Cost-sensitive computational adequacy of higher-order recursion in synthetic domain theory

MFPS 2024
June 21, 2024

Yue Niu
Jon Sterling
Robert Harper
Carnegie Mellon University
yuen@cs.cmu.edu

Acknowledgements

This work was funded by the United States Air Force Office of Scientific Research and the National Science Foundation. We thank Tristan Nguyen at AFOSR for support. Yue Niu was supported by the Air Force Research Laboratory through the NDSEG fellowship. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of AFOSR, AFRL, or NSF.

Introduction

The story begins with a type theory **calf** developed to unify *cost-sensitive* and *functional* verification [Niu+22].

- Functional: IO-behavior of programs, data structure invariants
- Cost-sensitive: computational cost or resource usage (time, space, *etc*.)

Functional properties are about *if* a program is correct, cost-sensitive properties are about *how much* resource a program uses.

Introduction

calf supports a *denotational* style of cost analysis — connection to operational semantics via a **cost-sensitive computational adequacy** property à la Plotkin [Plo77].

Prior work: cost-sensitive adequacy for first-order recursion [NH23].

This talk: cost-sensitive adequacy for *higher-order* recursion.

Outline

Introduction to **calf**:

- Cost-sensitive and functional reasoning in **calf**
- Cost-sensitive adequacy property

Integrating higher-order recursion in **calf**:

- Introduction to *synthetic domain theory* (SDT)
- Cost-sensitive SDT
- Cost-sensitive adequacy in SDT

Cost as an abstract effect

In **calf**, cost is an *abstract* effect $\mathsf{F}(A)$ supporting an operation $\mathsf{step} : \mathbb{C} \to \mathsf{F}(1)$. Think of step^c as taking c abstract steps:

$$\begin{aligned} &\mathit{insertSort} : \mathsf{list} \to \mathsf{F}(\mathsf{list}) \\ &\mathit{insertSort}(l) = \ldots \mathsf{step}^c; e \ldots \end{aligned}$$

Under the hood define $\mathsf{F}(A) = \mathbb{C} \times A$ and $\mathsf{step}^c = (c, \star)$. Can reason about step's equationally:

$$\begin{aligned} &\mathsf{step}^{c_1}; \mathsf{step}^{c}_{2} = \mathsf{step}^{c_1 + c_2} \\ &\mathsf{step}^{0}; e = e \end{aligned}$$

Functional reasoning in calf

How to reason about the purely *functional* properties of cost-sensitive programs?

$$\mathrm{isSorted}(\mathit{insertSort}(l)) \iff \mathrm{isSorted}(\mathit{mergeSort}(l))$$

Should be automatic because both are sorting algorithms. But *not* because $\mathit{insertSort} \neq \mathit{mergeSort}$ due to presence of cost structure!

Cost as a phase

The functional semantics of (total) programs is naturally modeled in **Set**.

Set is too "flat": the cost effect $\mathbb{C} \times - : \mathbf{Set} \to \mathbf{Set}$ does not distinguish data from cost structure.

calf : cost as a new **dimension** or **phase**.

Cost structure as families

calf = the internal type theory of the category of *families* $\mathbf{Set}^{\rightarrow}$

A type in $Set^{\rightarrow}$ is a cost-sensitive set equipped with a restriction action to the purely functional component:

$$A^{\bullet} \xrightarrow{\;\pi_A\;} A^{\circ}$$

$A^{\bullet}$ "cost-sensitive"

A° "functional"

Think Kripke/possibles world semantics over $\mathbb{I} = \{\circ \rightarrow \bullet\}$.

Functional *vs.* cost-sensitive phase

Presheaves over $\{\circ \to \bullet\}$ exhibits a **phase distinction**:

- World at $\circ$ = **functional phase**
- World at $\bullet$ = **cost-sensitive phase**
- In cost-sensitive phase, $\mathit{insertSort} \neq \mathit{mergeSort}$.
- In functional phase, $\mathit{insertSort} = \mathit{mergeSort}$.

Presheaf restriction $\bullet \to \circ$ trivializes/redacts cost structure!

Modal types

Can introduce *modal types* that are either purely functional or purely cost-sensitive.

Definition

A type is *purely functional* or *function-modal* when it is in the image of the constant presheaves functor $\mathbf{Set} \to \mathbf{Set}^{\to}$.

Definition

A type is *purely cost-sensitive* or *cost-modal* when it is given by a terminal map $A \to 1$.

