Type-Checking Program Generators Using the Record Calculus

Barış Aktemur

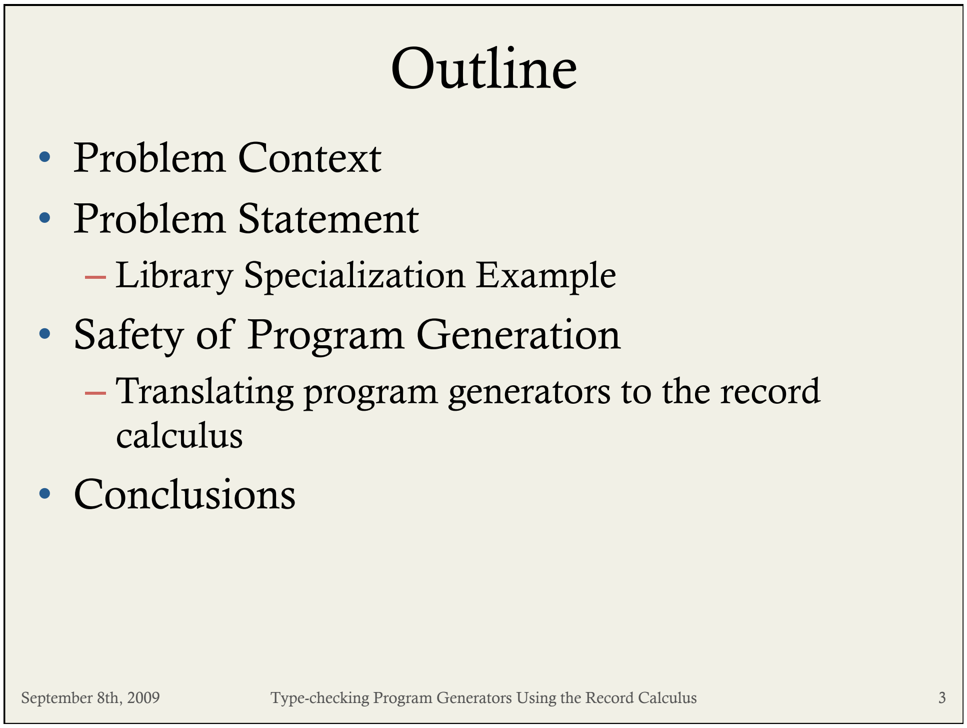
Özyeğin University, Turkey

`baris.aktemur@ozyegin.edu.tr`

About the Speaker

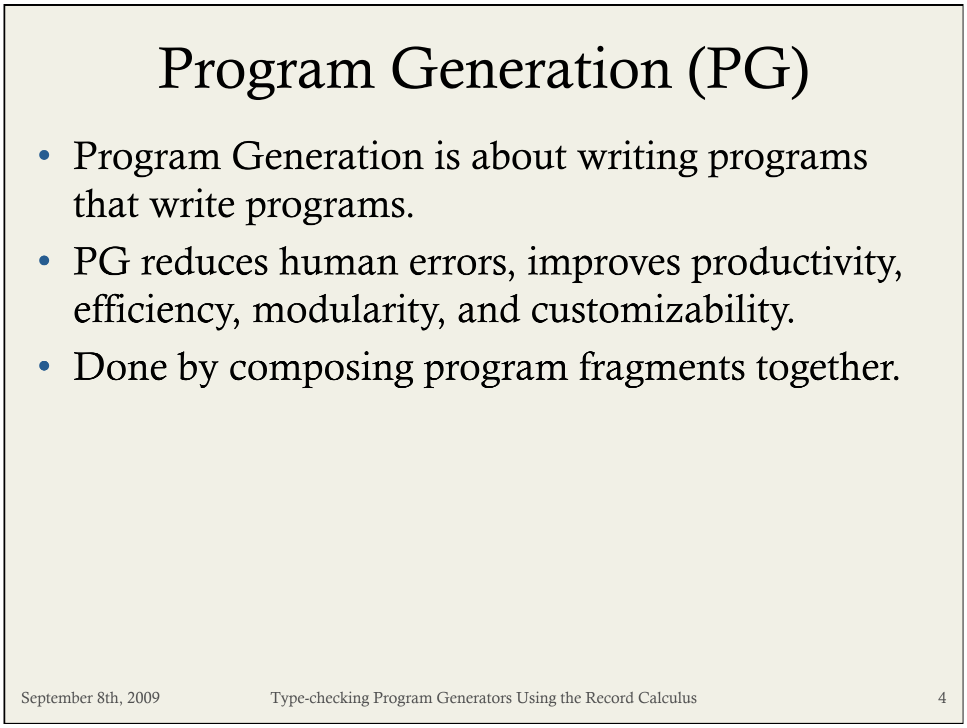
- B.S.: Bilkent University, Turkey, 2003
- M.S.: University of Illinois at Urbana-Champaign, USA, 2005
- Ph.D.: University of Illinois, USA, 2009
  - Advisor: Sam Kamin
- Now: Assistant prof. at Özyeğin University, Turkey

