Local temporal reasoning

Eric Koskinen
Visiting Assistant Professor
New York University

Tachio Terauchi
Professor
JAIST

15 July 2014

Supported by the Japan Society for the Advancement of Science (JSPS), MEXT Kakenhi, and a grant from the NYU Office of the Provost.
let rec halt _ =
  halt ()
and shrink f =
  if ( f() = 0 ) then
    halt ()
  else
    shrink (λ_. f() - 1)
and main() =
  let t = *+ in
  shrink (λ_. t)
let rec halt _ = \_ \rightarrow ev[halt];
    halt ()
and shrink f = \_ \rightarrow ev[shrink];
    if ( f() = 0 ) then
        halt ()
    else
        shrink (λ_. f() - 1)
and main() = \_ \rightarrow ev[main];
    let t = *\_ \rightarrow ev[halt];
    in
    shrink (λ_. t)

main X (shrink U halt)
let rec halt _ =
  halt ()
and shrink f =
  if ( f() = 0 ) then
    halt ()
  else
    shrink (λ_. f() - 1)

and main() =
  let t = #+ in
  shrink (λ_. t)

main X (shrink θ halt)
let rec halt _ =
    halt ()
and shrink f =
    if ( f() = 0 ) then
        halt ()
    else
        shrink (\_. f() - 1)
and main() =
    let t = *+ in
    shrink (\_. t)

main X (shrink U halt)
No previous technique can prove this property.

Previously:

• Expressive logics, but finite data [K/O:LICS’09]
• Infinite data, but just safety [Terauchi:POPL’10]
• Infinite data, but just termination [K/T/U/K:ESOP’14]
• Expressive logics, but first-order programs [CK:PLDI’13]
Terminates or diverges?

\[ e_1 \quad e_2 \]

Terminates or diverges?

\[ \text{shrink} \ (\lambda_. \ t) \]

(shrink U halt)
Decompose in two ways

1. Divide up program into expressions

$e_1 \quad e_2$

- $\Phi_1$: Temporal behavior as $e_1$ is reduced
- $\Phi_2$: Temporal behavior as $e_2$ is reduced
- $\Phi_3$: Latent behavior during application

$\Phi_1 \cdot \Phi_2 \cdot \Phi_3$
Decompose in two ways

1. Divide up program into expressions

Characterize temporal behavior of exprs. via type-and-effect:

\[ \Gamma \vdash e_1 \; e_2 : \tau \; \& \; \Phi \]

- Typing environment
- Dependent Type
- Temporal Effect
Decompose in two ways

1. Divide up program into expressions

Characterize temporal behavior of exprs. via type-and-effect:

\[ \Gamma \vdash e_1 : \tau \& \Phi \]

Closed Under:
- Union: \( \Phi_1 \cup \Phi_2 \)
- Isect.: \( \Phi_1 \cap \Phi_2 \)
- Comp.: \( \Phi_1 \cdot \Phi_2 \)

e.g. Buchi automata
Decompose in two ways

1. Divide up program into expressions

2. Track behavior of finite traces separate from infinite traces

\[ e_1 \cdot e_2 \]

\[ \Phi_1 \]
Temporal behavior as \( e_1 \) is reduced

\[ \Phi_2 \]
Temporal behavior as \( e_2 \) is reduced

\[ \Phi_3 \]
Latent behavior during application
Decompose in two ways

1. Divide up program into expressions

2. Track behavior of finite traces separate from infinite traces

\[
\Phi_{\text{fin}}^1, \Phi_{\text{inf}}^1 \quad \text{Temporal behavior as } e_1 \text{ is reduced}
\]

\[
\Phi_{\text{fin}}^3, \Phi_{\text{inf}}^3 \quad \text{Latent behavior during application}
\]

\[
(\Phi_{\text{fin}}^1, \Phi_{\text{inf}}^1) \cdot (\Phi_{\text{fin}}^2, \Phi_{\text{inf}}^2) \cdot (\Phi_{\text{fin}}^3, \Phi_{\text{inf}}^3)
\]

\[
\Phi_{\text{fin}}^2, \Phi_{\text{inf}}^2 \quad \text{Temporal behavior as } e_2 \text{ is reduced}
\]
Decompose in two ways

1. Divide up program into expressions

2. Track behavior of finite traces separate from infinite traces

\[ \Gamma \vdash e : \tau \& (\Phi^\text{fin}, \Phi^\text{inf}) \]
let rec halt _ =
  halt ()
and shrink f =
  if ( f() = 0 ) then
    halt ()
  else
    shrink (λ_. f() - 1)

and main() =
  let t = *^ in
  shrink (λ_. t)

main X (shrink U halt)
let rec halt _ =
    halt ()
and shrink f =
    if ( f() = 0 ) then
        halt ()
    else
        shrink (λ_. f() - 1)
and main() =
    let t = *+ in
    shrink (λ_. t)

