Formal Semantics for Weak References

Joe Hallett
joint work with
Kevin Donnelly and Assaf Kfoury

{jhallett,kevind,kfoury}@cs.bu.edu

Department of Computer Science
Boston University

What is a Weak Reference?

Excerpt from the SML/NJ Weak structure documention:
"A weak pointer is one that is not sufficient to keep an object alive. If some object is pointed to by ordinary pointers and by weak pointers, it will stay alive (not be garbage collected); but if it is pointed to only by weak pointers, then it may be garbage collected."
ie,

Let $M =$

```
let val a = 5
    val w = weak(a)
in w end
```

and $N =$

```
let val a = 5
    val w = weak(a)
in (w,a) end
```

Then `a` may be collected in M but not in N.

Why are Weak References Useful?

Weak references are useful when we want to store objects without allowing them to *permanently* occupy space, i.e., reachability via the data structure should not prevent garbage collection.

Example applications:

- memotables
- hash-consing
- caches

Why Problematic?

- Another excerpt from the SML/NJ Weak structure documention:
  “The semantics of weak pointers to immutable data structures in ML is ambiguous.”

- The introduction of weak references allows garbage collection to influence the result of a program.

Why Problematic? (cont)

Example (from SML/NJ documentation):

```
let val a = (1,2)
    val b = (1,2)
    val w = weak(a)
in (b, strong(w)) end
```

where `strong:'a weak -> 'a option`

Why Problematic? (cont)

Example (from SML/NJ documentation):

```
let val a = (1,2)
    val b = (1,2)
    val w = weak(a)
in (b, strong(w)) end
```

Result could be: ((1,2), NONE)

or could be: ((1,2), SOME(1,2))

where `strong:'a weak -> 'a option`

Why Problematic? (cont)

Example (from SML/NJ documentation):

```
let val a = (1,2)
    val b = (1,2)
    val w = weak(a)
in (b, strong(w)) end
```

Result could be: ((1,2), NONE)

or could be: ((1,2), SOME(1,2))

Problems:

- gc might not have happened

- common subexpression elimination

where `strong:'a weak -> 'a option`

Our Approach to Weak References

- In need of well-defined semantics

Our Approach to Weak References

- In need of well-defined semantics
- Semantics should be dependent on semantics of gc

Our Approach to Weak References

- In need of well-defined semantics
- Semantics should be dependent on semantics of gc
- Augment λ_{gc} (by Morrisett et. al.) with weak references and include gc rewrite rule in operational semantics

Our Approach to Weak References

- In need of well-defined semantics
- Semantics should be dependent on semantics of gc
- Augment λ_{gc} (by Morrisett et. al.) with weak references and include gc rewrite rule in operational semantics
- *Bad consequence:* programmer not oblivious to gc

Our Approach to Weak References

- In need of well-defined semantics
- Semantics should be dependent on semantics of gc
- Augment λ_{gc} (by Morrisett et. al.) with weak references and include gc rewrite rule in operational semantics
- *Bad consequence:* programmer not oblivious to gc
- Tension: precision vs. programmer knowledge

Our Approach to Weak References

- In need of well-defined semantics
- Semantics should be dependent on semantics of gc
- Augment λ_{gc} (by Morrisett et. al.) with weak references and include gc rewrite rule in operational semantics
- *Bad consequence:* programmer not oblivious to gc
- Tension: precision vs. programmer knowledge
  - define precise semantics (requires gc knowledge)

Our Approach to Weak References

- In need of well-defined semantics
- Semantics should be dependent on semantics of gc
- Augment λ_{gc} (by Morrisett et. al.) with weak references and include gc rewrite rule in operational semantics
- *Bad consequence:* programmer not oblivious to gc
- Tension: precision vs. programmer knowledge
  - define precise semantics (requires gc knowledge)
  - carve out subsets for reasoning without gc knowledge

