Cost-sensitive programming, verification, and semantics

Dissertation defense
July 29, 2024

Yue Niu

Carnegie Mellon University

Acknowledgements

Thank you for the six years.

Battle of users and designers

Programmers:

- Simplicity
- Familiarity ({...} and ;)
- *Ability* and *can do's*

Researchers:

- Global, mathematical properties
- *Inability* is a feature, not a bug

Balance between ergonomics and safety guarantees.

What is a safety property?

A **safey property** is a specification of the desired behavior of a program e.

Weak safety: e computes an integer.

Strong(er) safety: e computes the 57th prime number.

Program verification is the activity of verifying that programs meet their specifications.

What is semantics?

Semantics is the process of interpreting PLs into known mathematical structures.

Program verification reduces to mathematical proof.

Most reliable method for establishing properties of not just computer programs, but every discipline amenable to the axiomatic method.

Thesis statement: the internal modal type theory of presheaves furnishes an ergonomic language for ***cost-sensitive*** programming, verification, and semantics.

What is cost?

Computational resource of programs:

- Concrete: time, memory, energy
- Abstract: number of critical operations in an algorithm

A *cost model* is the determination of the resource of interest.

What is cost?

Computational resource of programs:

- Concrete: time, memory, energy
- Abstract: number of critical operations in an algorithm

A *cost model* is the determination of the resource of interest.

Example: number of comparisons in sorting.

What is type theory?

Type theory is a language for describing abstract *collections*. A *type* is a collection such that we have a criterion for when two elements are equal.

What is type theory?

Type theory is a language for describing abstract *collections*. A *type* is a collection such that we have a criterion for when two elements are equal.

	collection	when equal
Booleans	$\{0, 1\}$	$0 = 0$ and $1 = 1$
Natural numbers	$\{0, 1, \ldots\}$	$n = m$ when equal as natural numbers
Functions	$A \to B$	equal when equal pointwise
Sequences	$\mathrm{list}(A)$	$[a_1, \ldots, a_n] = [b_1, \ldots, b_m]$ when $n = m$ and $a_i = b_i$

What is a type?

Types can enforce strong safety properties.

Example

Let sort to be the type of functions $f : \mathrm{list}(\mathbb{N}) \to \mathrm{list}(\mathbb{N})$ such that $f(l)$ is a permutation of l and in ascending order.

Observe that sort contains a *unique* element *sort*.

The cost paradox

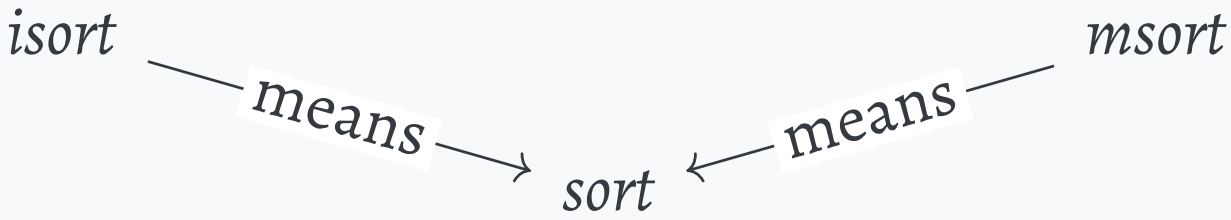
But we know there are *different* sorting algorithms!

For example, insertion sort and mergesort are distinguished by the number of comparisons required.

Nonetheless we have $isort = msort$ in type theory.

Descriptions *vs.* meanings

isort and *msort* are *descriptions*, *sort* is the *meaning*.



Type theory is a theory of *meanings* — descriptions are immaterial!

Cost as an explicit structure

Profiling: make cost an explicit observable part of the code.

$$\mathbb{N} \times A \to \mathbb{N} \times B$$

Cost as an explicit structure

Profiling: make cost an explicit observable part of the code.

$$\mathbb{N} \times A \to \mathbb{N} \times B$$

For example,

$$isort(c, l) = (c + k_1 \cdot |l|^2, sort(l))$$
$$msort(c, l) = (c + k_2 \cdot |l| \log |l|, sort(l))$$

Distinguish sorting algorithms by their computational cost (number of comparisons).

Problems with naive cost profiling (I)

The type $\mathbb{N} \times A \to \mathbb{N} \times B$ does not enforce the semantics of profiling.

Cost is observable by the *meaning* of algorithms:

$$badList : \mathbb{N} \times \mathsf{list}(\mathbb{N}) \to \mathbb{N} \times \mathsf{list}(\mathbb{N})$$
$$badList(c, l) = (c, c :: l)$$

Meaning of algorithms *depends* on cost profiling!

Cannot reason about pure meaning/functional semantics.

Problems with naive cost profiling (II)

Even when the program is correctly instrumented, we cannot easily reason about programs up to functional semantics.

For instance, $\mathit{isort} \neq \mathit{msort}$ as functions of the type $\mathbb{N} \times \mathsf{list}(\mathbb{N}) \to \mathbb{N} \times \mathsf{list}(\mathbb{N})$.

Highly unergonomic when reasoning about pure functional specifications or correctness properties.

Programming and verification

A new phase distinction

Cost structure *vs.* meaning as a **phase distinction**.

A new phase distinction

Cost structure *vs.* meaning as a **phase distinction**.

First introduced in **sterling-harper:2021**, type-theoretic phase distinctions have been used to solve many PL problems:

A new phase distinction

Cost structure *vs.* meaning as a **phase distinction**.