Safety: the latent behavior when shrink is applied

Γ ⊢ shrink : (unit → int) \xrightarrow{\text{shrink} \lor \text{halt}} unit \& (\varepsilon, \varepsilon)
let rec hal _ =
  hal ()
and shrink f =
  if ( f() = 0 ) then
    hal ()
  else
    shrink (λ_. f() - 1)
and main() =
  let t = *+ in
  shrink (λ_. t)
  (shrink U halt)

Safety: the latent behavior when shrink is applied

Γ ⊢ shrink : (unit → int) ◻ shrink W halt → unit & (⊤, ε)

Liveness: the conditions under which shrink terminates

Γ, f : unit → {i | i ≥ 0} ⊢ shrink f : unit & (⊤, F ¬ shrink)
let rec halt _ =
  halt ()
and shrink f =
  if ( f() = 0 ) then
    halt ()
  else
    shrink (λ_. f() - 1)
and main() =
  let t = *+ in
  shrink (λ_. t)

Γ,t:{i | i ≥ 0} ⊢ shrink (λ_. t):unit & (shrink U halt)
Liveness in a type system
Where do these pieces come from?

Safety

... App

Liveness

Comb
Safety via the type system

\[ \Gamma \vdash \text{shrink} : (\text{unit} \to \text{int}) \xrightarrow{\text{shrink \& \text{halt}}} \text{unit} \& (\varepsilon, \varepsilon) \]

This arises as a fixpoint solution to the typing context in the judgments over the body of \text{shrink}.

First, assume we already an env. $\Gamma$ such that

\[ \Gamma(\text{halt}) = \text{unit} \xrightarrow{(\perp, G \text{ halt})} \text{unit} \& (\varepsilon, \varepsilon) \]
let rec halt _ = ev[halt];
    halt ()
and shrink f = ev[shrink];
    if ( f() = 0 ) then
        halt ()
    else
        shrink (λ_. f() - 1)
and main() = ev[main];
    let t = *+ in
        shrink (λ_. t)
let rec halt _ = ev[halt];
    
halt ()
and shrink f = ev[shrink];
    
    if ( f() = 0 ) then
         halt ()
    else
         shrink (\_. f() - 1)

and main() = ev[main];
    
let t = *+ in
    shrink (\_. t)

Depending on which branch, either shrink or halt will occur.

Valid typing:  shrink \lor halt

Or look at a fixpoint to:  \alpha = shrink \land X(\alpha \lor G \text{halt})
let rec halt _ = ev[halt];
    halt ()
and shrink f = ev[shrink];
    if ( f() = 0 ) then
        halt ()
    else
        shrink (λ_. f() - 1)
and main() = ev[main];
    let t = *+ in
    shrink (λ_. t)

Or look at a fixpoint to: \( \alpha = shrink \land X(\alpha \lor G\text{halt}) \)

\[ \Gamma,\text{halt} : ... \vdash shrink f : \text{unit} \land shrink W (G\text{halt}) \]
Liveness
let rec halt _ = ev[halt];
    halt ()
and shrink f = ev[shrink];
    if ( f() = 0 ) then
        halt ()
    else
        shrink (λ_. f() - 1)
and main() = ev[main];
    let t = *+ in
        shrink (λ_. t)

If \( f \) is a fun that returns 0, \( \text{halt} \) will be invoked

**Under what conditions does \( \text{shrink} \) terminate?**
If \( f \) is a fun that returns 0, \( \text{halt} \) will be invoked.

Under what conditions does \( \text{shrink} \) terminate?

But this is a separate proof

Adapt prior work on higher-order termination (ESOP 2014) to prove *conditional* higher-order termination.

\[
\Gamma, f : \text{unit} \to \{ i \mid i \geq 0 \} \vdash \text{shrink} f : \text{unit} \& (\top, F \not\vdash \text{shrink})
\]
Let rec halt _ = ev[halt];
    halt ()
and shrink f = ev[shrink];
if ( f() = 0 ) then
    halt ()
else
    shrink (λ_. f() - 1)
and main() = ev[main];
let t = *+ in
shrink (λ_. t)

Under what conditions does shrink terminate?

Adapt prior work on higher-order termination (ESOP'2014) to prove conditional higher-order termination.

