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
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

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

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

Code genPower(int n) {
    Code c = ( 1 );
    for(int i=0; i<n; i++) {
        c = ( x * (c) );
    }
    return c;
}

Code genBody(int n) {
    ( x*x*x*x*x*1 ) for n=5
}

Code genBody(n) {
    return `( genBody(n) );
}
```

September 8th, 2009  Type-checking Program Generators Using the Record Calculus
Type-safety of the Generated Program

• Problem:
  – How can we guarantee \textit{statically} that a generator will produce type-safe code?

• Expectations from the type system motivated by the \textit{library specialization} problem:
  – Libraries come with advanced features
    • Large memory footprint
  – Produce a lightweight version of a library by excluding unused features

```java
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);
        ...
    }
}
```

Adapted from C5 [Kokholm and Sestoft]
Code genLL(Code field, Code inc) {
    return {
        class LinkedList implements List {
            Node first, last; // a doubly linked list
            int size;
            (field)
            void reverse() {
                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);
                ...}
            }
        }
    }
}

More details in [Aktemur and Kamin SAC09]

– Fragment type □(Γ ⊢ β)
  • “The fragment has type β if evaluated in the environment Γ.”

– Need declaration type ∅(Γ₁ ⊢ Γ₂)
  • “The declaration yields in environment Γ₂ if evaluated in environment Γ₁.”

[Kim-Yi-Calcagno POPL06]
let genLL cf ci = (let (cf) in (λz. (cf) ... z), (λw. (cf)... w))

\( (\rho_1 \triangleright \{z: \beta, w: \delta\} \triangleright \alpha) \rightarrow \Box (\rho_1 \triangleright (\beta \rightarrow \beta) \ast (\delta \rightarrow \delta)) \)

\( \triangleright (\rho_1 \triangleright \rho_2) \rightarrow \Box (\rho_2 \triangleright \alpha) \rightarrow \Box (\rho_1 \triangleright (\beta \rightarrow \beta) \ast (\delta \rightarrow \delta)) \)

where \( \{z: \beta\} \triangleright \rho_2 < \triangleright \rho_2 \) and \( \{w: \delta\} \triangleright \rho_2 < \triangleright \rho_2 \)

\( \triangleright (\rho_1 \triangleright (\beta \rightarrow \beta) \ast (\delta \rightarrow \delta)) \)

\( \triangleright ((\{z: \beta, w: \delta\} \triangleright (\beta \rightarrow \beta) \ast (\delta \rightarrow \delta)) \)

unnecessary requirement on the incoming environment makes the fragment unrunnable.

Subtyping can solve the problem.

let genLL cf ci = (let (cf) in (λz. (cf) ... z), (λw. (cf)... w))

\( \triangleright (\rho_1 \triangleright \rho_2) \rightarrow \Box (\rho_2 \triangleright \alpha) \rightarrow \Box (\rho_1 \triangleright (\beta \rightarrow \beta) \ast (\delta \rightarrow \delta)) \)

\( \triangleright (\rho_1 \triangleright \rho_2) \rightarrow \Box (\rho_2 \triangleright \alpha) \rightarrow \Box (\rho_1 \triangleright (\beta \rightarrow \beta) \ast (\delta \rightarrow \delta)) \)

\( \triangleright (\rho_1 \triangleright (\beta \rightarrow \beta) \ast (\delta \rightarrow \delta)) \)

\( \triangleright ((\{z: \beta, w: \delta\} \triangleright (\beta \rightarrow \beta) \ast (\delta \rightarrow \delta)) \)
Type-checking Program Generators

- $\lambda_{\text{poly}}$ does not completely satisfy the library specialization problem.

- Two requirements
  - Pluggable declarations
  - Subtyping

will come back to these

Code Fragments vs. Record Calculus

\[
\{ 2+3 \} \rightarrow \lambda r. 2+3 \\
\{ x+3 \} \rightarrow \lambda r. r\cdot x+3 \\
\{ (c)+3 \} \rightarrow \lambda r. c(r)+3 \\
\{ \lambda x.x+3 \} \rightarrow \lambda r. \lambda y. \text{let } r = r \text{ with } \{ x=y \} \\
\phantom{\{ \lambda x.x+3 \} } \quad \text{in } r \cdot x+3 \\
\text{run } \{ 2+3 \} \rightarrow (\lambda r. 2+3) \{ \}
\]
Transformation

\[ [c]^n = c \]
\[ [x]^n = r_n \cdot x \]
\[ [\lambda x.e]^n = \lambda y.\text{let } r_n = r_n \text{ with } \{ x = y \} \text{ in } [e]^n \]
\[ [e_1, e_2]^n = [e_1]^n [e_2]^n \]
