Guarded Computational Type Theory

Jonathan Sterling
Robert Harper
Carnegie Mellon University
The story begins almost 50 years ago with the invention of *domain theory* and *denotational semantics*.

**Main idea:** find category in which mixed variance operators have fixed points.
The story begins almost 50 years ago with the invention of domain theory and denotational semantics.

**Main idea:** find category in which mixed variance operators have fixed points.

Variety of techniques employed today, but the problem of mixed variance fixed points remained the fundamental struggle of PL semantics.
In 2001, Appel and McAllester invent new stratified semantic technique to construct fixed points of mixed variance, called *step-indexing*.

**Main idea:** index everything by its “stage” of construction. 
step-indexed predicate = monotone sequence of predicates
In 2001, Appel and McAllester invent new stratified semantic technique to construct fixed points of mixed variance, called \textit{step-indexing}.

**Main idea:** index everything by its “stage” of construction. step-indexed predicate $= \text{monotone sequence of predicates}$

$$\llbracket \mu \alpha. A \rrbracket \rho \triangleq \{ \text{fold}(e) \mid e \in \llbracket A \rrbracket (\rho, \alpha \mapsto \llbracket \mu \alpha. A \rrbracket \rho) \}$$

$$\llbracket A \rightarrow B \rrbracket \rho \triangleq \{ \lambda x. e \mid \forall v \in \llbracket A \rrbracket \rho. e[v/x] \in \llbracket B \rrbracket \rho \}$$

(no!! not well-defined)
In 2001, Appel and McAllester invent new stratified semantic technique to construct fixed points of mixed variance, called step-indexing.

**Main idea:** index everything by its “stage” of construction. step-indexed predicate = monotone sequence of predicates

$$\begin{align*}
[i \mid \mu \alpha. A] \rho & \triangleq \{ \text{fold}(e) \mid \forall j < i. e \in [j \mid A](\rho, \alpha \mapsto [j + 1 \mid \mu \alpha. A] \rho) \} \\
[i \mid A \to B] \rho & \triangleq \{ \lambda x. e \mid \forall j \leq i. \forall v \in [j \mid A] \rho. e[v/x] \in [j \mid B] \rho \}
\end{align*}$$
Using ideas from Nakano [2000], abstract version of step-indexing factored in terms of *approximation modality* ▷, called “later” [Appel et al., 2007].

\[
\begin{align*}
\llbracket i \mid \mu \alpha . A \rrbracket \rho & \triangleq \\{ \text{fold}(e) \mid \forall j < i. \; e \in \llbracket j \mid A \rrbracket (\rho, \alpha \mapsto \llbracket j + 1 \mid \mu \alpha . A \rrbracket \rho) \} \\
\llbracket i \mid A \rightarrow B \rrbracket \rho & \triangleq \{ \lambda x . e \mid \forall j \leq i. \forall v \in \llbracket j \mid A \rrbracket \rho . e[v/x] \in \llbracket j \mid B \rrbracket \rho \}
\end{align*}
\]
Using ideas from Nakano [2000], abstract version of step-indexing factored in terms of approximation modality ▷, called “later” [Appel et al., 2007].

\[
\begin{align*}
{\llbracket \mu \alpha. A \rrbracket (\rho) & \overset{\Delta}{=} \text{fix } R. \{ \text{fold}(e) \mid ▷(e \in \llbracket A \rrbracket (\rho, \alpha \mapsto R)) \} \\
\llbracket A \rightarrow B \rrbracket (\rho) & \overset{\Delta}{=} \{ \lambda x. e \mid \forall v \in \llbracket A \rrbracket (\rho). e[v/x] \in \llbracket B \rrbracket (\rho) \}
\end{align*}
\]
Using ideas from Nakano [2000], abstract version of step-indexing factored in terms of approximation modality \( \triangleright \), called “later” [Appel et al., 2007].

\[
\begin{aligned}
[\mu \alpha. A] \rho & \triangleq \text{fix } R. \{ \text{fold}(e) \mid \triangleright (e \in [A](\rho, \alpha \mapsto R))\} \\
[A \rightarrow B] \rho & \triangleq \{ \lambda x. e \mid \forall v \in [A] \rho. e[v/x] \in [B] \rho \}
\end{aligned}
\]

not just for predicates! can also form guarded-recursive “sets”:

\[
gstream \equiv \IN \times \triangleright gstream
\]
Programming Example: Guarded Streams

type gstr[X] = cons of X * |> gstr[X]

let head (xs : gstr[X]) : X =
    let cons (x, _) = xs in x

let tail (xs : gstr[X]) : |> gstr[X] =
    let cons (_, ys) = xs in ys

gfix F in
    fun (cons (x, xs)) (cons (y, ys)) ->
        cons (f x y, F <*> xs <*> ys)