Syntax of $\lambda_{\textbf{weak}}$

$$
\begin{array}{llcl}
\text{(variables)} & w, x, y, z & \in & \mathsf{Var} \\
\text{(integers)} & i & \in & \mathsf{Int} \quad ::= \quad \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \\
\text{(expressions)} & e & \in & \mathsf{Exp} \quad ::= \quad x \mid i \mid \langle e_1, e_2 \rangle \mid \pi_1\ e \mid \pi_2\ e \mid \lambda x.e \mid \\
 & & & \qquad\qquad\quad e_1\ e_2 \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid \\
 & & & \qquad\qquad\quad \mathtt{weak}\ e \mid \mathtt{ifdead}\ e_1\ e_2\ e_3 \\
\text{(heap values)} & hv & \in & \mathsf{Hval} \quad ::= \quad i \mid \langle x_1, x_2 \rangle \mid \lambda x.e \mid \mathtt{weak}\ x \mid \mathtt{d} \\
\text{(heaps)} & H & \in & \mathsf{Var} \xrightarrow{\mathsf{fin}} \mathsf{Hval} \\
\text{(programs)} & P & \in & \mathsf{Prog} \quad ::= \quad \mathsf{letrec}\ H\ \mathsf{in}\ e \\
\text{(answers)} & A & \in & \mathsf{Ans} \quad ::= \quad \mathsf{letrec}\ H\ \mathsf{in}\ x
\end{array}
$$

Rewrite Rules of $\lambda_{\mathbf{weak}}$

(alloc) $\quad \text{letrec } H \text{ in } E[hv] \xrightarrow{\text{alloc}} \text{letrec } H \uplus \{x \mapsto hv\} \text{ in } E[x]$

(proj) $\quad \text{letrec } H \text{ in } E[\pi_i\ x] \xrightarrow{\pi_i} \text{letrec } H \text{ in } E[x_i]$

$(H(x) = \langle x_1, x_2\rangle,\ i \in \{1, 2\})$

(app) $\quad \text{letrec } H \text{ in } E[x\ y] \xrightarrow{\text{app}} \text{letrec } H \text{ in } E[e\{z := y\}]$

$(H(x) = \lambda z.e)$

(let-in) $\quad \text{letrec } H \text{ in } E[\texttt{let } x = y \texttt{ in } e] \xrightarrow{\text{let-in}} \text{letrec } H \text{ in } E[e\{x := y\}]$

(ifdead) $\quad \text{letrec } H \text{ in } E[\texttt{ifdead } x\ e_1\ e_2] \xrightarrow{\text{ifdead}}$

$$\begin{cases} \text{letrec } H \text{ in } E[e_2\ w] & \text{if } H(x) = \texttt{weak } w \\ \text{letrec } H \text{ in } E[e_1] & \text{if } H(x) = \texttt{d} \end{cases}$$

Garbage Collection Rule

(garb) $\quad \mathsf{letrec}\ H\ \mathsf{in}\ e \xrightarrow{\mathsf{garb}} \mathsf{letrec}\ H'\ \mathsf{in}\ e$

provided $\mathsf{letrec}\ H\ \mathsf{in}\ e \xrightarrow{\mathsf{gc}} \mathsf{letrec}\ H''\ \mathsf{in}\ e$

and $\mathsf{letrec}\ H''\ \mathsf{in}\ e \Downarrow_{\mathsf{weak\text{-}gc}} \mathsf{letrec}\ H'\ \mathsf{in}\ e$

(gc) $\quad \mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ e \xrightarrow{\mathsf{gc}} \mathsf{letrec}\ H_1\ \mathsf{in}\ e$

provided $\mathsf{Dom}(H_2) \cap FV(\mathsf{letrec}\ H_1^s\ \mathsf{in}\ e) = \emptyset$,

and $H_2 \neq \emptyset$

(weak-gc) $\quad \mathsf{letrec}\ H \uplus \{x \mapsto \mathtt{weak}\ y\}\ \mathsf{in}\ e \xrightarrow{\mathsf{weak\text{-}gc}} \mathsf{letrec}\ H \uplus \{x \mapsto \mathtt{d}\}\ \mathsf{in}\ e$

provided $y \notin \mathsf{Dom}(H)$

Confluence is Lost

Introducing $\xrightarrow{\textsf{garb}}$ into the operational semantics means that confluence doesn't hold.