Cost effect with cost-modal types

Define $\mathsf{F}(A)$ by using a *cost-modal* monoid object $\mathbb{C}$:

$$\mathsf{F}\left(\begin{array}{c} A^{\bullet} \\ \downarrow \\ A^{\circ} \end{array}\right) = \begin{array}{c} \mathbb{N} \\ \downarrow \\ 1 \end{array} \times \begin{array}{c} A^{\bullet} \\ \downarrow \\ A^{\circ} \end{array} = \begin{array}{c} \mathbb{N} \times A^{\bullet} \\ \downarrow \\ A^{\circ} \end{array}$$

Internalization

Modal types can be phrased in the internal language of $\mathbf{Set}^{\rightarrow}$.

Let $\P : \Omega$ be the *intermediate* proposition in $\mathbf{Set}^{\rightarrow}$:

$$\bot = \begin{matrix} 0 \\ \downarrow \\ 0 \end{matrix} \qquad \P = \begin{matrix} 0 \\ \downarrow \\ 1 \end{matrix} \qquad \top = \begin{matrix} 1 \\ \downarrow \\ 1 \end{matrix}$$

Assuming $\P$ = restricting to the functional phase.

Internal characterization of modal types

Proposition

A type A is function-modal when $(\P \to A) \cong A$.

Proposition

A type A is cost-modal when $(\P \to A) \cong 1$.

In other words, a function-modal type “thinks” the functional phase holds and a cost-modal type “thinks” the functional phase is false.

Constructing modal types

Given A, $\P \to A$ is function-modal. Dually, construct a cost-modal type $\P \vee A$ as follows:

$$\begin{array}{ccc} A \times \P & \xrightarrow{\pi_2} & \P \\ {\scriptstyle \pi_1}\big\downarrow & & \big\downarrow{\scriptstyle *} \\ A & \xrightarrow[\eta]{} & \ulcorner\, \P \vee A \end{array}$$

The *cost modality* $\P \vee -$ quotients the type A to a unique point $*$ in the functional phase.

Functional and cost reasoning, internally

Semantically, $\mathsf{F}(A) = (\P \vee \mathbb{C}) \times A$.

Thus *insertSort* $\neq$ *mergeSort* since the cost monoid $\P \vee \mathbb{C}$ is nontrivial.

But, $\P \to ((\P \vee \mathbb{C}) \cong 1)$, so *insertSort* $=$ *mergeSort* in the functional phase!

calf *vs*. programming languages

Cost analysis in **calf** is *equational* or *denotational*.

Problems:

- How to relate cost analysis in **calf** to PLs with *operational* cost semantics?
- How to reconcile general recursive functions in PLs with total functions in **calf**?

calf *vs*. programming languages

Solution:

- Enrich **calf** with *partiality* via *synthetic domain theory*.
- Relate PLs and **calf** by an *internal, cost-sensitive* computational adequacy property.

Upshot:

- General recursive programming in **calf**
- Cost-sensitive generalization of Plotkin's classic adequacy property.

Cost-sensitive computational adequacy

Example: take STLC equipped with the cost effect $\mathsf{F}(A)$. Internal to **calf**, we have a language $\mathcal{L} = (\mathsf{Ty} : \mathcal{U}, \mathsf{Tm} : \mathsf{Ty} \to \mathcal{U})$.

Internal denotational cost semantics of $\mathcal{L}$:

- $[\![-]\!]_{\mathsf{Ty}} : \mathsf{Ty} \to \mathcal{U}$
- $([\![-]\!]_{\mathsf{Tm}})_A : \mathsf{Tm}(A) \to [\![A]\!]_{\mathsf{Ty}}$

Note $[\![\mathsf{F}(A)]\!] = \mathbb{C} \times [\![A]\!]$.

Internal operational cost semantics of $\mathcal{L}$:

- $\Downarrow_A \subseteq \mathsf{Tm}(A) \times \mathbb{C} \times \mathsf{Tm}(A)$

Cost-sensitive computational adequacy

Definition

A language satisfies **cost-sensitive computational adequacy** when for all $e : \mathsf{F}(2)$, $[\![e]\!] =_{\mathbb{C} \times [\![A]\!]} (c, [\![v]\!])$ if and only if $e \Downarrow^c v$.

Classic Plotkin adequacy: $[\![-]\!]$ carves out functions that are definable operationally.

Cost-sensitive adequacy: $[\![-]\!]$ carves out **calf** functions that are definable operationally *in a cost-reflecting way*.

Cost-sensitive adequacy for higher-order recursion

Prior work: $\mathcal{L}$ = Algol-like languages with while loops [NH23].

