Computer-Checked Recurrence Extraction for Functional Programs

Bowornmet Hudson

Thesis Committee:
Dan Licata
Norman Danner
Olivier Hermant

Wesleyan University

April 26, 2016

Big Picture

$$
\begin{aligned}
&\text{append} : \{A : \text{Set}\}\ (\text{l1 l2} : \text{List A}) \rightarrow \text{List A} \\
&\text{append } [\,]\ \text{ys} = \text{ys} \\
&\text{append (x :: xs) ys} = \text{x :: (append xs ys)}
\end{aligned}
$$

Big Picture

```
append : {A : Set} (l1 l2 : List A) → List A
append [] ys = ys
append (x :: xs) ys = x :: (append xs ys)
```

$$
\begin{aligned}
T_{\text{append}}(0) &= c_0 \\
T_{\text{append}}(Sn) &= c_1 + T_{\text{append}}(n)
\end{aligned}
$$

Big Picture

```
append : {A : Set} (l1 l2 : List A) → List A
append [] ys = ys
append (x :: xs) ys = x :: (append xs ys)
```

$$
\begin{aligned}
T_{\text{append}}(0) &= c_0 \\
T_{\text{append}}(Sn) &= c_1 + T_{\text{append}}(n)
\end{aligned}
$$

$$T_{\text{append}}(n) = c_1(n) + c_0$$

Big Picture

```
append : {A : Set} (l1 l2 : List A) → List A
append [] ys = ys
append (x :: xs) ys = x :: (append xs ys)
```

$$T_{\text{append}}(0) = c_0$$
$$T_{\text{append}}(Sn) = c_1 + T_{\text{append}}(n)$$

$$T_{\text{append}}(n) = c_1(n) + c_0$$

$$O(n)$$

Big Picture

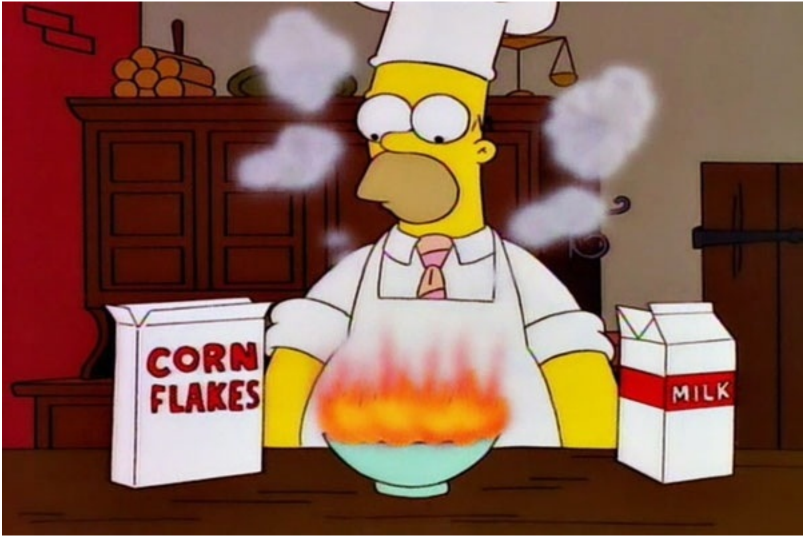

Figure: http://i.imgur.com/gzryb.jpg

Big Picture

- Automated complexity analysis...

Big Picture

- Automated complexity analysis...

...in Agda

Top-Down Reasoning

extract and formally reason about time complexity properties of functional programs

Bottom-Up Reasoning

proof assistants

Thesis Statement

It is possible to extract and formally reason about time complexity properties of functional programs using proof assistants

Our Approach

- Inspired by:
  - Danner, Paykin, Royer '13
  - Danner, Licata, Ramyaa '15

Our Approach

- Inspired by:
  - Danner, Paykin, Royer '13
  - Danner, Licata, Ramyaa '15
- Source Language

Our Approach

- Inspired by:
  - Danner, Paykin, Royer '13
  - Danner, Licata, Ramyaa '15
- Source Language $\rightarrow || \cdot ||$

Our Approach

- Inspired by:
  - Danner, Paykin, Royer '13
  - Danner, Licata, Ramyaa '15
- Source Language $\rightarrow || \cdot || \rightarrow$ Complexity Language

Our Approach

- Inspired by:
  - Danner, Paykin, Royer '13
  - Danner, Licata, Ramyaa '15
- Source Language $\rightarrow || \cdot || \rightarrow$ Complexity Language
- $|| \, e \, || = (E_c, E_p)$