First introduced in **sterling-harper:2021**, type-theoretic phase distinctions have been used to solve many PL problems:

- Metatheory of type theories and programming languages [**sterling:2021:thesis**]
- Interactive theorem proving [**gratzer-sterling-angiuli-coquand-birkedal:2022**]
- Information flow security [**sterling-harper:2022**]

A new phase distinction

Cost structure *vs.* meaning as a **phase distinction**.

First introduced in **sterling-harper:2021**, type-theoretic phase distinctions have been used to solve many PL problems:

- Metatheory of type theories and programming languages [**sterling:2021:thesis**]
- Interactive theorem proving [**gratzer-sterling-angiuli-coquand-birkedal:2022**]
- Information flow security [**sterling-harper:2022**]

This thesis: use a phase distinction enable both **cost-sensitive** and **functional** reasoning in a single type theory.

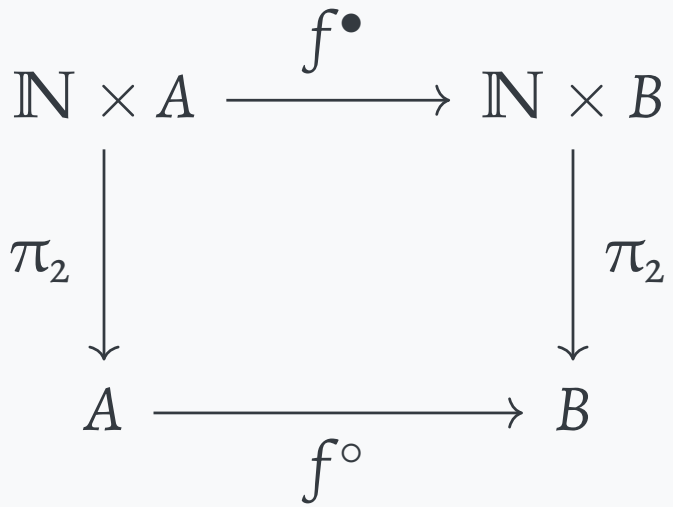
The function-cost phase distinction

We have a cost-profiled function:

$$\mathbb{N} \times A \xrightarrow{f^\bullet} \mathbb{N} \times B$$

The function-cost phase distinction

Ensure that every cost-profiled function $f^\bullet$ *lies over* a plain function f°:



Commutativity rules out programs disobeying the profiling semantics (*e.g.* badList).

Presheaf semantics of cost

Semantics encapsulated by the category $\widehat{\mathbb{I}}$ of presheaves over the interval $\{\circ \to \bullet\}$. Think Kripke/possible world semantics.

Presheaf semantics of cost

Semantics encapsulated by the category $\widehat{\mathbb{I}}$ of presheaves over the interval $\{\circ \to \bullet\}$. Think Kripke/possible world semantics.

- 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!

Algorithms *vs.* functions

$$\begin{array}{ccc} \mathbb{N} \times \mathrm{list}(\mathbb{N}) & \overset{isort}{\longrightarrow} & \mathbb{N} \times \mathrm{list}(\mathbb{N}) \\ \pi_2 \downarrow & & \downarrow \pi_2 \\ \mathrm{list}(\mathbb{N}) & \underset{sort}{\longrightarrow} & \mathrm{list}(\mathbb{N}) \end{array}$$

$$\begin{array}{ccc} \mathbb{N} \times \mathrm{list}(\mathbb{N}) & \overset{msort}{\longrightarrow} & \mathbb{N} \times \mathrm{list}(\mathbb{N}) \\ \pi_2 \downarrow & & \downarrow \pi_2 \\ \mathrm{list}(\mathbb{N}) & \underset{sort}{\longrightarrow} & \mathrm{list}(\mathbb{N}) \end{array}$$

Distinguished in the cost-sensitive phase, identified in the functional phase.

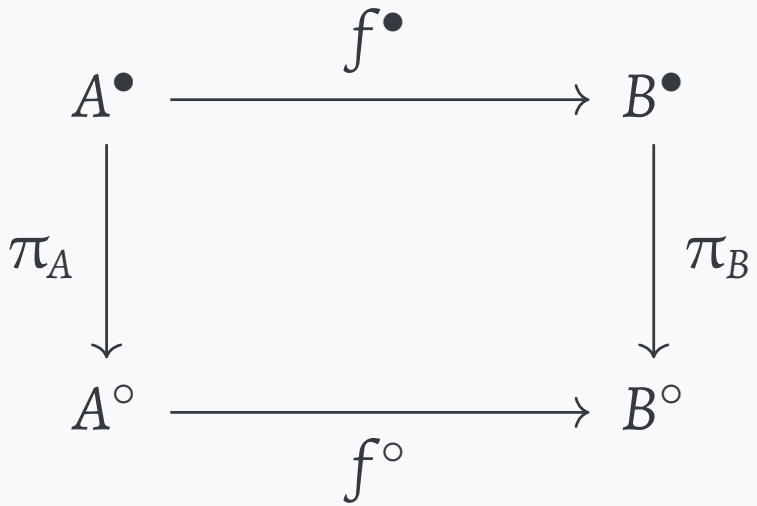
Internal modal type theroy

Internal type theory equipped with **modalities** for working with purely cost-sensitive or purely functional structures.

