Typing program generators using the record calculus

Sam Kamin

Tankut Barış Aktemur

WG 2.11

April 15th, 2009

Outline

- Motivation – The library specialization problem and the need for subtyping
- Translating staged ML to record calculus
- Typing staged ML via record calculus
- Subtyping
- “Pluggable declarations”
- Handling side effects
- Related work

Partial evaluation vs. program construction

- Partial evaluation
  - Generate program by staging normal program
  - Erasure property
  - No open terms
- Program construction
  - Build program from fragments
  - Allow open terms
  - No erasure property
- *This work takes a program construction approach to program generation.*

Outline

- **Motivation – The library specialization problem and the need for subtyping**
- Translating staged ML to record calculus
- Typing staged ML via record calculus
- Subtyping
- “Pluggable declarations”
- Handling side effects
- Related work

Library specialization

- Problem: Libraries too general – users pay for features they don't use.
- ***How can one provide a* set *of classes representing all subsets of a library class's features?***
- Problem was described by Peter Sestoft in WG2.11 meeting in Portland in 2006, in context of C5 collection library.
- A comparison of methods, including program generation, is given in Aktemur, Kamin, "*Writing Customizable Libraries - A comparative study* ," Symp. on Applied Computing, 2009.

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]

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

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

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

genLL ⟨ ⟩ ⟨ 0 ⟩

$: \Box(\rho_1 \triangleright (\beta \rightarrow \beta))$

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

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

$$\Diamond(\rho_1 \triangleright \{z:\beta, w:\delta\}\rho_2) \rightarrow \Box(\{z:\beta, w:\delta\}\rho_2 \triangleright \alpha) \rightarrow \Box(\rho_1 \triangleright (\beta \rightarrow \beta) * (\delta \rightarrow \delta))$$

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

$$: \Box(\{z:\beta, w:\delta\}\rho_1 \triangleright (\beta \rightarrow \beta) * (\delta \rightarrow \delta))$$

unnecessary requirement on the incoming environment makes the fragment unrunnable.

Subtyping can solve the problem.

let genLL cf ci = ⟨ **let** `(cf) **in** (λz. `(ci) … z), (λw.`(ci)… w)⟩

$$\Diamond(\rho_1 \triangleright \rho_2) \rightarrow \Box(\rho_2 \triangleright \alpha) \rightarrow \Box(\rho_1 \triangleright (\beta \rightarrow \beta) * (\delta \rightarrow \delta))$$

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

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

$$\Box(\rho_1 \triangleright (\beta \rightarrow \beta) * (\delta \rightarrow \delta))$$

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

Type-checking Program Generators

- λ_{open}^{poly} cannot completely satisfy the library specialization problem.
- Two requirements
  - Pluggable declarations
  - Subtyping

  } will come back to these

Outline

- Motivation – The library specialization problem and the need for subtyping
- **Translating staged ML to record calculus**
- Typing staged ML via record calculus
- Subtyping
- “Pluggable declarations”
- Handling side effects
- Related work

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

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

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

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

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

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

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

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

Examples

$$[\![\lambda c.\langle \text{let } x=5 \text{ in } `(c) \rangle]\!] = \lambda c.\text{let } r_0 = r_0 \text{ with } \{\overline{c}=c\} \text{in}$$

$$(\lambda r.\text{let } r_1 = r_1 \text{ with } \{\overline{x}=5\}$$

$$\text{in } r_0.\overline{c}(r_1))$$

$$[\![\lambda y.\langle y + `(y) \rangle]\!]^0 = \lambda y.\text{let } r_0 = r_0 \text{ with } \{\overline{y}=y\} \text{ in}$$

$$(\lambda r_1.\ r_1.\overline{y} + (r_0.\overline{y})(r_1))$$

Outline

- Motivation – The library specialization problem and the need for subtyping
- Translating staged ML to record calculus
- **Typing staged ML via record calculus**
- Subtyping
- “Pluggable declarations”
- Handling side effects
- Related work

Equivalence of Staged vs. Record Semantics

A major theorem

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

$$\Downarrow$$

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

- 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 <u>for free</u>.