Our Approach

- Inspired by:
  - Danner, Paykin, Royer '13
  - Danner, Licata, Ramyaa '15
- Source Language $\rightarrow || \cdot || \rightarrow$ Complexity Language
- $|| \, e \, || = (E_c, E_p)$
- Denotational Semantics

Our Approach

- Inspired by:
  - Danner, Paykin, Royer '13
  - Danner, Licata, Ramyaa '15
- Source Language $\rightarrow || \cdot || \rightarrow$ Complexity Language
- $|| \, e \, || = (E_c, E_p)$
- Denotational Semantics
- Everything formalized in Agda

Our Approach

- Inspired by:
  - Danner, Paykin, Royer '13
  - Danner, Licata, Ramyaa '15
- Source Language $\rightarrow || \cdot || \rightarrow$ Complexity Language
- $|| \, e \, || = (E_c, E_p)$
- Denotational Semantics
- Everything formalized in Agda
- 5,000+ lines of code

Our Approach

- Inspired by:
  - Danner, Paykin, Royer '13
  - Danner, Licata, Ramyaa '15
- Source Language $\rightarrow || \cdot || \rightarrow$ Complexity Language
- $|| \, e \, || = (E_c, E_p)$
- Denotational Semantics
- Everything formalized in Agda
- 5,000+ lines of code
- `https://github.com/benhuds/Agda/tree/master/complexity/complexity-final/submit`

Source Language: Types, Contexts, Variables

```
data Tp : Set where
  unit : Tp
  nat : Tp
  susp : Tp → Tp
  _->s_ : Tp → Tp → Tp
  _×s_ : Tp → Tp → Tp
  list : Tp → Tp
  bool : Tp
Ctx = List Tp
data _∈_ : Tp → Ctx → Set where
  i0 : ∀ {Γ τ} → τ ∈ τ :: Γ
  iS : ∀ {Γ τ τ1} → τ ∈ Γ → τ ∈ τ1 :: Γ
```

Source Language: Terms

```
data _|-_ : Ctx → Tp → Set where
  var : ∀ {Γ τ} → τ ∈ Γ → Γ |- τ
  rec : ∀ {Γ τ} → Γ |- nat → Γ |- τ → (nat :: (susp τ :: Γ)) |- τ
    → Γ |- τ
  lam : ∀ {Γ τ ρ} → (ρ :: Γ) |- τ
    → Γ |- (ρ ->s τ)
  app : ∀ {Γ τ1 τ2}
    → Γ |- (τ2 ->s τ1) → Γ |- τ2
    → Γ |- τ1
  force : ∀ {Γ τ}
    → Γ |- susp τ
    → Γ |- τ
  ...etc
```

Source Language: Substitutions and Renamings

```
sctx : Ctx → Ctx → Set
sctx Γ Γ' = ∀ {τ} → τ ∈ Γ' → Γ |- τ
subst : ∀ {Γ Γ' τ} → Γ' |- τ → sctx Γ Γ' → Γ |- τ
rctx : Ctx → Ctx → Set
rctx Γ Γ' = ∀ {τ} → τ ∈ Γ' → τ ∈ Γ
ren : ∀ {Γ Γ' τ} → Γ' |- τ → rctx Γ Γ' → Γ |- τ
```

Source Language: Substitutions and Renamings

```
_rr_  : ∀ {A B C} → rctx A B → rctx B C → rctx A C
_rr_ ρ1 ρ2 x  =  (ρ1 o ρ2) x
_rs_  : ∀ {A B C} → rctx A B → sctx B C → sctx A C
_rs_ ρ Θ x  =  ren (subst (var x) Θ) ρ
_ss_  : ∀ {A B C} → sctx A B → sctx B C → sctx A C
_ss_ Θ1 Θ2 x  =  subst (subst (var x) Θ2) Θ1
_sr_  : ∀ {A B C} → sctx A B → rctx B C → sctx A C
_sr_ Θ ρ x  =  subst (ren (var x) ρ) Θ
```

Source Language: Substitutions and Renamings

rr-comp : ∀ {Γ Γ' Γ'' τ} → (ρ1 : rctx Γ Γ') (ρ2 : rctx Γ' Γ'')
→ (e : Γ'' |- τ)
→ (ren (ren e ρ2) ρ1) ≡ (ren e (ρ1 rr ρ2))

rs-comp : ∀ {A B C τ} → (ρ : rctx C A) (Θ : sctx A B)
→ (e : B |- τ)
→ ren (subst e Θ) ρ ≡ subst e (ρ rs Θ)

sr-comp : ∀ {Γ Γ' Γ'' τ} → (Θ : sctx Γ Γ') → (ρ : rctx Γ' Γ'')
→ (e : Γ'' |- τ)
→ (subst (ren e ρ) Θ) ≡ subst e (Θ sr ρ)

ss-comp : ∀ {A B C τ} → (Θ1 : sctx A B) (Θ2 : sctx B C)
→ (e : C |- τ)
→ subst e (Θ1 ss Θ2) ≡ subst (subst e Θ2) Θ1