Upshot: can define and reason about cost-profiled programs and their meanings as a *single* function in the internal language of $\widehat{\mathbb{I}}$.

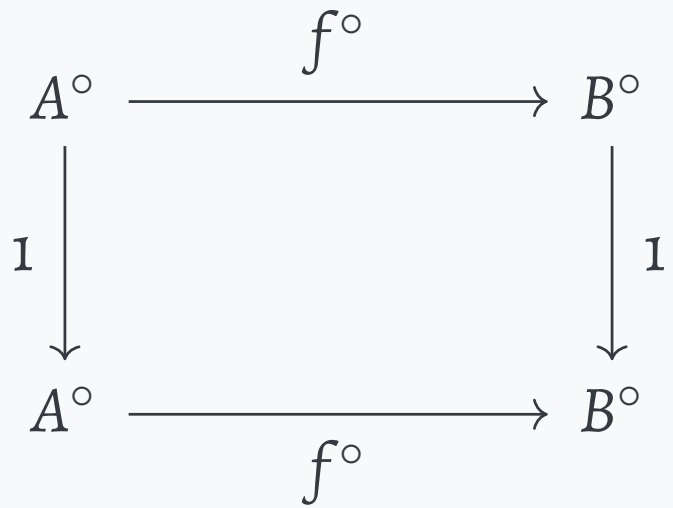
Modal types

In general, a function in $\widehat{\mathbb{\Pi}}$ looks like a square:



Function modal types

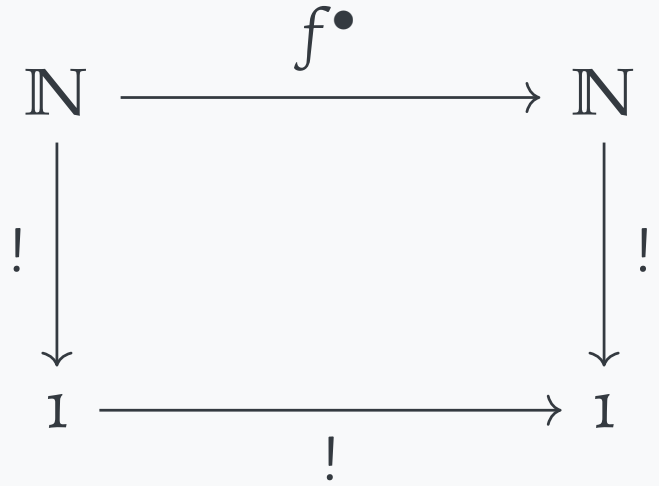
We can have *trivially* cost-profiled functions:



A type determined by the identity $A \to A$ is called ***function modal***.

Cost modal types

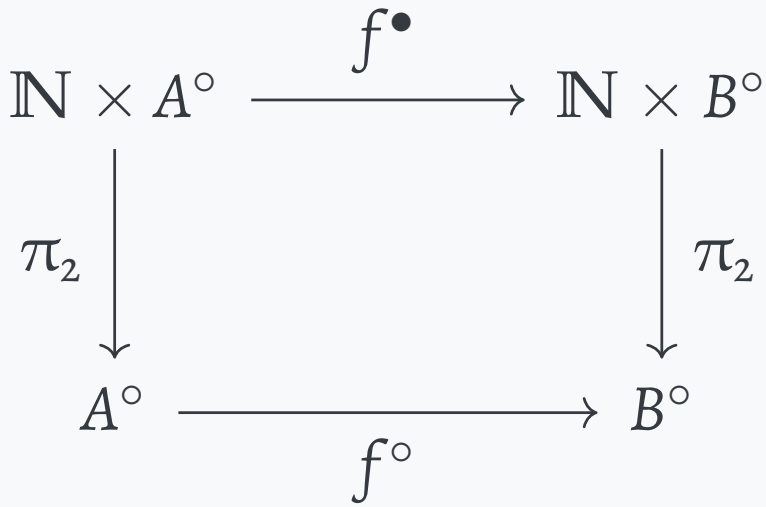
Conversely, we have cost-sensitive functions with trivial meanings:



A type determined by the terminal map $A \to 1$ is called ***cost modal***.

Combining modalities

A cost-sensitive function with nontrivial functional behavior given by amalgamating function and cost modal types:



Square encodes possible dependencies: in $\mathbb{N} \times B^\circ$, $\mathbb{N}$ can depend on both components of $\mathbb{N} \times A^\circ$, but B° can only depend on A°.

The functional phase proposition

Internally, identify function and cost modal types via the functional **phase proposition**.

Recall that there are three global propositions in $\widehat{\mathbb{I}}$:

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

Let the intermediate proposition $\P$ be the phase proposition.

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

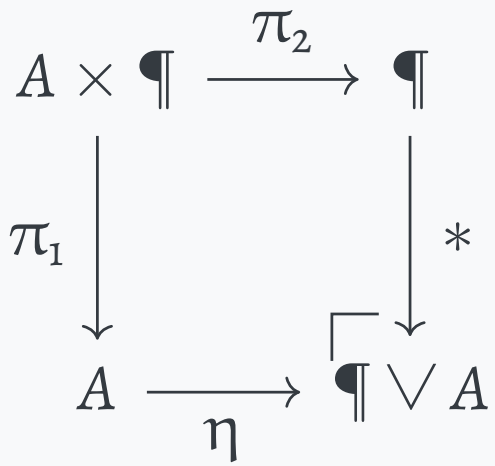
A type A is cost-modal when $(\P \to A) \cong 1$.