- Interests: Runtime program generation, programming language design and analysis, software engineering
- Today's talk: part of my dissertation



Outline

- Problem Context
- Problem Statement
  - Library Specialization Example
- Safety of Program Generation
  - Translating program generators to the record calculus
- Conclusions



Program Generation (PG)

- Program Generation is about writing programs that write programs.
- PG reduces human errors, improves productivity, efficiency, modularity, and customizability.
- Done by composing program fragments together.

High Degree of Generality

- Arbitrary code fragments are combined to construct a program
- Fragments are first-class citizens
- Turing-complete meta-language
- Flexibility in
  - what can be defined as fragments
  - how the fragments can be combined
- Uses a quotation/anti-quotation syntax
  - Quotation: ⟨ … ⟩ to define fragments
  - Antiquotation: ⟨… \`(…)… ⟩ to define holes

```
int power(int x, int n) {
    int c = 1;
    for(int i=0; i<n; i++) {
        c = x * c;
    }
    return c;
}
```

```
Code genBody(int n) {
    Code c = ⟨ 1 ⟩;
    for(int i=0; i<n; i++) {
        c = ⟨ x * `(c) ⟩;
    }
    return c;
}
```

⟨ x*x*x*x*x*1 ⟩ for n=5

```
int power(int x) {
    return  x*x*x*x*x*1;
}
```

```
Code genPower(int n) {
    return
        ⟨ int power(int x) {
            return `( genBody(n) );
          } ⟩;
}
```

Type-safety of the Generated Program

- Problem:
  - How can we guarantee *statically* that a generator will produce type-safe code?
- Expectations from the type system motivated by the *library specialization* problem:
  - Libraries come with advanced features
    - Large memory footprint
  - Produce a lightweight version of a library by excluding unused features

---

Adapted from C5
[Kokholm and Sestoft]

```
class LinkedList implements List {
  Node first,last; // a doubly linked list
  int size;
  int counter = 0;
  void reverse() {
    counter++;
    Node a = first.next, b = last.prev;
    for(int i=0; i<size/2; i++) {
      Object swap = a.item;
      a.item = b.item; b.item = swap;
      a = a.next; b = b.prev;
    }
  }
  void add(Object item) {
    counter++;
    Node a = new Node(item);
    ...
  }
}
```

```
Code genLL(Code field, Code inc) {
   return ⟨ class LinkedList implements List {
                Node first,last; // a doubly linked list
                int size;
               `(field)
                void reverse() {
                  `(inc)
                  Node a = first.next, b = last.prev;
                  for(int i=0; i<size/2; i++) {
                     Object swap = a.item;
                     a.item = b.item; b.item = swap;
                     a = a.next; b = b.prev;
                  }
                }
                void add(Object item) {
                  `(inc)
                  Node a = new Node(item);
                  ...
                }
             } ⟩;
}
```