Source Language: Cost Semantics

- Idea: add a cost measure to every evaluation relation
- Blelloch, Greiner '96

Source Language: Cost Semantics

- Idea: add a cost measure to every evaluation relation
- Blelloch, Greiner '96

$$\frac{}{e \downarrow^{n} v}$$

"e evaluates to v in n steps"

Source Language: Cost Semantics

- Idea: add a cost measure to every evaluation relation
- Blelloch, Greiner '96

$$\frac{}{e \downarrow^{n} v}$$

"e evaluates to v in n steps"

data evals : {τ : Tp} → [] |- τ → [] |- τ → Cost → Set **where**

Source Language: Cost Semantics

Example:

$$\frac{e1 \downarrow^{n1} v1 \qquad e2 \downarrow^{n2} v2}{\langle e1, e2 \rangle \downarrow^{n1+n2} \langle v1, v2 \rangle}$$

Source Language: Cost Semantics

Example:

$$\frac{e1 \downarrow^{n1} v1 \qquad e2 \downarrow^{n2} v2}{\langle e1, e2 \rangle \downarrow^{n1+n2} \langle v1, v2 \rangle}$$

```
pair-evals : ∀ {n1 n2}
  → {τ1 τ2 : Tp} {e1 v1 : [] |- τ1} {e2 v2 : [] |- τ2}
  → evals e1 v1 n1
  → evals e2 v2 n2
  → evals (prod e1 e2) (prod v1 v2) (n1 +c n2)
```

Complexity Language

- Recall $|| \cdot ||$ returns a cost-potential pair
- Gives us exact recurrences

Complexity Language

- Recall $|| \cdot ||$ returns a cost-potential pair
- Gives us exact recurrences
- Want: a setting where we can reason about costs directly
- Massage recurrences into closed forms/asymptotic bounds
- No need to refer to denotational semantics

Complexity Language: Types

data CTp : Set **where**

  ...

Complexity Language: Types

data CTp : Set **where**

...

- ~~susp~~

Complexity Language: Types

data CTp : Set **where**
  ...

- ~~susp~~
- Abstract costs in source language = C in complexity language

Complexity Language: Types

data CTp : Set **where**

…

- ~~susp~~
- Abstract costs in source language = C in complexity language
- rnat
  - Recursor equipped with proof of monotonicity

Complexity Language: Types

```
data CTp : Set where
  ...
```

- ~~susp~~
- Abstract costs in source language = C in complexity language
- rnat
  - Recursor equipped with proof of monotonicity
- ‘max’ types: notion of maximums

```
data CTpM : CTp → Set where
  rnat-max : CTpM rnat
  _×cm_ : ∀ {τ1 τ2}
    → CTpM τ1 → CTpM τ2 → CTpM (τ1 ×c τ2)
  _->cm_ : ∀ {τ1 τ2}
    → CTpM τ2 → CTpM (τ1 ->c τ2)
```

Complexity Language: Terms

$$\frac{}{\Gamma \vdash 0C : C}$$

$$\frac{}{\Gamma \vdash 1C : C}$$

$$\frac{\Gamma \vdash C1 : C \qquad \Gamma \vdash C2 : C}{\Gamma \vdash C1 + C2 : C}$$

Complexity Language: Terms

$$\frac{}{\Gamma \vdash \mathsf{0C} : \mathsf{C}}$$

$$\frac{}{\Gamma \vdash \mathsf{1C} : \mathsf{C}}$$

$$\frac{\Gamma \vdash \mathsf{C1} : \mathsf{C} \qquad \Gamma \vdash \mathsf{C2} : \mathsf{C}}{\Gamma \vdash \mathsf{C1} + \mathsf{C2} : \mathsf{C}}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{rnat} \qquad \Gamma \vdash \mathsf{e0} : \tau \qquad \Gamma \vdash \mathsf{e1} : \mathsf{rnat} \text{ -}{>}\mathsf{c}\ (\tau \text{ -}{>}\mathsf{c}\ \tau) \qquad \mathsf{e0} \leqslant\mathsf{s}\ \mathsf{e1}}{\Gamma \vdash \mathsf{rec}(\mathsf{e}\ ,\ \mathsf{e0}\ ,\ \mathsf{e1}) : \tau}$$

