# Foundations of Strong Call-by-Need


Thibaut Balabonski Pablo Barenbaum

Eduardo Bonelli Delia Kesner



INFINIS Workshop
October 19, 2017

# Weak vs. strong reduction

Typical programming languages use **weak reduction**.

- Functions are considered values.
- Bodies of functions are not evaluated.

The following are final results:

- $\lambda$x. `while (True)` $\{\}$ *does not hang,*
- $\lambda$x. `1 / 0` *does not throw a zero-division exception,*
- $\lambda$x. $\textsc{Ackermann}(42)$ *does not perform any computation.*

# Weak vs. strong reduction

**Example of weak reduction (in Python)**

```
def loop():
    while True:
        pass

>>> lambda x: loop()
<function <lambda> at 0x7f254a423320>
```

# Weak vs. strong reduction

Weak reduction is fine from a programming standpoint.

However, proof assistants based on dependent type-theory:

- Coq,
- Agda,
- Isabelle/HOL,
- Lean,
- etc.

require **strong reduction** to decide definitional equality.

# Weak vs. strong reduction

## Example of **strong reduction** (in Coq)

```
Definition times2 (x : nat) : nat := x + x.
Definition injective (f : nat -> nat) : Prop :=
  forall x y,  f x = f y  ->  x = y.
Hypothesis A : injective (fun x => x + x).
Lemma B : injective (fun x => times2 x).
  apply A.
Qed.
```

The propositions

```
injective (fun x => x + x)
```

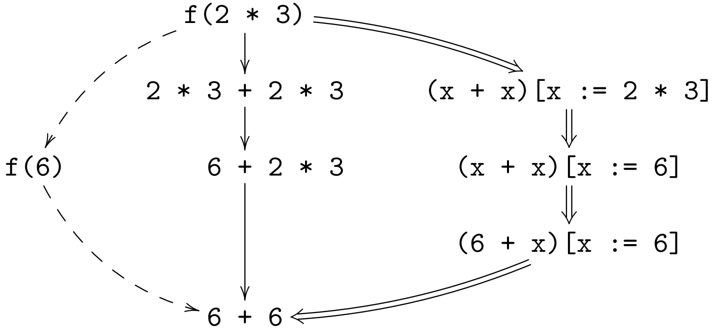
```
injective (fun x => times2 x)
```

should be equal by definition, which requires strong reduction.

# Weak evaluation strategies

There are many well-known evaluation strategies for weak reduction.

For instance, if `f(x) = x + x`, then:




$- - \succ$ = **call-by-value** $\longrightarrow$ = **call-by-name** $\Longrightarrow$ = **call-by-need** = “**lazy**”

# Goal

The call-by-need strategy is **optimal** for weak reduction:

- It does not duplicate computations.
  Subterms are *shared*.
- It does not perform useless computations.
  Only *needed* steps are taken.

[Balabonski, 2013]

**Our goal**

Extend the call-by-need strategy for strong reduction.

# The call-by-need calculus

The call-by-need calculus is an **explicit substitution calculus**.

**Term syntax**

$$
\begin{array}{rcll}
t & ::= & x & \text{variable} \\
 & \mid & \lambda x.t & \text{abstraction} \\
 & \mid & t\,t & \text{application} \\
 & \mid & t[x := t] & \text{explicit substitution}
\end{array}
$$

**Call-by-need reduction**

$$
\begin{array}{rcll}
(\lambda x.t)\mathtt{L}\,s & \to & t[x := s]\mathtt{L} & \\
\mathtt{C}[x][x := (\lambda y.t)\mathtt{L}] & \to & \mathtt{C}[\lambda y.t][x := \lambda y.t]\mathtt{L} & \\
t[x := s] & \to & t & \text{if } x \text{ does not occur free in } t
\end{array}
$$

**Note.** Calculi are usually non-deterministic.

# The weak call-by-need strategy

We are interested in **deterministic** call-by-need strategies.

**Weak call-by-need** is defined using simple **evaluation contexts**.

$$
\begin{array}{rcl}
\mathtt{E} & ::= & \Box \\
& | & \mathtt{E}\, t \\
& | & \mathtt{E}[x := t] \\
& | & \mathtt{E}[x][x := \mathtt{E}]
\end{array}
$$

This notion of weak call-by-need coincides with the standard weak call-by-need strategy by Ariola, Felleisen, Maraist, Odersky, and Wadler.