Can’t write unguarded tail function! Guarded-recursive types ensure that all observations are *causal*. 
Constant modality

Models of guarded recursion often include another modal operator □ which *neutralizes* ▷:

\[
\begin{align*}
□A & \to A \\
□A & \to □□A \\
□▷A & \to □A
\end{align*}
\]

Converts guarded recursion to coinduction.

\[
gstream \cong \mathbb{N} \times ▷gstream
\]
Models of guarded recursion often include another modal operator □ which neutralizes ▷:

□A → A
□A → □□A
□▷ A → □A

Converts guarded recursion to coinduction.

gstream ≅ ℕ × ▷gstream
□gstream ≅ ℕ × □gstream
Models of guarded recursion often include another modal operator □ which neutralizes ▷:

□A → A  □A → □□A  □ ▷ A → □A

Converts guarded recursion to coinduction.

gstream ≅ ℕ × ▷gstream
sequence ≅ ℕ × sequence
Constant modality

Models of guarded recursion often include another modal operator □ which neutralizes ▷:

□A → A  □A → □□A  □ ▷ A → □A

Converts guarded recursion to coinduction.

gstream ≅ ℕ × ▷gstream
sequence ≅ ℕ × sequence

Improved version [Atkey and McBride, 2013]: index ▷_κ in “clocks” κ, re-cast □ as quantifier ∀κ.
Constant modality

Models of guarded recursion often include another modal operator □ which *neutralizes* ▷:

\[
\begin{align*}
□A & \rightarrow A \\
□A & \rightarrow □□A \\
□ ▷ A & \rightarrow □A
\end{align*}
\]

Converts guarded recursion to coinduction.

\[
gstream_\kappa \cong \mathbb{N} \times ▷_\kappa gstream_\kappa \\
\forall \kappa. gstream_\kappa \cong \mathbb{N} \times \forall \kappa. gstream_\kappa
\]

Improved version [Atkey and McBride, 2013]: index ▷\kappa in “clocks” \kappa, re-cast □ as quantifier \(\forall\kappa\).
Main task: develop powerful metalanguages for guarded domain-theoretic semantics and programming.
Main task: develop powerful metalanguages for guarded domain-theoretic semantics and programming.

Most advances in the area of guarded higher-order logics, but dependent type theory indispensible.
Main task: develop powerful metalanguages for guarded domain-theoretic semantics and programming.

Most advances in the area of guarded higher-order logics, but dependent type theory indispensible.

Dependent type theory = semantic framework to unify the study of programs with the study of programming languages.
Bizjak and Møgelberg [2017] present elegant denotational account of guarded dependent type theory ($\Pi, \Sigma, \triangleright^\kappa, \forall \kappa, \ldots$)

A couple things gave us pause...
Bizjak and Møgelberg [2017] present elegant denotational account of guarded dependent type theory ($\Pi, \Sigma, \triangleright_{\kappa}, \forall \kappa, \ldots$)

A couple things gave us pause...

1. what about operational meaning? (status unknown for GDTT)
Bizjak and Møgelberg [2017] present elegant denotational account of guarded dependent type theory ($\Pi, \Sigma, \triangleright_{\kappa}, \forall \kappa, \ldots$)

A couple things gave us pause...
1. what about operational meaning? (status unknown for GDTT)
2. universes $\mathcal{U}_i$ are essential, but not directly supported in GDTT semantics
What’s the deal with universes?

Adequacy of $\forall \kappa$ to encode coinduction requires that functions of clocks are constant. Called clock irrelevance.

But what about $(\lambda \kappa. \lambda A. \triangleright_\kappa A) \in \forall \kappa.(\mathcal{U} \rightarrow \mathcal{U})$?
What’s the deal with universes?

Adequacy of \( \forall \kappa \) to encode coinduction requires that functions of clocks are constant. Called **clock irrelevance**.

But what about \( (\lambda \kappa. \lambda A. \triangleright_\kappa A) \in \forall \kappa. (\mathcal{U} \to \mathcal{U}) \)?

**global orthogonality constraint**
What’s the deal with universes?

Adequacy of $\forall \kappa$ to encode coinduction requires that functions of clocks are constant. Called clock irrelevance.