Complexity Language: Terms

$$\frac{}{\Gamma \vdash \mathsf{0C} : \mathsf{C}}$$

$$\frac{}{\Gamma \vdash \mathsf{1C} : \mathsf{C}}$$

$$\frac{\Gamma \vdash \mathsf{C1} : \mathsf{C} \qquad \Gamma \vdash \mathsf{C2} : \mathsf{C}}{\Gamma \vdash \mathsf{C1} + \mathsf{C2} : \mathsf{C}}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{rnat} \qquad \Gamma \vdash \mathsf{e0} : \tau \qquad \Gamma \vdash \mathsf{e1} : \mathsf{rnat} \text{ ->c } (\tau \text{ ->c } \tau) \qquad \mathsf{e0} \leqslant \mathsf{s}\ \mathsf{e1}}{\Gamma \vdash \mathsf{rec}(\mathsf{e}\ ,\ \mathsf{e0}\ ,\ \mathsf{e1}) : \tau}$$

$$\frac{\tau \text{ has maxes} \qquad \Gamma \vdash \mathsf{e1} : \tau \qquad \Gamma \vdash \mathsf{e2} : \tau}{\Gamma \vdash \mathsf{max}(\mathsf{e1},\mathsf{e2}) : \tau}$$

Complexity Language: Abstract Preorder Judgement

- $\_ \leqslant_{\mathsf{s}} \_$ judgement specifies ordering on terms

Complexity Language: Abstract Preorder Judgement

- _⩽s_ judgement specifies ordering on terms

```
data _⩽s_ where
  refl-s : ∀ {Γ T} → {e : Γ |- T} → e ⩽s e
  trans-s : ∀ {Γ T} → {e e' e'' : Γ |- T}
    → e ⩽s e' → e' ⩽s e'' → e ⩽s e''
  cong-app : ∀ {Γ τ τ'} {e e' : Γ |- (τ ->c τ')} {e1 : Γ |- τ}
           → e ⩽s e' → app e e1 ⩽s app e' e1
  l-proj-s : ∀ {Γ T1 T2} → {e1 : Γ |- T1} {e2 : Γ |- T2}
    → e1 ⩽s l-proj (prod e1 e2)
```

Complexity Language: Future

$$\begin{tikzcd}
 & \mathsf{rnat} \arrow[dr, dashed, "T'"] & \\
\mathsf{List} \arrow[ur, "\mathsf{length}"] \arrow[rr, "||f||"] & & \mathsf{C} \times _
\end{tikzcd}$$

Translation

$$
\begin{aligned}
||\_|| &: \mathsf{Tp} \to \mathsf{CTp} \\
||\tau|| &= \mathsf{C} \times\mathsf{c}\ \langle\langle\ \tau\ \rangle\rangle
\end{aligned}
$$

$$
\begin{aligned}
&||\_||\mathsf{e}\ :\ \forall\ \{\Gamma\ \tau\} \to \Gamma\ \mathsf{Source.|\text{-}}\ \tau \to \langle\langle\ \Gamma\ \rangle\rangle\mathsf{c}\ \mathsf{Complexity.|\text{-}}\ ||\tau|| \\
&||\mathsf{var\ x}||\mathsf{e}\ =\ \mathsf{prod\ 0\ C\ (var\ (lookup\ x))} \\
&||\mathsf{z}||\mathsf{e}\ =\ \mathsf{prod\ 0\ C\ z} \\
&||\mathsf{suc\ e}||\mathsf{e}\ =\ \mathsf{prod\ (l\text{-}proj\ (}||\mathsf{e}||\mathsf{e}))\ (\mathsf{s}\ (\mathsf{r\text{-}proj}\ (||\mathsf{e}||\mathsf{e}))) \\
&||\mathsf{rec\ e\ e0\ e1}||\mathsf{e}\ = \\
&\quad (\mathsf{l\text{-}proj}\ (||\mathsf{e}||\mathsf{e}))\ +\mathsf{C} \\
&\quad (\mathsf{rec}\ (\mathsf{r\text{-}proj}\ ||\mathsf{e}||\mathsf{e})\ (1\ \mathsf{C}\ +\mathsf{C}\ ||\mathsf{e0}||\mathsf{e})\ (1\ \mathsf{C}\ +\mathsf{C}\ ||\mathsf{e1}||\mathsf{e})) \\
&||\mathsf{lam\ e}||\mathsf{e}\ =\ \mathsf{prod\ 0\ C\ (lam}\ ||\mathsf{e}||\mathsf{e})
\end{aligned}
$$

Translation: Bounding

- $e \sqsubseteq E$: if $e \downarrow^n v$, then $n \leqslant E_c$ and $v \sqsubseteq^{val} E_p$

Translation: Bounding

- $e \sqsubseteq E$: if $e \downarrow^n v$, then $n \leqslant E_c$ and $v \sqsubseteq^{val} E_p$
- Theorem: If $\Gamma \vdash \mathrm{e} : \tau$, θ is a substitution of all variables in a source context, and Θ is a corresponding substitution of all variables in a complexity context, then $\mathrm{e}[\theta] \sqsubseteq_\tau ||\mathrm{e}||[\Theta]$.