# The weak call-by-need strategy

## Example of weak call-by-need reduction

Let $\texttt{id} = \lambda x.\, x$. Then:

$$
\begin{array}{rcl}
(\lambda x.\, x\, x)(\texttt{id}\,\texttt{id}) & \rightarrow & (x\, x)[x := \texttt{id}\,\texttt{id}] \\
& \rightarrow & (x\, x)[x := y[y := \texttt{id}]] \\
& \rightarrow & (x\, x)[x := \texttt{id}[y := \texttt{id}]] \\
& \rightarrow & (\texttt{id}\, x)[x := \texttt{id}][y := \texttt{id}] \\
& \rightarrow & z[z := x][x := \texttt{id}][y := \texttt{id}] \\
& \rightarrow & z[z := \texttt{id}][x := \texttt{id}][y := \texttt{id}] \\
& \rightarrow & \texttt{id}[z := \texttt{id}][x := \texttt{id}][y := \texttt{id}]
\end{array}
$$

# The strong call-by-need strategy

**Strong call-by-need** is also defined using evaluation contexts. The two main technical challenges are:

- **Context-dependency.**

  $(\lambda x.t)[y := s]$ the body $t$ must be evaluated

  $(\lambda x.t)[y := s]u$ the body $t$ must **not** be evaluated yet

- **Frozen variables.**

  $\lambda x.\, x\, t$ $x$ is frozen, so $t$ must be evaluated

  $\lambda x.\, y[y := x\, t]$ $x, y$ are frozen, so $t$ must be evaluated

  $(x\, t)[x := I]$ $x$ is not frozen, so $t$ must **not** be evaluated

# The strong call-by-need strategy

To deal with these, evaluation contexts are **parameterized**:

- Evaluation contexts depend on a **set of frozen variables** $\phi$.

$$\begin{array}{ll} (x\ z)[z := y\ \Box] & \text{is an } \{x, y\}\text{-evaluation context} \\ (x\ z)[z := y\ \Box] & \text{is not a } \{x\}\text{-evaluation context} \end{array}$$

- There are **two kinds of evaluation contexts**:
  - **Unrestricted evaluation contexts.**
  - **Inert evaluation contexts.**
    Unable to interact with a surrounding application.

$$\begin{array}{ll} (x\ y)[y := z\ \Box] & \text{is inert} \\ (\lambda x.y)[y := z\ \Box] & \text{is not inert} \end{array}$$

Details in the ICFP 2017 paper.

# The strong call-by-need strategy

## Example of strong call-by-need reduction

Let $\texttt{id} = \lambda x.\, x$. Then:

$$
\begin{array}{rcl}
(\lambda x.\, \lambda y.\, \texttt{id}\,(y\,x)])\,(\texttt{id}\,\texttt{id}) & \to & (\lambda y.\, \texttt{id}\,(y\,x)])[x := \texttt{id}\,\texttt{id}] \\
 & \to & (\lambda y.\, z[z := y\,x]])[x := \texttt{id}\,\texttt{id}] \\
 & \to & (\lambda y.\, z[z := y\,x]])[x := w[w := \texttt{id}]] \\
 & \to & (\lambda y.\, z[z := y\,x]])[x := \texttt{id}[w := \texttt{id}]] \\
 & \to & (\lambda y.\, z[z := y\,\texttt{id}]])[x := \texttt{id}][w := \texttt{id}]
\end{array}
$$

# Results: conservativity

**Theorem** (Conservativity)

Strong call-by-need is a **conservative extension** of weak call-by-need.

More precisely, if the weak call-by-need strategy picks a step $t \to s$, the strong call-by-need strategy picks the same step.

The proof is easy (once the calculus is got right).

- Observe that if $\mathtt{E}$ is a weak evaluation context, then $\mathtt{E}$ is also a strong evaluation context.

# Results: completeness

**Theorem** (Completeness)

If a term $t$ has a normal form in the $\lambda$-calculus, then the strong call-by-need strategy also reduces $t$ to a normal form.

Furthermore, there is a precise relation between the two:

$$\begin{tikzcd} t \arrow[rr, twoheadrightarrow, "\beta"] \arrow[d, twoheadrightarrow, "\texttt{need-strategy}"'] & & t' \\ t'' \arrow[urr, dotted, twoheadrightarrow, "\texttt{unfold}"'] & & \end{tikzcd}$$

The proof is hard and the core of the ICFP 2017 paper.
It relies on two technical tools:

- A non-idempotent intersection type system $\mathcal{W}$.
- Exhaustive case-analysis of permutation diagrams.

# Future/ongoing work

- Define an abstract machine for strong call-by-need evaluation.
- Characterize strong call-by-need via a big-step semantics.
- Extend the strategy for further programming constructs, such as pattern matching.

Questions?