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:



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

Cost-sensitive functions, internally

A cost-sensitive in the internal language of $\widehat{\mathbb{I}}$ is given by a function of the following form:

$$(\P \vee \mathbb{N}) \times A \to (\P \vee \mathbb{N}) \times B$$

Cost-sensitive functions, internally

A cost-sensitive in the internal language of $\widehat{\mathbb{I}}$ is given by a function of the following form:

$$(\P \vee \mathbb{N}) \times A \to (\P \vee \mathbb{N}) \times B$$

Example:

$$(\P \vee \mathbb{N}) \times \mathsf{list}(\mathbb{N}) \to (\P \vee \mathbb{N}) \times \mathsf{list}(\mathbb{N})$$

Cost-sensitive functions, internally

Assuming that A, B are function modal, we have an internal correctness property for cost-profiled functions:

Proposition

Assuming $\P$, *every* $f : (\P \vee \mathbb{N}) \times A \to (\P \vee \mathbb{N}) \times B$ *lies over a plain function* $|f| : A \to B$, *in the sense that* $\pi_2 f = |f| \pi_2$:

$$\begin{array}{ccc} (\P \vee \mathbb{N}) \times A & \xrightarrow{f} & (\P \vee \mathbb{N}) \times B \\ \downarrow{\scriptstyle \pi_2} & & \downarrow{\scriptstyle \pi_2} \\ A & \xrightarrow[|f|]{} & B \end{array}$$

Cost-sensitive functions, internally

The point is that $|f|$ is determined completely *mechanically* by assuming the proposition ¶.

Suffices to just work with f, and recover $|f|$ whenever by assuming ¶.

One more thing…

We still allow some "slack" in the representation by allowing functions that do not aggregate cost correctly.

$$\mathit{forgetful} : (\P \vee \mathbb{N}) \times A \to (\P \vee \mathbb{N}) \times A$$
$$\mathit{forgetful}(c, a) = (0, a)$$

Cost as an abstraction

Work with cost as an *abstract* data type.

interface Cost
  $\mathsf{F} : \mathsf{type} \to \mathsf{type}$
  $\mathsf{ret} : A \to \mathsf{F}A$
  $\mathsf{bind} : \mathsf{F}A \to (A \to \mathsf{F}B) \to \mathsf{F}B$
  $\mathsf{step} : \mathsf{F}A \to \mathsf{F}A$

Implement $\mathsf{F}A = (\P \vee \mathbb{N}) \times A$ under the hood, but not revealed to the user.

Cost as an abstraction

We can implement algorithms with abstract cost effects.

$$
\begin{aligned}
&\mathit{insert} : A \to \mathsf{list}(\mathsf{nat}) \to \mathsf{F}(\mathsf{list}(\mathsf{nat})) \\
&\mathit{insert}(x, \mathsf{nil}) = \mathsf{ret}([x]) \\
&\mathit{insert}(x, y :: l) = \\
&\quad \mathsf{step}; \mathsf{bind}(\mathit{lt}(x, y), \lambda b.\mathsf{case}\, b\, \mathsf{of} \left\{ \begin{array}{l} \mathsf{bind}(\mathit{insert}(x, l), \lambda r.\mathsf{ret}(y :: r)) \\ \mathsf{ret}(x :: y :: l) \end{array} \right.)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{isort} : \mathsf{list}(\mathsf{nat}) \to \mathsf{F}(\mathsf{list}(\mathsf{nat})) \\
&\mathit{isort}(\mathsf{nil}) = \mathsf{nil} \\
&\mathit{isort}(x :: l) = \mathsf{bind}(\mathit{isort}(l), \lambda l'.\, \mathit{insert}(x, l'))
\end{aligned}
$$

Cost as an abstraction

Abstract cost effects support equational reasoning:

$$\mathsf{bind}(\mathsf{ret}(a), f) = f(a)$$
$$\mathsf{bind}(\mathsf{step}(e), f) = \mathsf{step}(\mathsf{bind}(e, f))$$

For example we can derive a cost bound on insertion sort.

$$\mathit{isort}(l) = \mathsf{step}^{k \cdot |l|^2}(\mathsf{ret}(\mathit{sort}(l)))$$

Observe that this is also a *correctness* property.

Reasoning about meaning

In the functional phase $\mathsf{F}A \cong A$, justifying additional equations:

$$¶ \to (\mathsf{step}; e = e)$$

Recover functional reasoning by assuming ¶:

$$¶ \to (\mathit{isort} = \mathit{msort})$$

A cost-aware logical framework

The combination of the phase distinction and abstract cost effect yields a type theory called **calf**.

[1] https://github.com/jonsterling/agda-calf

A cost-aware logical framework

The combination of the phase distinction and abstract cost effect yields a type theory called **calf**.

Deployed on common textbook algorithms: sorting, Euclid's algorithm, batched queues [**niu-sterling-grodin-harper:2022**].

[1] https://github.com/jonsterling/agda-calf

A cost-aware logical framework

The combination of the phase distinction and abstract cost effect yields a type theory called **calf**.

Deployed on common textbook algorithms: sorting, Euclid's algorithm, batched queues [**niu-sterling-grodin-harper:2022**].

Verified very precise, *concrete* cost bounds and *correctness* specifications of algorithms in a computerized proof assistant[1].