Translation: Bounding

- $e \sqsubseteq E$: if $e \downarrow^n v$, then $n \leqslant E_c$ and $v \sqsubseteq^{val} E_p$
- Theorem: If $\Gamma \vdash \mathrm{e} : \tau$, θ is a substitution of all variables in a source context, and Θ is a corresponding substitution of all variables in a complexity context, then $\mathrm{e}[\theta] \sqsubseteq_\tau ||\mathrm{e}||[\Theta]$.
- Costs extracted by $|| \cdot ||$ are an upper bound on costs specified by source operational semantics

Translation: Bounding

- $e \sqsubseteq E$: if $e \downarrow^n v$, then $n \leqslant E_c$ and $v \sqsubseteq^{val} E_p$
- Theorem: If $\Gamma \vdash \mathrm{e} : \tau$, θ is a substitution of all variables in a source context, and Θ is a corresponding substitution of all variables in a complexity context, then $\mathrm{e}[\theta] \sqsubseteq_\tau ||\mathrm{e}||[\Theta]$.
- Costs extracted by $|| \cdot ||$ are an upper bound on costs specified by source operational semantics
- Detailed account of proof in previous thesis (150 lines of Agda, excluding lemmas and definitions)

Denotational Semantics

- Types = Preorders
- Terms = Monotone maps between preorders

Denotational Semantics: Preorders

- Preorder $= (A, \leqslant)$

Denotational Semantics: Preorders

- Preorder $= (A, \leqslant)$
- Reflexive: $\forall x,\ x \leqslant x$
- Transitive: $\forall xyz$, if $x \leqslant y$ and $y \leqslant z$, then $x \leqslant z$

Denotational Semantics: Preorders

- Preorder $= (A, \leqslant)$
- Reflexive: $\forall x,\ x \leqslant x$
- Transitive: $\forall xyz$, if $x \leqslant y$ and $y \leqslant z$, then $x \leqslant z$

```
record Preorder-str (A : Set) : Set1 where
  constructor preorder
  field
    _≤_  : A → A → Set
    refl : ∀ x → x ≤ x
    trans : ∀ x y z → x ≤ y → y ≤ z → x ≤ z
```

Denotational Semantics: Preorders

- Preorder $= (A, \leqslant)$
- Reflexive: $\forall x,\ x \leqslant x$
- Transitive: $\forall xyz$, if $x \leqslant y$ and $y \leqslant z$, then $x \leqslant z$

```
record Preorder-str (A : Set) : Set1 where
  constructor preorder
  field
    _≤_  : A → A → Set
    refl  : ∀ x → x ≤ x
    trans  : ∀ x y z → x ≤ y → y ≤ z → x ≤ z
```

PREORDER = Σ (λ (A : Set) → Preorder-str A)

Interpretation of Types

$[\_]t \; : \; \mathsf{CTp} \to \mathsf{PREORDER}$
$[\; \mathsf{nat}\;]t \;=\; \mathsf{Nat} \;,\; \flat\mathsf{nat\text{-}p}$
$[\; \tau \;\text{->c}\; \tau_1 \;]t \;=\; [\; \tau \;]t \;\text{->p}\; [\; \tau_1 \;]t$
$[\; \tau \times\mathsf{c}\; \tau_1 \;]t \;=\; [\; \tau \;]t \times\mathsf{p}\; [\; \tau_1 \;]t$
$[\; \mathsf{list}\; \tau \;]t \;=\; (\mathsf{List}\; (\mathsf{fst}\; [\; \tau \;]t)) \;,\; \mathsf{list\text{-}p}\; (\mathsf{snd}\; [\; \tau \;]t)$
$[\; \mathsf{bool} \;]t \;=\; \mathsf{Bool} \;,\; \mathsf{bool\text{-}p}$
$[\; \mathsf{C} \;]t \;=\; \mathsf{Nat} \;,\; \mathsf{nat\text{-}p}$
$[\; \mathsf{rnat} \;]t \;=\; \mathsf{Nat} \;,\; \mathsf{nat\text{-}p}$