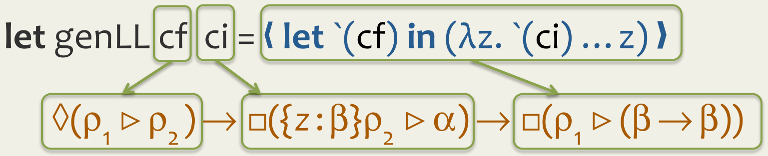
```
genLL(⟨ int counter = 0; ⟩,
      ⟨ counter++; ⟩ )          ✓

genLL(⟨ ⟩, ⟨ ⟩ )                ✓

genLL(⟨ ⟩, ⟨ counter++; ⟩ )     ✗
```

More details in
[Aktemur and Kamin SAC09]

λ_{open}^{poly} *[Kim-Yi-Calcagno POPL06]*

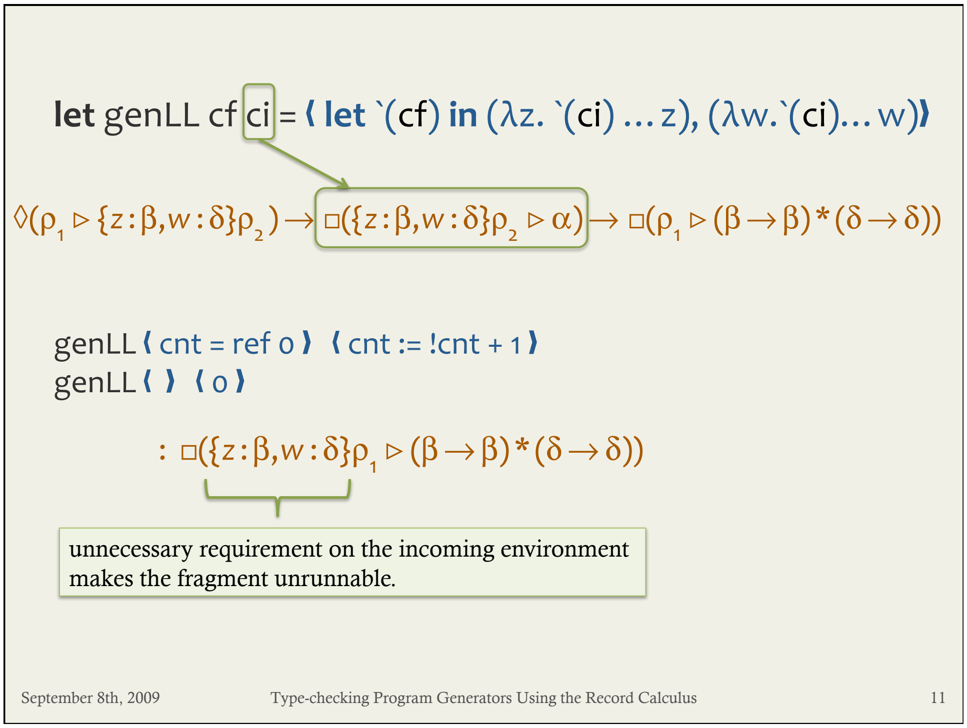


let genLL $\boxed{\text{cf}}$ $\boxed{\text{ci}}$ = $\boxed{\langle\, \textbf{let}\ \grave{}(\text{cf})\ \textbf{in}\ (\lambda z.\ \grave{}(\text{ci}) \ldots z)\,\rangle}$

$$\boxed{\Diamond(\rho_1 \rhd \rho_2)} \rightarrow \boxed{\Box(\{z:\beta\}\rho_2 \rhd \alpha)} \rightarrow \boxed{\Box(\rho_1 \rhd (\beta \rightarrow \beta))}$$

genLL ⟨ cnt = ref 0 ⟩ ⟨ cnt := !cnt + 1 ⟩
genLL ⟨ ⟩ ⟨ 0 ⟩ $: \Box(\rho_1 \rhd (\beta \rightarrow \beta))$