\[ [\text{let } x = e_1 \text{ in } e_2]^n = \text{let } r_n = r_n \text{ with } \{ x = [e_1]^n \} \text{ in } [e_2]^n \]
\[ [\langle e \rangle]^n = \lambda r_{n+1}. [e]^{n+1} \]
\[ [\cdot (e)]^{n+1} = [e]^n r_{n+1} \]
\[ [\text{run}(e)]^n = [e]^n \{ \} \]

Equivalence of Staged vs. Record Semantics

- 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 $e_1 \stackrel{n}{\rightarrow} e_2$.”
  – Has to be proven explicitly.

• Need to put restrictions on record type system
  – $\lambda x.\{ 42 \} x \Rightarrow \lambda x.(\lambda r. 42)x$
  – Distinguish record variables from other variables

<table>
<thead>
<tr>
<th>record variables</th>
<th>other variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \in \text{RecordType}$</td>
<td></td>
</tr>
<tr>
<td>$A \in \text{LegType} ::= \alpha \mid t \mid T \rightarrow A$</td>
<td></td>
</tr>
<tr>
<td>$T \in \text{Type} ::= A \mid \Gamma$</td>
<td></td>
</tr>
</tbody>
</table>

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?

$$\Delta_0 \ldots \Delta_n \vdash_S e : A \iff \llbracket \Delta_0 \ldots \Delta_n \rrbracket \vdash_R \llbracket e \rrbracket^n : \llbracket A \rrbracket$$

[Kim-Yi-Calcagno POPL06]
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)
\[ G = \lambda c. \{ \text{let } x = 1 \text{ in } (c), \text{ let } y = 1 \text{ in } (c) \} \]

\[ \square(\{x : \text{int}, y : \text{int}\} \rho \triangleright \alpha ) \rightarrow \square(\{x : \text{int}, y : \text{int}\} \rho \triangleright (\alpha \ast \alpha )) \]

\[ \square(\{x : \text{int}, y : \text{int}\} \rho \triangleright \alpha ) \rightarrow \square(\{x : \text{int}, y : \text{int}\} \rho \triangleright (\alpha \ast \alpha )) \]

where int \(<:\theta_1\) and int \(<:\theta_2\)

Absence or concrete type

\[ G (\circ) \longrightarrow \{ \text{let } x = 1 \text{ in } o, \text{ let } y = 1 \text{ in } o \} \]

Not Runnable

\[ \square(\{x : \text{int}, y : \text{int}\} \rho \triangleright (\text{int} \ast \text{int})) \]

Runnable

because int \(<:\text{Abs}\) and int \(<:\text{Abs}\)

---

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

\[
\begin{align*}
\Gamma_2 &<:\Gamma_1 & A_1 &<:\ A_2 \\
\square(\Gamma_1 \triangleright A_1) &<:\square(\Gamma_2 \triangleright A_2) \\
\Gamma_1 \rightarrow A_1 &<:\Gamma_2 \rightarrow A_2 \\
\end{align*}
\]
Pluggable Declarations

Let $\text{genLL}\ cf\ ci = (\text{let}\ (cf)\ in\ (\lambda z. (ci) \ldots z))$

$\text{genLL}\ (\text{cnt} = \text{ref}\ 0)\ (\text{cnt} := !\text{cnt} + 1)$

$\text{genLL}\ (\ )\ (\ )$

- Extend the $\lambda_{\text{open}}^{\text{poly}}$ syntax, semantics and the type system
- Soundness is preserved

---

Pluggable Declarations

- Pluggable declarations are syntactic sugar.\(^\dagger\)
- Define a desugaring function $\delta$:
  
  \[
  \{ x = e \} \Rightarrow \lambda c.\{ \text{let}\ x = e\ \text{in}\ (c) \}
  \]
  
  \[
  \text{let}\ (e,)\ \text{in}\ e_2 \Rightarrow (e, (e_2))
  \]

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

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

\(^\dagger\ Thanks\ to\ Prof.\ Chung-chieh\ Shan\)
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? ✓
  - \{ ... `\(e\) ... \} => \(\lambda r. ... \ e' ... \)
  - A more complicated translation is defined
  - \{ ... `\(e\) ... \} => (\(\lambda \pi.\lambda r. ... \ \pi ... \) \(e'\))
  - 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”

- [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
Translating Pluggable Declarations

• First attempt  \([\langle x = e \rangle]^n = \lambda r_n.r_n\) with \(\{ x = [e]^{n+1} \} \)
  \(- \langle 5 \rangle \langle l x = 2 \rangle \Rightarrow (\lambda r_2.5)((\lambda r_2.r_2\) with \(\{x=2\})\rangle r_1)\)

• 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:
  \([\langle x = e \rangle]^n = \lambda c.[\lambda c.\langle\text{let } x = e \text{ in } '(c)'\rangle]^n\)
  \([\text{let } '(e_1)\text{ in } e_2]\]^n = [\langle e_1 \times (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

\[ \lambda y. (y \ 1, y \ 'a') \]

• Polymorphic types are not preserved after antiquotation/quotation

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