[1] https://github.com/jonsterling/agda-calf

A cost-aware logical framework

The combination of the phase distinction and abstract cost effect yields a type theory called **calf**.

Deployed on common textbook algorithms: sorting, Euclid's algorithm, batched queues [**niu-sterling-grodin-harper:2022**].

Verified very precise, *concrete* cost bounds and *correctness* specifications of algorithms in a computerized proof assistant[1].

Since then **calf** has been used for analyzing amortized algorithms [**grodin-harper:2023**] and nontrivial computational effects [**grodin-niu-sterling-harper:2024**].

[1] https://github.com/jonsterling/agda-calf

Related work

Three perspectives on extant frameworks for cost analysis:

Related work

Three perspectives on extant frameworks for cost analysis:

1. Operational [**atkey:2010**; **mevel-jourdan-pottier:2019**]
   - (+) Faithful representation of cost profiling
   - (-) Mathematical reasoning about program is compromised

Related work

Three perspectives on extant frameworks for cost analysis:

1. Operational [**atkey:2010**; **mevel-jourdan-pottier:2019**]
   - (+) Faithful representation of cost profiling
   - (-) Mathematical reasoning about program is compromised
2. Equational [**danielsson:2008**; **handley-vazou-hutton:2019**]
   - (+) Ordinary mathematical reasoning of cost-profiled programs
   - (-) No way to work directly with pure meaning/functional semantics
   - (-) Cost profiling is not enforced strictly: cost is *observable* by functional component of programs

Related work

3. Type systems [**hoffmann-aehlig-hofmann:2012**; **wang-wang-chlipala:2017**; **rajani-gaboardi-garg-hoffmann:2021**]
   - (+) Cost profiling is syntactically enforced
   - (-) No pure functional reasoning
   - (-) Not suitable for general verification purposes

Related work

3. Type systems [**hoffmann-aehlig-hofmann:2012**; **wang-wang-chlipala:2017**; **rajani-gaboardi-garg-hoffmann:2021**]
   - (+) Cost profiling is syntactically enforced
   - (-) No pure functional reasoning
   - (-) Not suitable for general verification purposes

calf fills a previously empty design space: a general-purpose verification framework with a faithful cost-profiling semantics that supports both cost-sensitive and functional reasoning.

Semantics

Cost semantics

calf supports *denotational/equational* cost analysis for *total functions*.

The rest of the thesis addresses the following equations:

1. How does this relate to *operational cost semantics*?
2. How to deal with *partial functions*?

Cost semantics

calf supports *denotational/equational* cost analysis for *total functions*.

The rest of the thesis addresses the following equations:

1. How does this relate to *operational cost semantics*?
2. How to deal with *partial functions*?

Extend **calf** with **general recusion** and prove an internal **computational adequacy** property.

Operational semantics

PL researchers often define the semantics of PLs in terms of an **operational semantics** given by a system of transitions between program states [**plotkin_81_structural**].

$$\frac{}{\mathsf{bind}(\mathsf{ret}(a), f) \mapsto f(a)} \qquad \frac{}{\mathsf{ret}(a)\ \mathsf{val}}$$

Operational semantics

PL researchers often define the semantics of PLs in terms of an **operational semantics** given by a system of transitions between program states [**plotkin_81_structural**].

$$\frac{}{\mathsf{bind}(\mathsf{ret}(a), f) \mapsto f(a)} \qquad \frac{}{\mathsf{ret}(a)\ \mathsf{val}}$$

Meaning not by what a program *is*, but by what it *does*.

Operational cost semantics

Can extend operational semantics to account for cost.

$$\frac{}{\mathsf{step}(e) \mapsto 1, e}$$

Operational cost semantics

Can extend operational semantics to account for cost.

$$\frac{}{\mathsf{step}(e) \mapsto 1, e}$$

How relate operational and denotational cost semantics?

Internal program semantics

Study operational semantics as an object theory *within* **calf**!

Internal program semantics

Study operational semantics as an object theory *within* **calf**!

Fixing an internal PL $\mathcal{L}$ in **calf**, a denotational model of $\mathcal{L}$ is just a function $[\![-]\!]$.

Internal program semantics

Study operational semantics as an object theory *within* **calf**!

Fixing an internal PL $\mathcal{L}$ in **calf**, a denotational model of $\mathcal{L}$ is just a function $[\![-]\!]$.

The image of $[\![-]\!]$ carves out cost-profiled functions that are operationally definable.

Internal program semantics

Study operational semantics as an object theory *within* **calf**!

Fixing an internal PL $\mathcal{L}$ in **calf**, a denotational model of $\mathcal{L}$ is just a function $[\![-]\!]$.

The image of $[\![-]\!]$ carves out cost-profiled functions that are operationally definable.

We have equational cost bounds on such functions (*e.g.* *isort*).

Internal program semantics

Study operational semantics as an object theory *within* **calf**!

Fixing an internal PL $\mathcal{L}$ in **calf**, a denotational model of $\mathcal{L}$ is just a function $[\![-]\!]$.

The image of $[\![-]\!]$ carves out cost-profiled functions that are operationally definable.

We have equational cost bounds on such functions (*e.g.* *isort*).

Such equational cost bound in **calf** also holds wrt operational semantics of $\mathcal{L}$ when *computational adequacy* holds.

Cost-sensitive computational adequacy

Suppose we have a program of ground type $e : \mathrm{F}1$.