But what about $(\lambda \kappa. \lambda A. \triangleright_\kappa A) \in \forall \kappa.(\mathcal{U} \to \mathcal{U})$?

**global orthogonality constraint**

**Idea:** get rid of ordinary universes, replace with weaker clock-context-indexed universes $\mathcal{U}_i^\kappa$.
Experience in behavioral type theoretic tradition (MLTT 1979, Nuprl/CTT) led us to believe we could surmount these problems.
Experience in behavioral type theoretic tradition (MLTT 1979, Nuprl/CTT) led us to believe we could surmount these problems.

Idea: types aren’t abstract collections of abstract things. Types are specifications of program behavior. Natural for programs to exhibit multiple behaviors, have multiple types.
Experience in behavioral type theoretic tradition (MLTT 1979, Nuprl/CTT) led us to believe we could surmount these problems.

**Idea:** types aren’t abstract collections of abstract things. Types are specifications of *program behavior*. Natural for programs to exhibit multiple behaviors, have multiple types.

Behavioral perspective led us immediately to an alternative solution (hard to see from more abstract vantage point).
Experience in behavioral type theoretic tradition (MLTT 1979, Nuprl/CTT) led us to believe we could surmount these problems.

**Idea:** types aren’t abstract collections of abstract things. Types are specifications of *program behavior*. Natural for programs to exhibit multiple behaviors, have multiple types.

Behavioral perspective led us immediately to an alternative solution (hard to see from more abstract vantage point).

We call it **Guarded Computational Type Theory / CTT**. 
Solution to universe problem

Inspired by extension of propositions-as-types discipline to intersection connective [Bickford and Constable, 2012, Allen et al., 2006].

Decompose clock quantifiers into two parts:
1. uniform quantifier (intersection): \{\kappa \div \text{clk}\} \rightarrow A_\kappa
2. non-uniform quantifier (product): (\kappa : \text{clk}) \rightarrow A_\kappa

Idea: don’t impose global orthogonality condition, instead change the quantifier.
Solution to universe problem

Inspired by extension of propositions-as-types discipline to intersection connective [Bickford and Constable, 2012, Allen et al., 2006].

Decompose clock quantifiers into two parts:
1. **uniform** quantifier (intersection): \( \{ \kappa \div \text{clk} \} \rightarrow A_\kappa \)
2. **non-uniform** quantifier (product): \( (\kappa : \text{clk}) \rightarrow A_\kappa \)

**Idea:** don’t impose global orthogonality condition, instead change the quantifier.

Programs have more than one behavior: in particular, the specification \( M \in \{ \kappa \div \text{clk} \} \rightarrow A_\kappa \) means that \( M \) exhibits *all* the behaviors \( A_\kappa \) simultaneously.
Initial version of \texttt{CTT} had subtle+critical bug, wasn’t clear if it could be salvaged. Need for formalization.
To gain confidence, formalized corrected version in Coq using internal topos-theoretic technique.

**Basic idea:** design a suitable presheaf topos $\mathcal{S}$ whose internal logic has clocks, $\triangleright \kappa$ and $\forall \kappa$. See our paper, but think of presheaves on finitely many copies of $\omega$. 
The logic of clocks

There is a presheaf of clocks \( \mathbb{K} : \mathcal{S} \downarrow \).

The topos \( \mathcal{S} \downarrow \) contains a rich higher-order logic with a \( \mathbb{K} \)-indexed family of modalities \( \triangleright_K \), enjoying crucial principles including the following:

<table>
<thead>
<tr>
<th>IRRELEVANCE</th>
<th>“IMPATIENCE”</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists \kappa : \mathbb{K}. \top )</td>
<td>( \forall \phi : \Omega^K. (\forall \kappa : \mathbb{K}. \triangleright_K \phi(\kappa)) \Rightarrow \forall \kappa : \mathbb{K}. \phi(\kappa) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MONOTONICITY</th>
<th>( \wedge )-DISTRIBUTIVITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall \kappa : \mathbb{K}. \forall \phi : \Omega. \phi \Rightarrow \triangleright_K \phi )</td>
<td>( \forall \kappa : \mathbb{K}. \forall \phi, \psi : \Omega. \triangleright_K (\phi \wedge \psi) \equiv (\triangleright_K \phi \wedge \triangleright_K \psi) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( \Rightarrow )-DISTRIBUTIVITY</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall \kappa : \mathbb{K}. \forall \phi, \psi : \Omega. \triangleright_K (\phi \Rightarrow \psi) \equiv (\triangleright_K \phi \Rightarrow \triangleright_K \psi) )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STRONG LÖB INDUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall \kappa : \mathbb{K}. \forall \phi : \Omega. (\triangleright_K \phi \Rightarrow \phi) \Rightarrow \phi )</td>
</tr>
</tbody>
</table>
Key idea: higher-order internal encoding of syntax for later modality and clock quantification¹, De Bruijn encoding for ordinary binders:

\[
\begin{align*}
\kappa : \mathbb{K} & \quad A : \text{Prog} \\
\hline
\Downarrow_{\kappa} A : \text{Prog}
\end{align*}
\]

\[
\begin{align*}
A : \text{Prog}^{\mathbb{K}} \\
\hline
(\kappa : \text{clk}) \to A(\kappa) : \text{Prog}
\end{align*}
\]

\[
\begin{align*}
A : \text{Prog}^{\mathbb{K}} \\
\hline
\{\kappa \triangleright \text{clk}\} \to A(\kappa) : \text{Prog}
\end{align*}
\]

¹See Fiore et al. [1999], Hofmann [1999].
Next, we construct the PER-model of type theory inside the *internal logic* of $S\Downarrow$.

**Idea:** internal inductive-recursive definition of a universe hierarchy, assigning relations to type-codes.

All clauses are standard, except the relations which define the clock-related connectives; approximately:

\[
\begin{align*}
\llbracket \uparrow \kappa A \rrbracket &= \{(M, N) \mid \uparrow \kappa ((M, N) \in \llbracket A \rrbracket)\} \\
\llbracket (\kappa : \text{clk} \to A(\kappa)) \rrbracket &= \{(M, N) \mid \forall \kappa : \text{K}. (\text{app}(M, \kappa), \text{app}(N, \kappa)) \in \llbracket A(\kappa) \rrbracket\} \\
\llbracket \{\kappa \div \text{clk} \to A(\kappa)\} \rrbracket &= \{(M, N) \mid \forall \kappa : \text{K}. (M, N) \in \llbracket A(\kappa) \rrbracket\} \\
\vdots
\end{align*}
\]
Programming in CTT

Using a combination of both clock quantifiers and the later modality, we can program the type of guarded-recursive streams “qua fixed points on universes”\(^2\)

\[
gstr : (\_ : \text{clk}) \rightarrow \mathbb{U}_0 \rightarrow \mathbb{U}_0 \\
gstr = \lambda \kappa. \text{fix } F \text{ in } \lambda A. A \times \uprime \kappa F(A) \\
\]

\[
\text{seq} : \mathbb{U}_0 \rightarrow \mathbb{U}_0 \\
\text{seq} = \lambda A. \{ \kappa \div \text{clk} \} \rightarrow \text{gstr } \kappa A
\]

\[
gzipwith : (f : X \rightarrow Y \rightarrow Z) \rightarrow \{ \kappa \div \text{clk} \} \rightarrow \text{gstr } \kappa X \rightarrow \text{gstr } \kappa Y \rightarrow \text{gstr } \kappa Z \\
gzipwith = \lambda f. \text{fix } F \text{ in } \lambda \alpha, \beta. \langle f \alpha.1 \beta.1, F \alpha.2 \beta.2 \rangle
\]

\[
\text{zipwith} : (f : X \rightarrow Y \rightarrow Z) \rightarrow \text{seq } X \rightarrow \text{seq } Y \rightarrow \text{seq } Z \\
\text{zipwith} = \text{gzipwith}
\]

\(^2\)Birkedal and Møgelberg [2013]
In Guarded Computational Type Theory / $\text{CTT}^\oplus$, we propose a guarded dependent type theory which supports a predicative hierarchy of standard/non-indexed type-theoretic universes $U_i$.

1. standard dependent type connectives ($\Pi, \Sigma, \text{Eq}$)
2. clock-indexed later modality ($\triangleright_{\kappa} A$)
3. parametric clock quantifier ($\{\kappa \div \text{clk}\} \rightarrow A$)
4. non-parametric clock quantifier ($\langle\kappa : \text{clk}\rangle \rightarrow A$)
5. type-theoretic universes $U_i$ which contain $\triangleright_{\kappa} A$
6. operational semantics, canonicity result
7. formalized in Coq using internal-topos-theoretic technique
(questions?)


Bibliography III


Bibliography VII