Denotational Semantics: Monotone Functions

- $(A, \leqslant_A)$, $(B, \leqslant_B)$
- $f : A \to B$ is monotone if $\forall x, y \in A$, if $x \leqslant_A y$, then $f(x) \leqslant_B f(y)$

Denotational Semantics: Monotone Functions

- $(A, \leqslant_A)$, $(B, \leqslant_B)$
- $f : A \to B$ is monotone if $\forall x, y \in A$, if $x \leqslant_A y$, then $f(x) \leqslant_B f(y)$

```
record Monotone (A : Set) (B : Set)
        (PA : Preorder-str A) (PB : Preorder-str B) : Set where
  constructor monotone
  field
    f : A → B
    is-monotone : ∀ (x y : A)
      → Preorder-str.≤ PA x y
      → Preorder-str.≤ PB (f x) (f y)
```

Denotational Semantics: Monotone Functions

- $(A, \leqslant_A)$, $(B, \leqslant_B)$
- $f : A \to B$ is monotone if $\forall x, y \in A$, if $x \leqslant_A y$, then $f(x) \leqslant_B f(y)$

```
record Monotone (A : Set) (B : Set)
        (PA : Preorder-str A) (PB : Preorder-str B) : Set where
  constructor monotone
  field
    f : A → B
    is-monotone : ∀ (x y : A)
      → Preorder-str.≤ PA x y
      → Preorder-str.≤ PB (f x) (f y)
```

```
MONOTONE : (PΓ PA : PREORDER) → Set
MONOTONE (Γ , PΓ) (A , PA) = Monotone Γ A PΓ PA
```

Characterizing Terms as Monotone Functions

- Must know how to characterize terms as monotone functions between preorders

E.g. cartesian products as monotone functions:

```
pair' : ∀ {PΓ PA PB}
  → MONOTONE PΓ PA
  → MONOTONE PΓ PB
  → MONOTONE PΓ (PA ×p PB)
pair' (monotone f f-ismono) (monotone g g-ismono) =
  monotone (λ x → f x , g x)
    (λ x y z → f-ismono x y z , g-ismono x y z)
```

Interpretation of Terms

$$
\begin{aligned}
&\text{interpE} \;:\; \forall\, \{\Gamma\ \tau\} \to \Gamma \mid\!\text{-}\ \tau \to \text{el}\ ([\ \Gamma\ ]\text{c} \text{ -}\!>\!\text{p } [\ \tau\ ]\text{t})\\
&\text{interpE (lam e)} \;=\; \text{lam' (interpE e)}\\
&\text{interpE (app e } e_1) \;=\; \text{app' (interpE e) (interpE } e_1)\\
&\text{interpE (prod e } e_1) \;=\; \text{pair' (interpE e) (interpE } e_1)\\
&\text{interpE (l-proj e)} \;=\; \text{fst' (interpE e)}\\
&\text{interpE (r-proj e)} \;=\; \text{snd' (interpE e)}\\
&\text{interpE nil} \;=\; \text{nil'}\\
&\text{interpE (e ::c } e_1) \;=\; \text{cons' (interpE e) (interpE } e_1)
\end{aligned}
$$

Soundness

- Complexity language is sound with respect to the interpretation we give!

Soundness

- Complexity language is sound with respect to the interpretation we give!

$\forall\ \Gamma \vdash e : \tau$, $\Gamma \vdash e' : \tau$, and elements $k \in \Gamma$,
if $e \leqslant_{\mathsf{s}} e'$
then $[\![e]\!]k \leqslant_{[\![\tau]\!]} [\![e']\!]k$.

Soundness

- Complexity language is sound with respect to the interpretation we give!

$\forall\ \Gamma \vdash e : \tau$, $\Gamma \vdash e' : \tau$, and elements $k \in \Gamma$,
if $e \leqslant$s e'
then $[\![e]\!]k \leqslant_{[\![\tau]\!]} [\![e']\!]k$.

```
sound : ∀ {Γ τ} (e e' : Γ |- τ)
   → e ≤s e'
   → PREORDER≤ ([ Γ ]c ->p [ τ ]t) (interpE e) (interpE e')
```