Definition

Classic computational adequacy [**plotkin:1977**]: $[\![e]\!]$ defined if and only if e terminates operationally.

Definition

Cost-sensitive Plotkin adequacy: $[\![e]\!] = c$ if and only if e terminates in c steps operationally.

Recursion in type theory

We aim to prove computational adequacy for **PCF**, a language with general recursive functions.

But type theory is a theory of *total functions* — divergence not in the image of $[\![-]\!]$!

Introduce partiality by using some form of **domains**.

Recursion in type theory

1. *Internal domain theory* [**dejong:2023:thesis:1**]

Recursion in type theory

1. *Internal domain theory* [**dejong:2023:thesis:1**]
   - (+) Mathematically canonical solution.

Recursion in type theory

1. *Internal domain theory* [**dejong:2023:thesis:1**]
   - (+) Mathematically canonical solution.
   - (-) Mismatch between domain-theoretic functions and type-theoretic functions.

Recursion in type theory

1. *Internal domain theory* [**dejong:2023:thesis:1**]
   - (+) Mathematically canonical solution.
   - (-) Mismatch between domain-theoretic functions and type-theoretic functions.
2. *Synthetic guarded domain theory (SGDT)* [**sgdt:2011**; **paviotti:2016**]

Recursion in type theory

1. *Internal domain theory* [**dejong:2023:thesis:1**]
   - (+) Mathematically canonical solution.
   - (-) Mismatch between domain-theoretic functions and type-theoretic functions.
2. *Synthetic guarded domain theory (SGDT)* [**sgdt:2011**; **paviotti:2016**]
   - (+) Can work naively with guarded domains — *every* (guarded) type-theoretic endofunction has a (guarded) fixed-point.

Recursion in type theory

1. *Internal domain theory* [**dejong:2023:thesis:1**]
   - (+) Mathematically canonical solution.
   - (-) Mismatch between domain-theoretic functions and type-theoretic functions.
2. *Synthetic guarded domain theory (SGDT)* [**sgdt:2011**; **paviotti:2016**]
   - (+) Can work naively with guarded domains — *every* (guarded) type-theoretic endofunction has a (guarded) fixed-point.
   - (-) Fixed-point equations do not hold exactly; correct notion of equality is *weak bisimulation*.

Recursion in type theory

1. *Internal domain theory* [**dejong:2023:thesis:1**]
   - (+) Mathematically canonical solution.
   - (-) Mismatch between domain-theoretic functions and type-theoretic functions.
2. *Synthetic guarded domain theory (SGDT)* [**sgdt:2011**; **paviotti:2016**]
   - (+) Can work naively with guarded domains — *every* (guarded) type-theoretic endofunction has a (guarded) fixed-point.
   - (-) Fixed-point equations do not hold exactly; correct notion of equality is *weak bisimulation*.
3. ***Synthetic domain theory (SDT)*** [**hyland:1991**; **reus-streicher:199"9**]

Recursion in type theory

1. *Internal domain theory* [**dejong:2023:thesis:1**]
   - (+) Mathematically canonical solution.
   - (-) Mismatch between domain-theoretic functions and type-theoretic functions.
2. *Synthetic guarded domain theory (SGDT)* [**sgdt:2011**; **paviotti:2016**]
   - (+) Can work naively with guarded domains — *every* (guarded) type-theoretic endofunction has a (guarded) fixed-point.
   - (-) Fixed-point equations do not hold exactly; correct notion of equality is *weak bisimulation*.
3. ***Synthetic domain theory (SDT)*** [**hyland:1991**; **reus-streicher:199"9**]
   - (+) Like internal domain theory, fixed-points hold on the nose.

Recursion in type theory

1. *Internal domain theory* [**dejong:2023:thesis:1**]
   - (+) Mathematically canonical solution.
   - (-) Mismatch between domain-theoretic functions and type-theoretic functions.
2. *Synthetic guarded domain theory (SGDT)* [**sgdt:2011**; **paviotti:2016**]
   - (+) Can work naively with guarded domains — *every* (guarded) type-theoretic endofunction has a (guarded) fixed-point.
   - (-) Fixed-point equations do not hold exactly; correct notion of equality is *weak bisimulation*.
3. ***Synthetic domain theory (SDT)*** [**hyland:1991**; **reus-streicher:199"9**]
   - (+) Like internal domain theory, fixed-points hold on the nose.
   - (+) Like SGDT, every type-theoretic function is domain-theoretic (*i.e.* continuous), justifying one to work naively in type theory

Synthetic domain theory

Identify a class of *predomains* amongst ordinary types that behave like *e.g.* ω-cpos from classic domain theory.

Concretely, models of SDT are given by toposes equipped with a distinguished object Σ called the *dominance*.

Everything derived by imposing structures on Σ (*cf.* the line object R in synthetic differential geometry).

Axioms of synthetic domain theory

The dominance serves as the classifier of the *termination support* of partial maps.

Definition

A *dominance* is a collection of propositions closed under $\bot$, $\top$ and dependent sums.

A Σ*-subset* is one whose characteristic map is valued in Σ (think *computational* predicate).