- Fragment type $\Box(\Gamma \rhd \beta)$
  - "The fragment has type β if evaluated in the environment Γ."
- Need declaration type $\Diamond(\Gamma_1 \rhd \Gamma_2)$
  - "The declaration yields in environment Γ_2 if evaluated in environment Γ_1.



let genLL cf $\boxed{\text{ci}}$ = ⟨ **let** \`(cf) **in** (λz. \`(ci) … z), (λw.\`(ci)… w)⟩

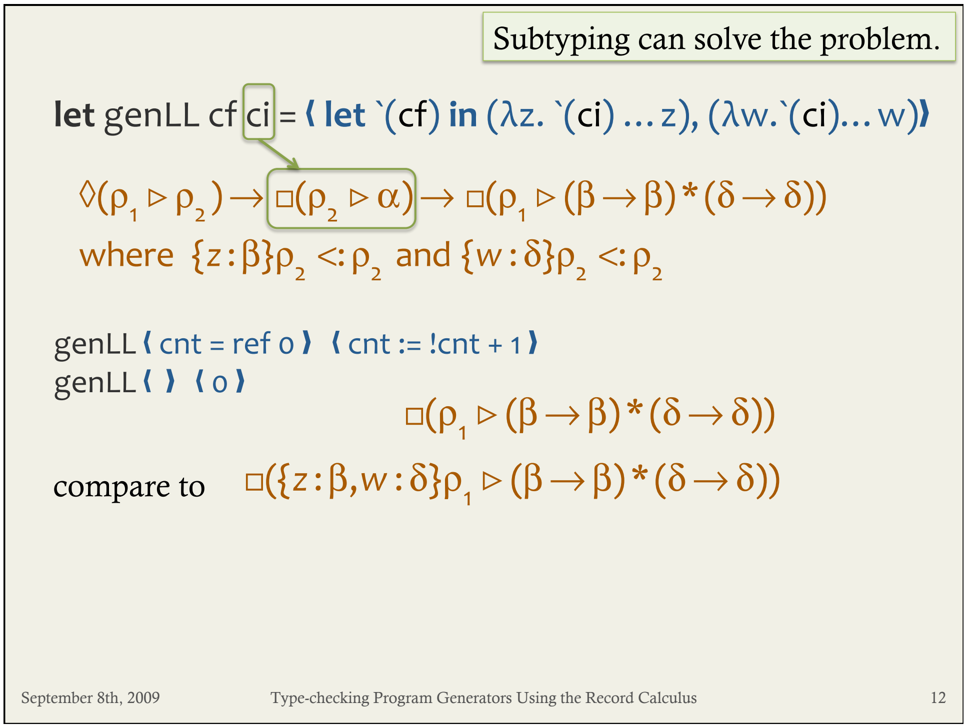
$$\Diamond(\rho_1 \triangleright \{z:\beta,w:\delta\}\rho_2) \to \boxed{\square(\{z:\beta,w:\delta\}\rho_2 \triangleright \alpha)} \to \square(\rho_1 \triangleright (\beta \to \beta) * (\delta \to \delta))$$

genLL ⟨ cnt = ref 0 ⟩ ⟨ cnt := !cnt + 1 ⟩
genLL ⟨ ⟩ ⟨ 0 ⟩

$$: \square(\underbrace{\{z:\beta,w:\delta\}}\rho_1 \triangleright (\beta \to \beta) * (\delta \to \delta))$$

unnecessary requirement on the incoming environment makes the fragment unrunnable.

