to mean

$$\mathtt{thunk}(e_1) \doteq_{U(F(A))} \mathtt{thunk}(e_2).$$

They are "equal computations."

Acounting for Cost

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

Want to state and prove complexity bounds!

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

But how can equal functions have different properties?

And what does cost even mean in this setting?

- What are the steps?
- Sequential vs parallel?

Sense and Reference

Frege distinguished sense from reference.

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

A similar distinction is considered here:

- Reference: a (computable) function.
- Sense: an algorithm.

Here cost is a precise formulation of sense, and may even be used to compare proofs.

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.

Blelloch's language-based formulation is a big improvement.

- Cost semantics specifies a dependency graph whose edges constrain execution order of steps.
- Provable implementation by a Brent-type theorem whose proof defines scheduler as a function of platform characteristics.

Cost Measures

Cost is not absolute, ie per-model, but rather relative, ie per-algorithm.

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

These concepts are not definable at the RAM or TM level!

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

How can this be expressed?

Method

Abstract Cost Accounting

First idea: introduce step counting aka profiling.

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

where $\mathbb{C}$ is a type of costs (think $(\mathbb{N}, 0, +)$ for now).

eg, for sorting, use `step` to count comparisons.

But simple-minded instrumentation allows behavior to influence on cost!

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

Such programs ought to be ruled out, but how?

Abstract Cost Accounting

Second idea, introduce a writer monad $\mathbb{C} \times -$ for computations [Danielsson 98]

- $\texttt{step}^c(e)$ adds $c : \mathbb{C}$ to count.
- No operation to branch on step count.

Doing so permits tracking, specification, and verification of costs of programs … but to the exclusion of pure behavior!

eg, $\texttt{isort} \stackrel{?}{\neq} \texttt{msort} : \mathsf{seq} \to F(\mathsf{seq})$, precisely because of profiling.

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, security level.

What do they have in common?

1. Types are hybrid structures: syntax+computability, types+code, classified+public.
2. Phase (syntactic, static, level) imposes equations that "collapse" aspects (computability, code, classified).

Phases Distinctions in Type Theory

In general a phase is given a proposition, ϕ.

- True only by assumption: $x : \phi \vdash J$.
- Subterminal/proof-irrelevant: $\Gamma \vdash M \doteq M' : \phi$.

Phases induce two modalities [Rijke, Shulman, Spitters]:

- Open mode: $\bigcirc_\phi(A) := \phi \supset A$. "The ϕ part of A."
- Closed mode: $\bullet_\phi(A) := \phi \vee A$. "All of A, with no ϕ part."

These aspects of a type are exhaustive, but not necessarily exclusive.

Two basic properties of phases:

- $\bigcirc_\phi(\bullet_\phi(A)) \cong 1$, but $\bullet_\phi(\bigcirc_\phi(A)) \ncong 1$ ("fringe").
- $A \cong \bigcirc_\phi(A) \times_{\bullet_\phi(\bigcirc_\phi(A))} \bullet_\phi(A)$ (pullback wrt fringe).

Non-interference: If $f : \bullet_\phi(A) \to \bigcirc_\phi(A)$, then f is constant!

eg, syntax prior to semantics, types do not depend on code, classified cannot depend on public.

Here: the extensional phase, `ext`, eliminates step counting.

(Hereafter: $\bigcirc(A)$, $\bullet(A)$ for $\bigcirc_{\mathtt{ext}}(A)$, $\bullet_{\mathtt{ext}}(A)$, respectively.)

Synthetic Cost Analysis

Computation types form a writer monad $\bullet(\mathbb{C}) \times -$:

- $\mathbb{C}$ is a cost monoid, e.g. $(\mathbb{N}, 0, +)$.
- $\texttt{step}^c(e)$ increments cost by c, then executes e.

Use of closed modality is essential!

- Cost analysis depends on behavioral analysis.
- Costs collapse under open modality.

(The injection of $\mathbb{C}$ into $\bullet(\mathbb{C})$ is usually elided to lighten notation.)

Stepping Laws

General laws for step counting:

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

CBPV-style stepping laws for computations:

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

Any enrichment must mesh with stepping in this way.

Synthetic Cost Analysis

Extensional phase erases step counting:

$$\_ : \bigcirc(\mathtt{step}^c(e) \simeq_{F(A)} e)$$

But $\bigcirc(\bullet(\mathbb{C})) \cong 1$, so $\bigcirc(\eta_\bullet(c) \doteq_{\bullet(\mathbb{C})} \eta_\bullet(0))$, and so

$$\bigcirc(\mathtt{step}^c(e) \simeq \mathtt{step}^0(e) \simeq e).$$

Thus, the extensional phase isolates behavior:

$$\_ : \bigcirc(\mathsf{isort} \simeq_{\mathsf{seq} \to F(\mathsf{seq})} \mathsf{msort})$$

(Proof: they both sort, functions equate extensionally.)

Synthetic Cost Analysis

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

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

(Here using $\mathbb{C} = \mathbb{N}$, but will be generalized.)

Intensionally, ie non-extensionally, one may specify costs of algorithms:

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

(or discharge premise using dep. function type.)

Integrates cost and behavior with guaranteed non-interference!

Analyses

Analyzing Algorithms in Calf

How are interesting algorithms defined in total type theory?

- Non-structural recursions are typical.
- Instrumented with step's counting "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:

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

- Define upper bound on recursion depth:

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

- Define gcd itself:

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

(cf Kleene normal form theorem for TM's.)

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

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

and

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

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

The total function $\mathtt{gcd}_{depth}$ computes recursion depth for a given input as a generalized value.