This work: $\mathcal{L}$ = **PCF**.

Method: *synthetic domain theory*

Recursion in type theory

To define the denotational cost semantics of **PCF** in **calf**, we need a notion of *partial* functions in type theory.

Attempt: model **calf** in presheaves valued in ωcpo's: $\omega\mathbf{CPO}^{\to}$.

Unfortunately not a model of dependent type theory.

Synthetic domain theory

Integrate higher-order recursion into type theory by means of *synthetic domain theory* (SDT):

- Intuitionistic type theory
- Class of *predomains*
- *All* definable predomain maps automatically continuous

Concretely: a topos $\mathcal{E}$ equipped with a full subcategory Predom.

Axioms of SDT

To start, we need an object called the *dominance* that serves as the classifier of the *support* of partial maps.

Definition

A *dominance* subobject $\Sigma \hookrightarrow \Omega$ that is closed under $\top : \Omega$ and dependent sums.

Frequently Σ is also required to be closed under $\bot : \Omega$.

The dominance also has the dual role as the classifier of *Scott-open* subsets. For example, in $\omega\mathbf{CPO}$ $\Sigma = \{0 \leqslant 1\}$.

Lifting structure

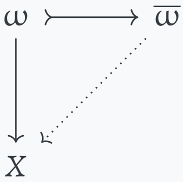
The dominance Σ induces a *lifting structure* $\mathsf{L}(A) = \Sigma_{\phi:\Sigma}.\phi \to A$: partial maps $A \overset{\Sigma}{\hookleftarrow} D \to B$ as total maps $A \to \mathsf{L}(B)$.

Lifting induces an incidence relation $\omega \hookrightarrow \overline{\omega}$ including the initial lift algebra ω into the final lift coalgebra $\overline{\omega}$.

Think of $\omega \hookrightarrow \overline{\omega}$ as a *figure shape* that we use to state the completeness properties of predomains.

Predomains in SDT

A *predomain* has the unique extension property along $\omega \hookrightarrow \overline{\omega}$:



Synthetic counterpart to ωcpos, which extend along the figure shape $\{0 \leqslant 1 \leqslant \dots\} \hookrightarrow \{0 \leqslant 1 \leqslant \cdots \leqslant \infty\}$.

Model of SDT

A *model of SDT* is given by a topos $\mathcal{E}$ equipped with a predomain dominance Σ.

Every such model induces a full subcategory of predomains that is a *reflective exponential ideal*:

- Closed under limits and exponentials: types of **PCF**
- All colimits exist: used to define the cost-modal type $\P \vee \mathbb{C}$
- Every endomap of domains has a fixed-point: fix operator

Model of SDT

Remark: the choice of the dominance is critical — taking $D = \{\bot, \top\}$ trivializes partiality and taking Ω itself results in maps that are not continuous; commonly we have $D \subsetneq \Sigma \subsetneq \Omega$.

Denotational semantics of **PCF** in cost-sensitive SDT

To interpret **PCF** with the cost effect $\mathsf{F}(A)$, need a proposition ϕ for the *functional phase*:

Definition

A *model of SDT with a phase distinction* is a model of SDT $(\mathcal{E}, \Sigma, \phi)$ where ϕ is a Σ-proposition.

Semantically: $[\![\mathsf{F}(A)]\!] = \mathsf{L}(\mathbb{C} \times [\![A]\!])$ with $\mathbb{C}$ cost-modal.

Need $\phi : \Sigma$ to ensure $\phi \vee A$ is a predomain when A is one.

Operational semantics of **PCF**

Our proof of computational adequacy relies on the fact that $e \Downarrow^c v$ is a Σ-proposition.

Define the operational semantics as a partial function $\mathsf{eval} : \mathsf{Tm}(\mathsf{F}(A)) \to \mathsf{Tm}(\mathsf{F}(A)) \to \mathsf{L}(\mathbb{C})$:

$$\mathsf{eval}(e, v) = \begin{cases} c \boxplus \mathsf{eval}(e', v) & \mathsf{out}(e) = \mathsf{inr} \cdot (c, e') \\ (e = v, \lambda u.0) & \mathsf{out}(e) = \mathsf{inl} \cdot \star \end{cases}$$

In the above, we write $\mathsf{out} : \mathsf{Tm}(A) \to 1 + (\mathbb{C} \times \mathsf{Tm}(A))$ for the one step transition relation, and $- \boxplus -$ for the cost algebra map.

Logical relation for computational adequacy

Define a family of relations $\lhd_A \subseteq [\![A]\!] \times \mathsf{Tm}(A)$ between the syntax and semantics of **PCF**.