September 8th, 2009 Type-checking Program Generators Using the Record Calculus 11



Subtyping can solve the problem.

let genLL cf $\boxed{\text{ci}}$ = ⟨ **let** \`(cf) **in** (λz. \`(ci) … z), (λw.\`(ci)… w)⟩

$$\Diamond(\rho_1 \triangleright \rho_2) \to \boxed{\square(\rho_2 \triangleright \alpha)} \to \square(\rho_1 \triangleright (\beta \to \beta) * (\delta \to \delta))$$

$$\text{where } \{z:\beta\}\rho_2 <: \rho_2 \text{ and } \{w:\delta\}\rho_2 <: \rho_2$$

genLL ⟨ cnt = ref 0 ⟩ ⟨ cnt := !cnt + 1 ⟩
genLL ⟨ ⟩ ⟨ 0 ⟩

$$\square(\rho_1 \triangleright (\beta \to \beta) * (\delta \to \delta))$$

compare to $\square(\{z:\beta,w:\delta\}\rho_1 \triangleright (\beta \to \beta) * (\delta \to \delta))$

September 8th, 2009 Type-checking Program Generators Using the Record Calculus 12

Type-checking Program Generators

- λ^{poly}_{open} does not completely satisfy the library specialization problem.
- Two requirements
  - Pluggable declarations
  - Subtyping

will come back to these

Code Fragments vs. Record Calculus

⟨ 2+3 ⟩	λr. 2+3
⟨ x+3 ⟩	λr. r•x+3
⟨ `(c)+3 ⟩	λr. c(r)+3
⟨ λx.x+3 ⟩	λr. λy. **let** r = r with {x=y} **in** r•x+3
run ⟨ 2+3 ⟩	(λr. 2+3) { }

Transformation

$[\![c]\!]^n = c$

stage = number of surrounding quotations

$[\![x]\!]^n = r_n \cdot x$

$[\![\lambda x.e]\!]^n = \lambda y.\textbf{let } r_n = r_n \textbf{ with } \{x = y\} \textbf{ in } [\![e]\!]^n$

$[\![e_1 e_2]\!]^n = [\![e_1]\!]^n [\![e_2]\!]^n$

$[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]^n = \textbf{let } r_n = r_n \textbf{ with } \{x = [\![e_1]\!]^n\} \textbf{ in } [\![e_2]\!]^n$

$[\![\langle e \rangle]\!]^n = \lambda r_{n+1}.[\![e]\!]^{n+1}$

$[\![\text{`}(e)]\!]^{n+1} = [\![e]\!]^n r_{n+1}$

$[\![run(e)]\!]^n = [\![e]\!]^n \{\}$

Equivalence of Staged vs. Record Semantics

A major theorem

Staged calculus $\boxed{\tau}\ e_1 \xrightarrow{n} e_2\ \boxed{\tau}$

$$\Downarrow$$

Record calculus $\boxed{\tau}\ [\![e_1]\!]^n \xrightarrow[\beta]{}{}^{*} [\![e_2]\!]^n\ \boxed{\tau}$

- Can we use a record type system to type-check a staged expression?
  - "Expression e is type-safe iff $[\![e]\!]^n$ is type-safe."
  - Soundness? (i.e. Preservation and Progress)
  - Preservation property comes for free.

Soundness of the Type System

- Progress: "If e_1 is typable, it is either a value or there exists e_2 such that $e_1 \xrightarrow{n} e_2$."
  - Has to be proven explicitly.
- Need to put restrictions on record type system
  - λx.⟨ 42 ⟩ x => λx.(λr. 42)x
  - Distinguish record variables from other variables

record variables

other variables

$$\Gamma \in \mathit{RecordType}$$
$$A \in \mathit{LegType} ::= \alpha \mid \iota \mid T \to A$$
$$T \in \mathit{Type} ::= A \mid \Gamma$$

Record Type System

- Record type system is sound with respect to program generation semantics.
- We can use the type inference algorithm to infer a type.
- So, how powerful is it?

$$\boxed{\Delta_0 \ldots \Delta_n \vdash_S e : A} \Leftrightarrow \boxed{[\![\Delta_0 \ldots \Delta_n]\!] \vdash_R [\![e]\!]^n : [\![A]\!]}$$