Γ, f: unit → {i | i ≥ 0} ⊢ shrink f: unit & (T, F ⊬ shrink)
let rec halt _ = ev[halt];
    halt ()
and shrink f = ev[shrink];
    if ( f() = 0 ) then
        halt ()
    else
        shrink (λ_. f() - 1)
and main() = ev[main];
    let t = *+ in
    shrink (λ_. t)
Contributions and benefits

• First technique for temporal properties of higher-order, infinite-data programs

• Instantiation to wide variety of spec. logics, Instantiation to type environments, Instantiation to oracles

• Compositional

• Does not require input program in CPS

• First-order interprocedural programs
Language

\begin{align*}
P & ::= P \cup \{ F \ \overline{x} = e \} \mid \emptyset \\
\alpha & ::= \Sigma \\
e & ::= x \mid c \mid F \mid ev[\alpha] \mid \text{let } x = e_1 \text{ in } e_2 \mid xy \\
& \quad \mid x \ op \ y \mid \text{if } x \text{ then } e_1 \text{ else } e_2 \mid \lambda x. e
\end{align*}

Inductive big-step semantics for terminating runs:
\[ e \downarrow_P \nu \& \varpi \quad \varpi \in \Sigma^* \]

Co-inductive big-step semantics for non-terminating runs:
\[ e \uparrow_P \bot \& \pi \quad \pi \in \Sigma^\omega \]
Our semantics has no infinite, invisible computations.

\[
\begin{array}{c}
e_{v[x]} \Downarrow_P () & a \\
\text{ev[a]} \Downarrow_P () & a
\end{array}
\]

\[
\frac{e[v/x] \Downarrow_P v' & \bar{\omega}}{(\lambda x.e) v \Downarrow_P v' & \text{step} \cdot \bar{\omega}}
\]

\[
F(x < 0)
\]

\[
x := 1; \\
\text{increment}(x); \\
x := -1;
\]
Language

$$e[v/x] \uparrow_P \bot \& \pi \quad \frac{(\lambda x.e) v \uparrow_P \bot \& \text{step} \cdot \pi}{\text{snt-App}}$$
Type-and-effect

\( \Phi^{\text{fin}} :: \subseteq \Sigma^* \)

\( \Phi^{\text{inf}} :: \subseteq \Sigma^\omega \)

\( \Phi ::= (\Phi^{\text{fin}}, \Phi^{\text{inf}}) \)

\( B ::= \text{int} \mid \text{bool} \mid \text{unit} \)

\( \tau, \sigma ::= \{ u : B \mid \theta \} \mid x : \sigma \xrightarrow{\Phi} \tau \)

Trace set operations

\[
(\Phi_1^{\text{fin}}, \Phi_1^{\text{inf}}) \cup (\Phi_2^{\text{fin}}, \Phi_2^{\text{inf}}) = (\Phi_1^{\text{fin}} \cup \Phi_2^{\text{fin}}, \Phi_1^{\text{inf}} \cup \Phi_2^{\text{inf}})
\]

\[
(\Phi_1^{\text{fin}}, \Phi_1^{\text{inf}}) \cap (\Phi_2^{\text{fin}}, \Phi_2^{\text{inf}}) = (\Phi_1^{\text{fin}} \cap \Phi_2^{\text{fin}}, \Phi_1^{\text{inf}} \cap \Phi_2^{\text{inf}})
\]

\[
(\Phi_1^{\text{fin}}, \Phi_1^{\text{inf}}) \cdot (\Phi_2^{\text{fin}}, \Phi_2^{\text{inf}}) = (\Phi_1^{\text{fin}} \cdot \Phi_2^{\text{fin}}, \Phi_1^{\text{inf}} \cup (\Phi_1^{\text{fin}} \cdot \Phi_2^{\text{inf}}))
\]
**Soundness**

Semantics of Type and Effect

\[ [\Theta \vdash \tau & \Phi]_P \]

Oracle Conditions

For any \( e \) in \( P \)

\[ \Theta \triangleright P e : \tau & \Phi \]

Soundness

*Suppose*

\[ \Delta \vdash * \]

\[ \text{dom}(\Theta) = \text{fv}(e) \]

\[ \Delta, \Theta \vdash e : \tau & \Phi \]

*Then*

\[ e \in [\Theta \vdash \tau & \Phi] \]

simply typed

type & effect
Examples

- Modular reasoning
- Nesting G within F, nesting U within G
- Oracles for Termination, Non-termination
- Dependent typing (e.g. bar x returns non-positive)
- Type system fix points
Thank you!