A Case of Soundness

```
sound {Γ} {τ} .(subst e (lem3' (lem3' (lem3' Θ v3) v2) v1)) ._ (subst-compose5-r {.Γ} {Γ'} {.τ} {τ1} {τ2} {τ3} Θ e v1 v2 v3) k =
  Preorder-str.trans (snd [ τ ]t)
    (Monotone.f (interpE (subst e (lem3' (lem3' (lem3' Θ v3) v2) v1))) k)
    (Monotone.f (interpE (subst e (s-extend (s-extend (s-extend Θ))))) (Monotone.f (interpS (lem3' (lem3' (lem3' ids v3) v2) v1)) k))
    (Monotone.f (interpE (subst (subst e (s-extend (s-extend (s-extend Θ)))) (lem3' (lem3' (lem3' ids v3) v2) v1))) k)
    (Preorder-str.trans (snd [ τ ]t)
      (Monotone.f (interpE (subst e (lem3' (lem3' (lem3' Θ v3) v2) v1))) k)
      (Monotone.f (interpE e) (Monotone.f (interpS (lem3' (lem3' (lem3' Θ v3) v2) v1)) k))
      (Monotone.f (interpE (subst e (s-extend (s-extend (s-extend Θ))))) (Monotone.f (interpS (lem3' (lem3' (lem3' ids v3) v2) v1)) k))
      (subst-eq-l (lem3' (lem3' (lem3' Θ v3) v2) v1) e k)
      (Preorder-str.trans (snd [ τ ]t)
        (Monotone.f (interpE e) (Monotone.f (interpS (lem3' (lem3' (lem3' Θ v3) v2) v1)) k))
        (Monotone.f (interpE e) (Monotone.f (interpS {τ1 :: τ2 :: τ3 :: Γ} {τ1 :: τ2 :: τ3 :: Γ'} (s-extend (s-extend (s-extend Θ))))
          (Monotone.f (interpS {Γ} {τ1 :: τ2 :: τ3 :: Γ} (lem3' (lem3' (lem3' ids v3) v2) v1)) k)))
        (Monotone.f (interpE (subst e (s-extend (s-extend (s-extend Θ))))) (Monotone.f (interpS (lem3' (lem3' (lem3' ids v3) v2) v1)) k))
        (Monotone.is-monotone (interpE e)
          (Monotone.f (interpS (lem3' (lem3' (lem3' Θ v3) v2) v1)) k)
          (Monotone.f (interpS {τ1 :: τ2 :: τ3 :: Γ} {τ1 :: τ2 :: τ3 :: Γ'} (s-extend (s-extend (s-extend Θ))))
            (Monotone.f (interpS {Γ} {τ1 :: τ2 :: τ3 :: Γ} (lem3' (lem3' (lem3' ids v3) v2) v1)) k))
            ((((Preorder-str.trans (snd [ Γ' ]c)
              (Monotone.f (interpS Θ) k)
              (Monotone.f (interpS (λ x → subst (ren (Θ x) iS) (lem3' ids v3))) k)
              (Monotone.f (interpS (λ x → ren (ren (ren (Θ x) iS) iS) iS))
                (((Monotone.f (interpS {Γ} ids) k , Monotone.f (interpE v3) k) , Monotone.f (interpE v2) k) , Monotone.f (interpE v1) k))
              (interp-subst-comp-r Θ v3 k)
              (Preorder-str.trans (snd [ Γ' ]c)
                (Monotone.f (interpS (λ x → subst (ren (Θ x) iS) (lem3' ids v3))) k)
                (Monotone.f (interpS (λ x → ren (ren (Θ x) iS) iS)) ((Monotone.f (interpS {Γ} ids) k , Monotone.f (interpE v3) k) , Monotone.f (interpE v2) k))
                (Monotone.f (interpS (λ x → ren (ren (ren (Θ x) iS) iS) iS))
                  (((Monotone.f (interpS {Γ} ids) k , Monotone.f (interpE v3) k) , Monotone.f (interpE v2) k) , Monotone.f (interpE v1) k))
                (interp-subst-comp2-r Θ k v2 v3)
                (interp-subst-comp3-r Θ k v3 v2 v1))) ,
            (Preorder-str.refl (snd [ τ3 ]t) (Monotone.f (interpE v3) k))) ,
            (Preorder-str.refl (snd [ τ2 ]t) (Monotone.f (interpE v2) k))) ,
            (Preorder-str.refl (snd [ τ1 ]t) (Monotone.f (interpE v1) k))))
        (subst-eq-r (s-extend (s-extend (s-extend Θ))) e (Monotone.f (interpS (lem3' (lem3' (lem3' ids v3) v2) v1)) k))))
    (subst-eq-r (lem3' (lem3' (lem3' ids v3) v2) v1) (subst e (s-extend (s-extend (s-extend Θ)))) k)
```

(full proof including lemmas is 1600 lines of code)

Example: dbl

```
{- dbl (n : nat) : nat = 2 * n -}
dbl : ∀ {Γ} → Γ Source.|- (nat ->s nat)
dbl = lam (rec (var i0) z (suc (suc (force (var (iS i0))))))
```