[Kim-Yi-Calcagno POPL06] *Record type system*

Type-checking Program Generators

- Translation converts program generators to record calculus expressions.
- Record calculus provides a sound and powerful type system to type-check program generators.
- How about the two requirements motivated by the library specialization problem?
  - Subtyping
  - Pluggable declarations

Subtyping

- Record subtyping
  - Pottier defines a constraint system combining subtyping and records
  - Can instantiate Odersky, Sulzmann, Wehr's HM(X)

$G = \lambda c.\ \langle\ \textbf{let}\ x{=}1\ \textbf{in}\ \grave{}(c)\ ,\ \textbf{let}\ y{=}1\ \textbf{in}\ \grave{}(c)\ \rangle$

$$\Box(\{x:\text{int},y:\text{int}\}\rho \rhd \alpha) \to \Box(\{x:\text{int},y:\text{int}\}\rho \rhd (\alpha * \alpha))$$

$$\Box(\{x:\theta_1,y:\theta_2\}\rho \rhd \alpha) \to \Box(\{x:\theta_1,y:\theta_2\}\rho \rhd (\alpha * \alpha))$$

where $\text{int} <: \theta_1$ and $\text{int} <: \boxed{\theta_2}$

Absence or concrete type

$G\langle\, o\, \rangle \longrightarrow \langle\ \textbf{let}\ x{=}1\ \textbf{in}\ o\ ,\ \textbf{let}\ y{=}1\ \textbf{in}\ o\ \rangle$

Not Runnable

$\Box(\{x:\text{int},y:\text{int}\}\rho \rhd (\text{int} * \text{int}))$

Runnable

$\Box(\{x:Abs,y:Abs\}\rho \rhd (\text{int} * \text{int}))$

because $\text{int} <: Abs$ and $\text{int} <: Abs$

Subtyping

- Record type system with subtyping
  - still sound w.r.t. program generation semantics
  - subsumes plain record type system
- Translation preserves contra/co-variance properties

$$\frac{\Gamma_2 <: \Gamma_1 \qquad A_1 <: A_2}{\Box(\Gamma_1 \rhd A_1) <: \Box(\Gamma_2 \rhd A_2)} \qquad\qquad \frac{\Gamma_2 <: \Gamma_1 \qquad A_1 <: A_2}{\Gamma_1 \to A_1 <: \Gamma_2 \to A_2}$$

Pluggable Declarations

```
let genLL cf ci = ⟨ let `(cf) in (λz. `(ci) ... z) ⟩

genLL ⟨ cnt = ref 0 ⟩ ⟨ cnt := !cnt + 1 ⟩
genLL ⟨ ⟩ ⟨ ⟩
```

- Extend the λ^{poly}_{open} syntax, semantics and the type system
- Soundness is preserved

Pluggable Declarations

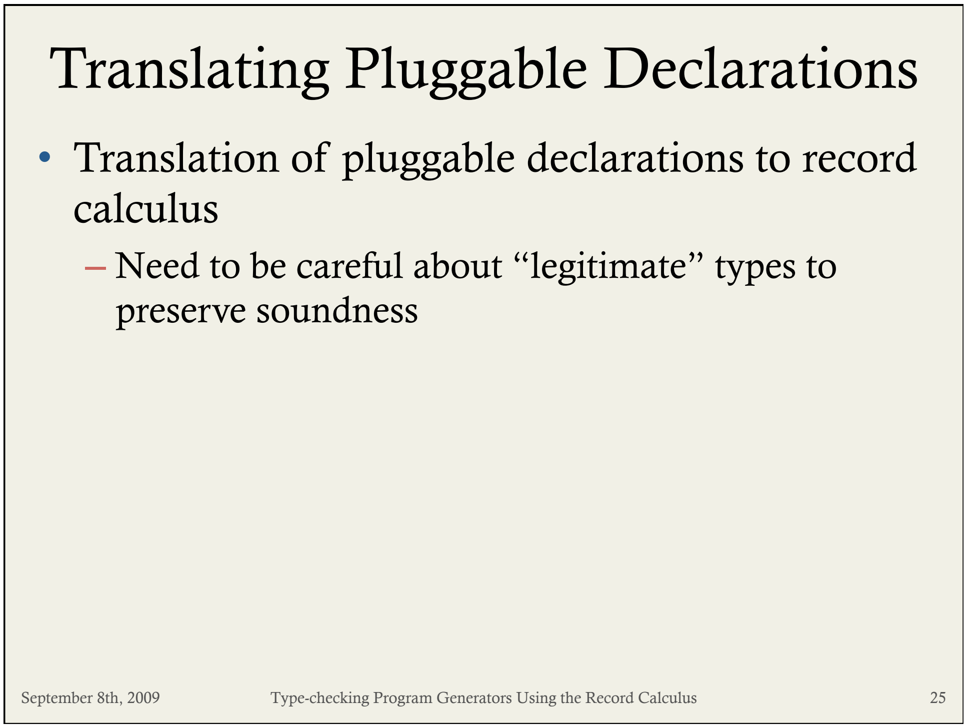
- Pluggable declarations are syntactic sugar.[♮]
- Define a desugaring function δ:

⟨ x = e ⟩ => λc.⟨ **let** x = e **in** `(c) ⟩
let `($e_1$) **in** $e_2$ => `(e_1⟨e_2⟩)

$$e_1 \xrightarrow{n} e_2 \Rightarrow \delta(e_1) \xrightarrow{n}{}^{*} \delta(e_2)$$

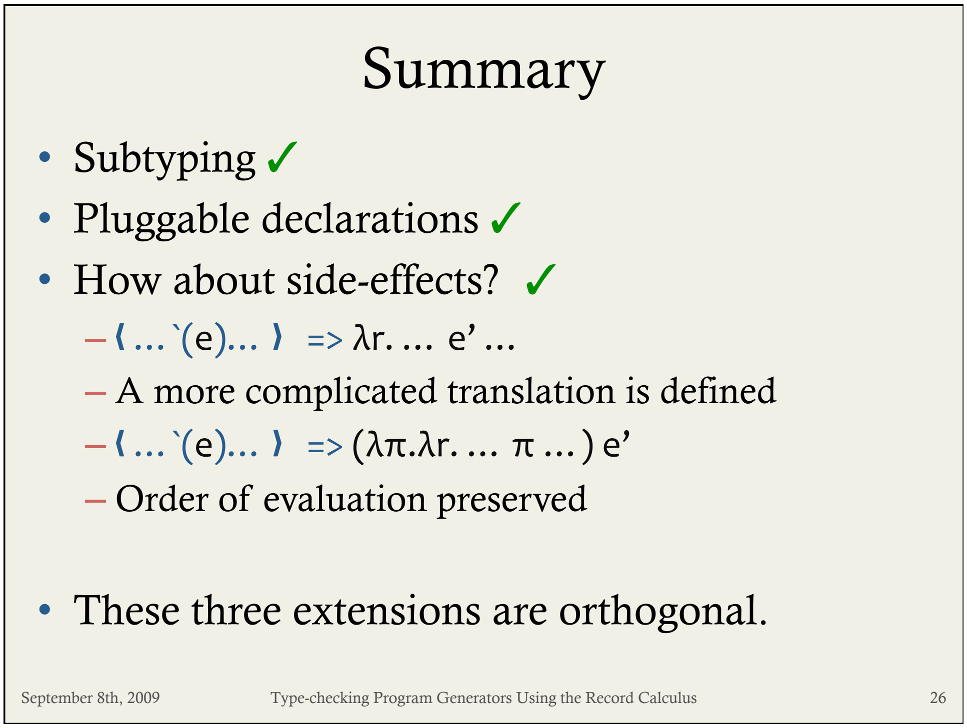
$$\Delta_0 \ldots \Delta_n \vdash e : A \Rightarrow \delta(\Delta_0) \ldots \delta(\Delta_n) \vdash \delta(e) : \delta(A)$$

[♮] *Thanks to Prof. Chung-chieh Shan*



Translating Pluggable Declarations

- Translation of pluggable declarations to record calculus
  - Need to be careful about “legitimate” types to preserve soundness



Summary

- Subtyping ✓
- Pluggable declarations ✓
- How about side-effects? ✓
  - ⟨ … `(e)… ⟩ => λr. … e’ …
  - A more complicated translation is defined
  - ⟨ … `(e)… ⟩ => (λπ.λr. … π …) e’
  - Order of evaluation preserved

- These three extensions are orthogonal.