$$\textsf{letrec}\ \{\}\ \textsf{in}\ (\lambda x.\texttt{ifdead}\ (\texttt{weak}\ x)\ 0\ (\lambda y.\pi_1\ y))\ \langle 5, 6\rangle \xrightarrow{\textsf{G}}{}^{*}$$

$$\begin{cases} \textsf{letrec}\ \{a \mapsto \lambda x.\texttt{ifdead}\ (\texttt{weak}\ x)\ 0\ (\lambda y.\pi_1\ y), b \mapsto 5, c \mapsto 6, e \mapsto \langle b, c\rangle, \\ f \mapsto \texttt{weak}\ e, g \mapsto \lambda y.\pi_1\ y\}\ \textsf{in}\ b \\ \\ \textsf{letrec}\ \{f \mapsto \texttt{d}, g \mapsto 0\}\ \textsf{in}\ g \end{cases}$$

Well-Behavedness

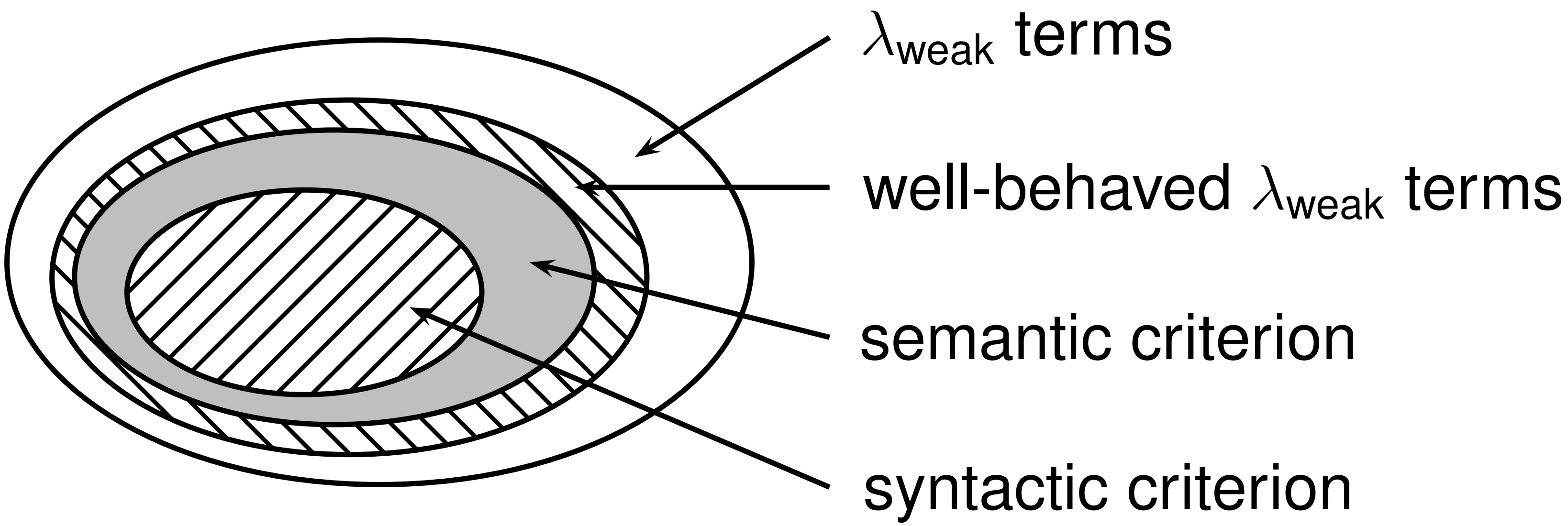
Definition 0.1 (Evaluation Set)**.** The evaluation set of a program P relative to a set of rewrite rules G:

$$\begin{aligned} \textsf{eval-set}(P, G) &= \{\bot \mid P \Uparrow_G\} \cup \\ &\quad \{\text{error} \mid P \Downarrow_G P' \text{ and } P' \text{ is stuck}\} \cup \\ &\quad \{\textsf{result}(H, x) \mid P \Downarrow_G \textsf{letrec } H \textsf{ in } x\} \end{aligned}$$

$\square$

Definition 0.2 (Well-Behaved Programs)**.** A program, P, is well-behaved iff $\textsf{eval-set}(P, G)$ is a singleton set – in words, iff either all evaluations of P diverge, or all evaluations of P get stuck, or all evaluations of P converge and return the same result. $\square$