Example: dbl

```
{- dbl (n : nat) : nat = 2 * n -}
dbl : ∀ {Γ} → Γ Source.|- (nat ->s nat)
dbl = lam (rec (var i0) z (suc (suc (force (var (iS i0))))))
```

Ideally, we want a recurrence of the form

$$T_{dbl}(0) = c_0$$
$$T_{dbl}(Sn) = c_1 + T_{dbl}(n)$$

Example: dbl

```
{- dbl (n : nat) : nat = 2 * n -}
dbl : ∀ {Γ} → Γ Source.|- (nat ->s nat)
dbl = lam (rec (var i0) z (suc (suc (force (var (iS i0))))))
```

Ideally, we want a recurrence of the form

$$T_{dbl}(0) = c_0$$
$$T_{dbl}(Sn) = c_1 + T_{dbl}(n)$$

```
prod 0C
(lam
 (letc
  (letc
   (prod (plusC (l-proj (var (iS i0))) (l-proj (var i0)))
    (r-proj (var i0)))
   (rec (r-proj (var i0))
    (letc (prod (plusC 1C (l-proj (var i0))) (r-proj (var i0)))
     (prod 0C z))
    (letc (prod (plusC 1C (l-proj (var i0))) (r-proj (var i0)))
     (letc (prod (l-proj (var i0)) (s (r-proj (var i0))))
      (letc (prod (l-proj (var i0)) (s (r-proj (var i0))))
       (letc
        (prod (plusC (l-proj (var i0)) (l-proj (r-proj (var i0))))
         (r-proj (r-proj (var i0))))
        (prod 0C (var (iS i0)))))))))
  (prod 0C (var i0))))
```

Example: dbl

Semantics gives us:

```
monotone
(λ x → 0 ,
   monotone
   (λ p₁ →
      (natrec (1 , 0)
      (λ n x₂ → S (fst x₂) , S (S (snd x₂))) p₁)) ,
   (λ a b c → ERASED))
(λ x y z₁ → ERASED)
```

Example: dbl

Semantics gives us:

```
monotone
(λ x → 0 ,
   monotone
   (λ p₁ →
      (natrec (1 , 0)
      (λ n x₂ → S (fst x₂) , S (S (snd x₂))) p₁)) ,
   (λ a b c → ERASED))
(λ x y z₁ → ERASED)
```

Corresponds to a complexity recurrence of the form:

$$|| \, dbl(Z) \, ||_c = 1$$
$$|| \, dbl(Sn) \, ||_c = 1 + || \, dbl(n) \, ||_c$$

Example: dbl

Semantics gives us:

```
monotone
(λ x → 0 ,
   monotone
   (λ p₁ →
      (natrec (1 , 0)
      (λ n x₂ → S (fst x₂) , S (S (snd x₂))) p₁)) ,
   (λ a b c → ERASED))
(λ x y z₁ → ERASED)
```

Corresponds to a complexity recurrence of the form:

$$\| \, dbl(Z) \, \|_c = 1$$
$$\| \, dbl(Sn) \, \|_c = 1 + \| \, dbl(n) \, \|_c$$

Compare with:

$$T_{dbl}(0) = c_0$$
$$T_{dbl}(Sn) = c_1 + T_{dbl}(n)$$

Contributions

- End-to-end system for recurrence extraction in Agda:
  - Source and complexity languages (Ch. 2)
  - $|| \cdot ||$ (Ch.3)
  - Denotational semantics for complexity language (Ch. 4)
- Correctness properties:
  - bounding (Ch. 3)
  - soundness (Ch. 4)

Thesis Statement

It is possible to extract and formally reason about time complexity properties of functional programs using proof assistants

Future Work

- Adding rules to $\leqslant$s

Future Work

- Adding rules to $\leqslant$s
- Rewriting within the complexity language

Future Work

- Adding rules to $\leqslant$s
- Rewriting within the complexity language
- User-friendliness

Thanks to

- My advisor, Dan Licata

Thanks to

- My advisor, Dan Licata
- My committee, Norman Danner and Olivier Hermant

Thanks to

- My advisor, Dan Licata
- My committee, Norman Danner and Olivier Hermant
- My colleagues, Ted and Justin

Thanks to

- My advisor, Dan Licata
- My committee, Norman Danner and Olivier Hermant
- My colleagues, Ted and Justin
- My friends

Thanks to

- My advisor, Dan Licata
- My committee, Norman Danner and Olivier Hermant
- My colleagues, Ted and Justin
- My friends
- My parents