Correctness

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

1. $\bigcirc(\texttt{gcd}(x, \texttt{zero}) \simeq \texttt{ret}(x))$
2. $\bigcirc(\texttt{gcd}(x, \texttt{suc}(y)) \simeq \texttt{gcd}(\texttt{suc}(y), \texttt{mod}(x, \texttt{suc}(y))))$

Intensionally cost is characterized by a recurrence:

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

Solve recurrence (purely mathematical):

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

Sorting, Revisited

Instrument comparisons with `step`.

Define `isort` and `msort` as above.

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

Behavioral equivalence:

$$s : \mathsf{seq} \vdash \bigcirc(\mathtt{isort}(s) \simeq_{F(\mathsf{seq})} \mathtt{msort}(s)).$$

Cost discrepancy:

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

Parallel Cost Analysis

Following 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 langage with parallel pairs, $e_1 \,\&\, e_2$, such that

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

(Brent-type theorem relates abstract parallel cost to implementation on p-RAM, taking account of scheduling.)

Parallel Cost Analysis

Insertion sort remains quadratic in work and span.

Merge sort can be parallelized:

- Sequential merge:

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

- Parallel merge:

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

NB: same algorithm, different cost analysis!

(See Agda repo for details.)

Amortized Analysis

Two approaches to amortization:

- Inductive definition of instruction sequences.
- Coinductive definition of abstraction.

eg, batched queues with separate front and back "halves."

- Enqueueing takes zero steps.
- Dequeueing takes length of back half steps.

The two formulations are shown to be equivalent in the companion paper in CALCO.

Computational Adequacy

Computational Adequacy in Calf

Computational adequacy relates denotational to operational semantics for programs.

- Plotkin's LCF Considered as a P.L. is paradigmatic.
- Germane to giving Calf operational meaning.

Can Plotkin's results be generalized to account for cost as well as behavior?

- LICS '23: Yes, for Gödel's T, a total language, and, yes, for a first-order "while" language with partiality.
- Ongoing: cost-aware adequacy for PCF (and FPC) using SDT within Calf.

Admitting General Recursion

Extend Calf with a lifting monad $L(A)$ satisfying compactness:

If $\mathtt{iter}(f, v) \simeq \mathtt{step}^c(\mathtt{ret}_{\mathsf{L}}(v'))$, then for some $k \geq 0$, $f^k(v) \simeq \mathtt{step}^c(\mathtt{ret}_{\mathsf{L}}(v'))$.

Consider while programs with first-order store.

- Define cost-aware denotational semantics $||p||$.
- Define cost-aware operational semantics $e \Downarrow^{\eta \bullet (c)} v$.

Cost is defined as number of β-steps in execution.

As earlier, the use of the closed modality is critical (costs collapse extensionally.)

Admitting General Recursion

Theorem: Cost-aware adequacy:

For closed while programs p of type $\texttt{bool}$, if $||p|| \simeq \texttt{step}^c(\texttt{ret}(b))$, then $e \Downarrow^{\eta \bullet (c)} \overline{b}$.

Corollary: Extensional adequacy:

For closed programs p of type $\texttt{bool}$, if $\bigcirc(||p|| \simeq \texttt{ret}(b))$, then $\bigcirc(e \Downarrow^{\eta \bullet (c)} \overline{b})$, ie $e \Downarrow \overline{b}$ in the usual sense.

(Proof uses logical relations defined internally to relate denotational to operational behavior.)

Admitting General Recursion

Internal adequacy may be used to "implement" Calf programs as while programs.

- Define $\texttt{msort}_{\mathit{calf}}$ as earlier, counting comparisons.
- Define $\texttt{msort}_{\mathit{while}}$ such that

$$\bigcirc(\texttt{msort}_{\mathit{calf}} \doteq ||\texttt{msort}_{\mathit{while}}||).$$

Adequacy ensures

- Correct behavior.
- Proportionate cost.

A possible framework for cost-aware compiler correctness?

Origin and Other Applications

Phase Distinction in STC

Sterling's 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

The phase distinction may be understood in terms of information-flow security:

- Profiling is a private matter.
- Delivered code is public.
- Non-interference: Public behavior is independent of profiling.

Generalize $\texttt{ext} \leq \top$ to security levels.

- Two-phase sets are maps $\mathbb{I}^{op} \to \mathrm{Set}$.
- Generalize to $P^{op} \to \mathrm{Set}$ with many levels of "visibility."

Program Modules

The language of program modules is a dependent type theory a la MacQueen, enriched with

- Static phase, $\mathtt{stat}$, for “compile-time” aspects of a module (types; static data/indices.)
- Dynamic phase for “run-time” aspects (incl. static).
- Extension types to express sharing:

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

Program Modules

The type 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.:

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

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.

Verification uses embedding of Calf into Agda prover.

So far, all verifications are for purely functional algorithms:

- Insertion and merge sort, sequential and parallel cost.
- Parallelizable red-black trees with join and singleton.

But probabilistic methods are also important, as are other effects.

Summary

The phase distinction integrates

- Extensional behavior.
- Intensional cost.

Moreover, the theory of phases

- Ensures non-interference.
- Supports abstract cost accounting.

Summary

Phase distinctions abound!

- Synthetic Tait Computability.
- Design of module systems.
- Integration of development and delivery.
- Parametricity structures for abstraction.
- Information flow security.

There is nothing more practical than a good theory!

References (w/Links Therein)

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

Y. Niu and R. Harper.
A metalanguage for cost-aware denotational semantics.
CoRR, abs/2209.12669, 2022.

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

J. Sterling and R. Harper.
Sheaf semantics of termination-insensitive noninterference.
In FSCD, volume 228 of LIPIcs, pages 5:1–5:19, 2022.