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
  - $\lambda x.\langle\, 42 \,\rangle\, x \Rightarrow \lambda x.(\lambda r.\ 42)x$
  - Distinguish record variables from other variables

record variables

other variables

$$\Gamma \in RecordType$$

$$A \in \ LegType ::= \alpha \mid \iota \mid T \to A$$

$$T \in Type ::= A \mid \Gamma$$

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?

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

[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

Outline

- Motivation – The library specialization problem and the need for subtyping
- Translating staged ML to record calculus
- Typing staged ML via record calculus
- **Subtyping**
- "Pluggable declarations"
- Handling side effects
- Related work

Subtyping

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

$G = \lambda c.\ \langle\ \mathbf{let}\ x{=}1\ \mathbf{in}\ \text{\`{}}(c)\ ,\ \mathbf{let}\ y{=}1\ \mathbf{in}\ \text{\`{}}(c)\ \rangle$

$\square(\{x:\text{int}, y:\text{int}\}\rho \triangleright \alpha) \rightarrow \square(\{x:\text{int}, y:\text{int}\}\rho \triangleright (\alpha * \alpha))$

$\square(\{x:\theta_1, y:\theta_2\}\rho \triangleright \alpha) \rightarrow \square(\{x:\theta_1, y:\theta_2\}\rho \triangleright (\alpha * \alpha))$

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

Absence or concrete type

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

Not Runnable

$\square(\{x:\text{int}, y:\text{int}\}\rho \triangleright (\text{int} * \text{int}))$

Runnable

$\square(\{x:\mathit{Abs}, y:\mathit{Abs}\}\rho \triangleright (\text{int} * \text{int}))$

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

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

$$\frac{\Gamma_2 <: \Gamma_1 \qquad A_1 <: A_2}{\Box(\Gamma_1 \triangleright A_1) <: \Box(\Gamma_2 \triangleright A_2)} \qquad \frac{\Gamma_2 <: \Gamma_1 \qquad A_1 <: A_2}{\Gamma_1 \rightarrow A_1 <: \Gamma_2 \rightarrow A_2}$$

Outline

- Motivation – The library specialization problem and the need for subtyping
- Translating staged ML to record calculus
- Typing staged ML via record calculus
- Subtyping
- **“Pluggable declarations”**
- Handling side effects
- Related work

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, proof provided in the thesis

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

Outline

- Motivation – The library specialization problem and the need for subtyping
- Translating staged ML to record calculus
- Typing staged ML via record calculus
- Subtyping
- "Pluggable declarations"
- **Handling side effects**
- Related work

Handling side effects

- In current translation, terms at level zero go inside abstractions: < …`(e)… > => λr. … e' ….
This changes order of evaluation.
- A more complicated translation is defined, such that <…`(e)… > => (λπ.λr. … π …) e'
- Order of evaluation preserved
- Properties proven for side-effect free language above can be proven here.

Outline

- Motivation – The library specialization problem and the need for subtyping
- Translating staged ML to record calculus
- Typing staged ML via record calculus
- Subtyping
- "Pluggable declarations"
- Handling side effects
- **Related work**

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

Contributions

- 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

(See loome.cs.uiuc.edu pubs page for details…)

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

Extra Slides

Translating Pluggable Declarations

- First attempt $[\![\langle x = e \rangle]\!]^n = \lambda r_n . r_n \text{ with } \{x = [\![e]\!]^{n+1}\}$
  - $\langle 5 \rangle \,\grave{}(\langle x = 2 \rangle) \Rightarrow (\lambda r_2.5)\,((\lambda r_2.r_2 \text{ with } \{x{=}2\})r1)$

    type-incorrect — type-correct

- Second attempt $[\![\langle x = e \rangle]\!]^n = [\![\lambda c.\langle \text{let } x = e \text{ in } \grave{}(c)\rangle]\!]^n$
  - $\langle x = 1 \rangle \langle 5 \rangle$ passes the type checker.
- Solution:

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

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

Cannot Type

- Because of rank-1 polymorphism, cannot type

$$\lambda y.(y\ 1,\ y\ 'a')$$

- Polymorphic types are not preserved after antiquotation/quotation

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