Lifting structure

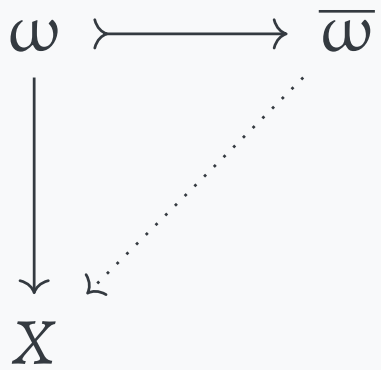
The dominance Σ induces a *lifting structure* $\mathsf{L}(A) = \sum_{\phi:\Sigma} .\phi \to A$: partial maps $A \overset{\Sigma}{\hookleftarrow} D \to B$ as total maps $A \to \mathsf{L}(B)$.

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.

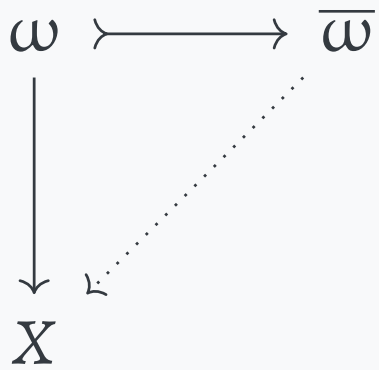
Complete types

A type is *complete* when it has the unique extension property along $\omega \hookrightarrow \overline{\omega}$:



Complete types

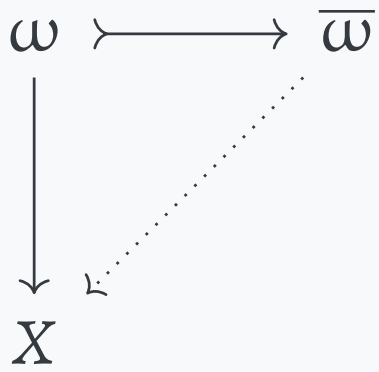
A type is *complete* when it has the unique extension property along $\omega \hookrightarrow \overline{\omega}$:



Unique extension properties as closure under *generalized limit processes* [**streicher:1997:repletion**].

Complete types

A type is *complete* when it has the unique extension property along $\omega \hookrightarrow \overline{\omega}$:



Unique extension properties as closure under *generalized limit processes* [**streicher:1997:repletion**].

For example, ω-cpos defined by the unique extension property relative to $\{0 \sqsubseteq 1 \sqsubseteq \ldots\} \hookrightarrow \{0 \sqsubseteq 1 \sqsubseteq \ldots \sqsubseteq \infty\}$.

Candidates for predomains

To close complete types under lifting, one may consider well-complete types [**longley-simpson:1997**].

Definition

A type A is *well-complete* when LA is complete.

Well-complete types are closed under lifting and form a reflective exponential ideal — all the structures necessary for everyday denotational semantics.

An ergonomic interface for SDT

Well-complete types are adequate to obtain all results in this thesis.

However, notions like unique extension properties are not familiar to many *users* of denotational semantics.

In the thesis, work relative to an *order-theoretic interface* of SDT that closes the gap between classic and synthetic domains.

An order-theoretic formulation of SDT

Observe that we did not use the *information/approximation* order to define predomains.

An order-theoretic formulation of SDT

Observe that we did not use the *information/approximation* order to define predomains.

Can equip types with a *derived synthetic order* $\sqsubseteq$.

An order-theoretic formulation of SDT

Observe that we did not use the *information/approximation* order to define predomains.

Can equip types with a *derived synthetic order* $\sqsubseteq$.

The order $\sqsubseteq$ its closure properties (ω-joins $\bigvee$) furnish a convenient and familiar *interface* to users of SDT.

An order-theoretic formulation of SDT

Observe that we did not use the *information/approximation* order to define predomains.

Can equip types with a *derived synthetic order* $\sqsubseteq$.

The order $\sqsubseteq$ its closure properties (ω-joins $\bigvee$) furnish a convenient and familiar *interface* to users of SDT.

Think of $\sqsubseteq$ and $\bigvee$ as the "{...}" and ";" of SDT.

An order-theoretic formulation of SDT

The way to achieve this order-theoretic interface is beyond the scope of this talk, but follows from existing results in the literature [**phoa:1991**; **reus:1995**; **longley-simpson:1997**].

Work with a more restrictive class of predomains called *replete* [**hyland:1991**] types.

Predomains and domains

We have a topos $\mathscr{E}$ equipped with a class of predomains Predom:

- Predom is Cartesian closed and (internally) (co)complete
- Domains (predomains with least elements) support fixed-points of endomaps.

Predomains and domains

We have a topos $\mathscr{E}$ equipped with a class of predomains Predom:

- Predom is Cartesian closed and (internally) (co)complete
- Domains (predomains with least elements) support fixed-points of endomaps.

Predomains $\sim$ ω-cpos and domains $\sim$ pointed ω-cpos.

Complete relative to ω-chains instead of $\mathbb{N}$-chains.

Sufficient for the ordinary denotational semantics of **PCF**.

SDT model of the phase distinction

To incorporate cost structure as a phase distinction, assume a distinguished proposition $\phi : \Sigma$ — ensures the cost modality $\P \vee -$ produces a predomain.

Definition

An *SDT model of the phase distinction* is a topos $\mathscr{E}$ equipped with a complete dominance Σ and a Σ-proposition $\phi : \Sigma$.

In this SDT we define a model of *cost-sensitive* version of **PCF**.