A technical point is the definition of $\lhd_{\mathsf{F}(A)}$:

$$e\ (R \Rightarrow S)\ e' = \forall [a\ R\ a']\ (e\ a)\ S\ (e'\ a')$$
$$e \lhd_{\mathsf{F}A} e' = \forall [f\ (\lhd_A \Rightarrow \leqslant)\ f']\ e;f \leqslant e';f'$$

In the above we write $e \leqslant e'$ for the *specialization order* or *definedness order* on $\mathsf{F}(1) \cong \mathsf{L}(\mathbb{C})$.

Ensures that $(- \lhd_{\mathsf{F}(A)} e') \subseteq [\![\mathsf{F}(A)]\!]$ is always a sub-predomain or *admissible*.

Fundamental lemma and computational adequacy

We may prove the fundamental lemma of the logical relation:

Theorem

Given $\Gamma \vdash e : A$*, we have* $\Gamma \vdash [\![e]\!] \lhd_A e$*.*

Cost-sensitive computational adequacy follows directly from the fundamental lemma:

Theorem

Given $e : \mathsf{F}(1)$*, we have that* $[\![e]\!] = \mathsf{eval}(e, \star)$.

Model of cost-sensitive SDT

To incorporate cost structure as a phase distinction, define a model of SDT fibred over $\mathbf{Set}^{\rightarrow}$.

Isolate a (small) category $\mathcal{C}$ of *internal dcpos* in $\mathbf{Set}^{\rightarrow}$.

- Presheaves on $\mathcal{C}$ is *almost* a model of SDT.
- Restrict to sheaves on $\mathcal{C}$ for the extensive coverage: preserves $\emptyset$ and $+$.

Theorem

The category of (internal) sheaves on $\mathcal{C}$ furnishes a model of SDT such that the functional phase proposition $\P : \mathcal{C}$ is preserved by the Yoneda embedding.

Related work

- Computational adequacy in SDT [Sim99; Sim04]
- Relative sheaf models of SDT [SH22]
- Rooted in the type-theoretic framework **calf**
- Extended the results of Niu and Harper [NH23] to **PCF**
- Denotational cost semantics based on prior work on effectful **PCF** [Kav+19]

Conclusion

- Integrated higher-order recursion into **calf** type theory
- Internal cost-sensitive computational adequacy theorem for **PCF**
- Connecting denotational and operational reasoning for cost analysis in type theory
- Relative sheaf model of the function-cost phase distinction

Future work

- Recursive types [Sim04]
- Relating internal and *external* cost-sensitive adequacy

Thanks for listening!

References I

[1] G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. "Recurrence Extraction for Functional Programs through Call-by-Push-Value". In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019). DOI: 10.1145/3371083.

[2] Yue Niu and Robert Harper. "A Metalanguage for Cost-Aware Denotational Semantics". In: *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2023, pp. 1–14. DOI: 10.1109/LICS56636.2023.10175777.

References II

[3] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. "A Cost-Aware Logical Framework". In: *Proceedings of the ACM on Programming Languages* 6.POPL (Jan. 2022). DOI: 10.1145/3498670. arXiv: 2107.04663 [cs.PL].

[4] Gordon D. Plotkin. "LCF considered as a programming language". In: *Theoretical Computer Science* 5.3 (1977), pp. 223–255. ISSN: 0304-3975. DOI: 10.1016/0304-3975(77)90044-5.

References III

[5] Alex Simpson. "Computational adequacy for recursive types in models of intuitionistic set theory". In: *Annals of Pure and Applied Logic* 130.1 (2004). Papers presented at the 2002 IEEE Symposium on Logic in Computer Science (LICS), pp. 207–275. ISSN: 0168-0072. DOI: 10.1016/j.apal.2003.12.005.

[6] Alex K. Simpson. "Computational Adequacy in an Elementary Topos". In: *Computer Science Logic*. Ed. by Georg Gottlob, Etienne Grandjean, and Katrin Seyr. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1999, pp. 323–342. ISBN: 978-3-540-48855-2. DOI: 10.1007/10703163_22.

References IV

[7] Jonathan Sterling and Robert Harper. "Sheaf semantics of termination-insensitive noninterference". In: *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. Ed. by Amy P. Felty. Vol. 228. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Aug. 2022, 5:1–5:19. ISBN: 978-3-95977-233-4. DOI: 10.4230/LIPIcs.FSCD.2022.5. arXiv: 2204.09421 [cs.PL].