Well-behavedness criteria



Syntactically Well-Behaved

Perhaps we can syntactically capture a subset of these programs:

$$\mathsf{letrec}\ \{\}\ \mathsf{in}\ \mathtt{ifdead}\ (\mathtt{weak}\ e)\ (e'\ e)\ e'$$

Syntactically Well-Behaved

We define a set of programs $\mathsf{Exp}^* \subset \mathsf{Exp}$ called *gc-oblivious* with rules such as:

$$\frac{e \ \in \mathsf{Exp}^*}{\lambda x.e \ \in \mathsf{Exp}^*}$$

$$\frac{(e_1, e_2) \ \in \mathsf{ExpPair} \quad e_3 \in \mathsf{Exp}^*}{\texttt{ifdead}\ e_1\ (e_3\ e_2)\ e_3 \ \in \mathsf{Exp}^*}$$

$$\frac{e \in \mathsf{Exp}^*}{(\texttt{weak}\ e, e) \ \in \mathsf{ExpPair}}$$

Theorem 0.3. *If P is gc-oblivious then it is well-behaved.* $\square$

General Semantic Criterion

- many well-behaved programs are not gc-oblivious
- $(e_1, e_2, e_3) \in \mathsf{Loc}_\Gamma$ means for all heaps respecting Γ if e_1 evaluates to `weak` hv then e_2 and e_3 hv evaluate to the same result.

Definition 0.4. A closed program letrec H in e is *locally well-behaved* iff it has a typing derivation $\vdash$ letrec H in $e : \tau$ and for all ifdead occurrences, $\Gamma \vdash \texttt{ifdead}\ e_1\ e_2\ e_3 : \tau'$, in the derivation we have $(e_1, e_2, e_3) \in \mathsf{Loc}_\Gamma$. □

Theorem 0.5. *If a program P is locally well-behaved then it is well-behaved.* □

Other Work

- Model other forms of weak references
  - Haskell (key/value weak references)
- Type system and type-inference-based gc
  - prevents overhead of re-computing data cached in weak references

Other Work

- Generalize semantics (weaken gc rule) to model compiler optimizations.

$$\text{letrec } \{a \mapsto \lambda x.\lambda y.x, b \mapsto 6, x \mapsto \langle a, b\rangle, y \mapsto \texttt{weak } x, f \mapsto \lambda x.x\}$$
$$\text{in } f\ (\pi_1\ x)\ (\texttt{ifdead } y\ 2\ (\lambda z.\pi_1\ z))\ (\pi_1\ x)$$

After CSE ...

$$\text{letrec } \{a \mapsto \lambda x.\lambda y.x, b \mapsto 6, x \mapsto \langle a, b\rangle, y \mapsto \texttt{weak } x, f \mapsto \lambda x.x\}$$
$$\text{in } \texttt{let } c = \pi_1\ x \texttt{ in } f\ c\ (\texttt{ifdead } y\ 2\ (\lambda z.\pi_1\ z))\ c$$

Summary

- λ_{weak} provides framework for reasoning about semantics of weak references, a troublesome programming feature

Summary

- λ_{weak} provides framework for reasoning about semantics of weak references, a troublesome programming feature
- Flexible framework - encode Haskell key/value weak references (Java?)

Summary

- λ_{weak} provides framework for reasoning about semantics of weak references, a troublesome programming feature
- Flexible framework - encode Haskell key/value weak references (Java?)
- Prove efficiently-decidable (but restrictive) set of well-behaved programs

Summary

- λ_{weak} provides framework for reasoning about semantics of weak references, a troublesome programming feature
- Flexible framework - encode Haskell key/value weak references (Java?)
- Prove efficiently-decidable (but restrictive) set of well-behaved programs
- Give general (but undecidable) semantic criterion for well-behavedness - used by programmer as an alternative to strict syntatic discipline

Related Work

- Morrisett, Felleisen, Harper give formal semantics for garabage collection
- Peyton-Jones, Marlow, and Elliot discuss key/value weak references of Haskell
- Most weak reference related work is in implementations of programming languages (SML, Haskell, Java, Scheme)

Future Work

- Model weak/soft/phantom references in Java
- Extend λ_{weak} with other programming language features: mutable references, recursion, etc.
- Extend decidable syntax criterion for well-behavedness - dependent types
- Statically capture other properties besides well-behavedness: no space leaks due to unintented aliasing, prioritized finalization