Cost-sensitive **PCF**

We work with a version of **PCF** equipped with the cost interface $\mathsf{F}A$:

interface Cost
$\mathsf{F} : \mathsf{type} \to \mathsf{type}$
$\mathsf{ret} : A \to \mathsf{F}A$
$\mathsf{bind} : \mathsf{F}A \to (A \to \mathsf{F}B) \to \mathsf{F}B$
$\mathsf{step} : \mathsf{F}A \to \mathsf{F}A$
$\mathsf{fix} : (\mathsf{F}A \to \mathsf{F}A) \to \mathsf{F}A$

Note that we now support *partial* cost-sensitive functions.

Denotational cost semantics

The Cartesian closed structure of **PCF** is interpreted in a canonical way into predomains.

For cost structure, we lift the interpretation of $\mathsf{F}A$ in **calf**:

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

Assume $\mathbb{C}$ *cost-modal*, *i.e.* $(\P \to \mathbb{C}) \cong 1$. Recover ordinary semantics in the functional phase: $\P \to ([\![\mathsf{F}A]\!] = \mathsf{L}[\![A]\!])$.

Operational cost semantics

Recall that we may define the operational cost semantics of the cost effect:

$$\frac{}{\mathsf{step}(e) \mapsto 1, e}$$

We can extend this to a big-step semantics $e \Downarrow^c v$.

Operational cost semantics

Writing $\mathsf{Tm}(A)$ for terms of **PCF**, define a recursive function $\mathsf{eval} : \mathsf{Tm}(\mathsf{F}A) \to \mathsf{Tm}(\mathsf{F}A) \to \mathsf{L}(\mathbb{C})$:

$$
\begin{array}{l}
\textbf{fun}\ \mathsf{eval}(e, v) = \\
\quad \textbf{case}\ \mathsf{out}(e)\ \textbf{of} \\
\qquad \mathsf{inl} \cdot * \hookrightarrow (e = v, \lambda u : (e = v), \mathsf{O}) \\
\qquad \mathsf{inr} \cdot (c, e') \hookrightarrow (c \boxplus \mathsf{eval}(e', v))
\end{array}
$$

- We have $\mathsf{out}(e, v) = \mathsf{inr}(c, e')$ if and only if $e \mapsto c, e'$.
- We write $c \boxplus e$ to increment the cost of a computation e by c units.

Operational cost semantics

Define $e \Downarrow^c v$ if and only if $\mathrm{eval}(e, v) = c$.

Observe that $e \Downarrow^c v$ is a Σ-proposition — we will use this in the proof of computational adequacy.

Defining $e \Downarrow^c v$ as the reflexive-transitive closure of $e \mapsto c, e'$ need *not* yield a Σ-proposition without additional assumptions.

Logical relation for computational adequacy

Following **plotkin:1977**, define a family of relations $\lhd_A \subseteq [\![A]\!] \times \mathsf{Tm}(A)$ between the semantics and syntax of **PCF**.

Interpretation of $\lhd_{FA}$:

$$
\begin{aligned}
&e \lhd_{FA} e' = \\
&\quad \forall [f : [\![A \to \mathsf{F1}]\!], f' : \mathsf{Tm}(A \to \mathsf{F1})] \\
&\qquad (\forall [a \lhd_A a'] f(a) \ll f'(a')) \to \\
&\qquad\quad \mathsf{bind}(e, f) \ll \mathsf{bind}(e', f')
\end{aligned}
$$

For $e : [\![\mathsf{F1}]\!]$ and $e' : \mathsf{Tm}(\mathsf{F1})$, define $(e \ll e') \stackrel{\text{def}}{=} (e \sqsubseteq \mathsf{eval}(e', *))$.

Computational adequacy

Observe that $(- \ll e') \subseteq \mathsf{L}(\mathbb{C})$ is an *admissible* subset, allowing us to prove the fundamental lemma by fixed-point induction.

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* $[\![e]\!] = c$ *if and only if* $e \Downarrow^c *$.

Model construction

Model of cost-sensitive SDT

Define a *relative* model of SDT following **sterling-harper:2022**, *i.e.* a model fibred over the presheaf model of **calf**.

Model of cost-sensitive SDT

Define a *relative* model of SDT following **sterling-harper:2022**, *i.e.* a model fibred over the presheaf model of **calf**.

Isolate a (small) category $\mathcal{C}$ of *internal dcpos* in $\widehat{\mathbb{I}}$.

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

Model of cost-sensitive SDT

Define a *relative* model of SDT following **sterling-harper:2022**, *i.e.* a model fibred over the presheaf model of **calf**.

Isolate a (small) category $\mathcal{C}$ of *internal dcpos* in $\widehat{\mathbb{I}}$.

- 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.

Conclusions

The big picture (programming and verification)

The PL community has been converging towards type theory as the premier (meta)language for doing formalized mathematics and programming.

The big picture (programming and verification)

The PL community has been converging towards type theory as the premier (meta)language for doing formalized mathematics and programming.

From a computer science perspective, cost analysis and cost-sensitive verification has been a glaring gap.

The big picture (programming and verification)

The PL community has been converging towards type theory as the premier (meta)language for doing formalized mathematics and programming.

From a computer science perspective, cost analysis and cost-sensitive verification has been a glaring gap.

This thesis addressess the question of cost in a framework that is both ergonomic in practice (battle tested in Agda) and theoretically sound.

The big picture (semantics)

Relate denotational and operational cost semantics by viewing PLs as *internal* theories.

The big picture (semantics)

Relate denotational and operational cost semantics by viewing PLs as *internal* theories.

The results suggest an *internal* extraction procedure for producing real executable programs that meet their proven cost bounds.

Thanks for listening!

References I