Related Work

- [Kameyama-Kiselyov-Shan PEPM08]
  - Not multi-stage
  - Driven by type annotations
    - Higher-rank polymorphism
  - No type inference
  - Conjecture stated for operational semantics relation
- [Chen-Xi ICFP03]
  - Translation to first-order abstract syntax
    - Can convert back to staged language
  - Program variables converted to de Bruijn indices
    - Bindings vanishing or occurring “unexpectedly”

Related Work

- [Kim-Yi-Calcagno POPL06]
  - Starting point for our work (added recursion)
- [Nanevski 02]
  - Free variables of a fragment become part of its type
  - The list of free variables in a type can be loosened
    - Subtyping
    - Not sufficient for library specialization because no type information is kept – only names

Conclusions

- Safety of program generation
  - Record calculus provides a sound and powerful type system for program generation
  - Existing knowledge in the record calculus research is very useful
    - E.g. subtyping
  - Type system is extensible with pluggable declarations and side-effecting expressions
  - Library specialization problem

Future Work

- Staged typing
  - A staged type system with subtyping that does not depend on record calculus
  - Extending the type system to a procedural/object-oriented language
    - Side-effecting expressions are already handled
    - Inheritance may pose difficulty
- Analysis of program generators
- Optimization of generators by translation to record calculus

Extra Slides

Translating Pluggable Declarations

- First attempt $[\![\langle x = e \rangle]\!]^n = \lambda r_n . r_n \text{ with } \{x = [\![e]\!]^{n+1}\}$
  - $\underbrace{\langle 5 \rangle\, `(\langle x = 2 \rangle)}_{\text{type-incorrect}} => \underbrace{(\lambda r_2 . 5)\ ((\lambda r_2 . r_2 \text{ with } \{x{=}2\}) r1)}_{\text{type-correct}}$
- Second attempt $[\![\langle x = e \rangle]\!]^n = [\![\lambda c . \langle \text{let } x = e \text{ in } `(c) \rangle]\!]^n$
  - $\langle x = 1 \rangle \langle 5 \rangle$ passes the type checker.
- Solution:

$$[\![\langle x = e \rangle]\!]^n = \lambda \kappa . [\![\lambda c . \langle \text{let } x = e \text{ in } `(c) \rangle]\!]^n$$

$$[\![\text{let } `(e_1) \text{ in } e_2 ]\!]^n = [\![ `(e_1\ \kappa\ \langle e_2 \rangle)]\!]^n$$

Complexity

- [Pottier 98]: Accumulation of constraints by type inference is at best linear in program size; at worst exponential because let-constructs duplicate them.
- [Su-Aiken-Niehren-Priesnitz-Treinen 02]: Constraint-solving is decidable; constraint-entailment is undecidable.
- [Frey 97]: Constraint-solving in PSPACE.
- [Palsberg-Zhao 04]: Type inference algorithm for record concatenation, subtyping, and recursive types. Based on Abadi-Cardelli calculus. Type inference problem is proved to be NP-complete.

Cannot Type

- Because of rank-1 polymorphism, cannot type

$$\lambda y.(y\ 1,\ y\ \text{'}a\text{'})$$

- Polymorphic types are not preserved after antiquotation/quotation

$$\langle \text{let } y = \lambda x.x \text{ in } \grave{}(\langle y\ 1,\ y\ \text{'}a\text{'}\rangle)\rangle$$