Operational Semantics

- Granularity
- Small-step semantics, big-step semantics
- Order of evaluation: call-by-name, call-by-value, call-by-need, perpetual strategy
- Abstract machines

Big-step Semantics

Consider a very elemental calculator handling expressions belonging to the following grammar:

\[ e ::= n \mid X \mid e + e. \]

Each rule completely evaluates an expression under a substitution to a value.

\[
\begin{align*}
\langle n, \sigma \rangle &\Downarrow n \\
\langle X, \sigma \rangle &\Downarrow \sigma(X) \\
\langle a_1 + a_2, \sigma \rangle &\Downarrow n \\
\langle a_1, \sigma \rangle &\Downarrow n_1 \\
\langle a_2, \sigma \rangle &\Downarrow n_2 \quad n \text{ is } "n_1 \text{ plus } n_2" \\
\langle a_1, \sigma \rangle &\Downarrow n_1 \\
\langle a_2, \sigma \rangle &\Downarrow n_2
\end{align*}
\]

Properties

- Abstract
- Allows to avoid details
- No specification of evaluation order (e.g. \((1 + 3) + (5 − 3)\))
- No specification of control of errors
- No specification of interleaving
Evaluation of an expression under a substitution is given by a sequence of state changes which terminates when the state cannot be reduced further.

\[ \langle a_1, \sigma \rangle \xrightarrow{\cdot} \langle a'_1, \sigma' \rangle \]
\[ \langle a_1 + a_2, \sigma \rangle \xrightarrow{\cdot} \langle a'_1 + a'_2, \sigma' \rangle \]
\[ (X, \sigma) \xrightarrow{\cdot} \langle \sigma(X), \sigma \rangle \]

\[ n \text{ is } n_1 + n_2 \]

\[ \langle n_1 + n_2, \sigma \rangle \xrightarrow{\cdot} \langle n, \sigma \rangle \]

\[ \langle X, \sigma \rangle \xrightarrow{\cdot} \langle \sigma(X), \sigma \rangle \]

Properties

Less abstract

Specification of order of evaluation

Control of errors: \( \frac{n_2 \neq 0}{n_1/n_2 \rightarrow n} \), where \( n \) is "\( n_1 \) divided by \( n_2 \)".

Interleaving:

\[ \langle c_1, \sigma \rangle \xrightarrow{\cdot} \langle c'_1, \sigma' \rangle \]
\[ \langle c_1 \| c_2, \sigma \rangle \xrightarrow{\cdot} \langle c'_1 \| c'_2, \sigma' \rangle \]

From Small-step to Multi-step Semantics

The multi-step semantics is given by the relation \( t \rightarrow^* t' \) which is the reflexive and transitive closure of \( t \rightarrow t' \).

(P1) \( t \rightarrow^* t \) for every \( t \)

(P2) \( t \rightarrow t' \) implies \( t \rightarrow^* t' \)

(P3) \( t \rightarrow^* t' \) and \( t' \rightarrow^* t'' \) implies \( t \rightarrow^* t'' \)

Properties of the small and big step semantics

The relation \( \rightarrow^* \) is deterministic.

The relation \( \parallel \) is deterministic.

\( t \parallel v \text{ iff } t \rightarrow^* v \), where \( v \) is a "value".
In small-step semantics evaluation stops at errors. In big-step semantics errors occur deeply inside derivation trees.

The order of evaluation is explicit in small-step semantics but implicit in big-step semantics.

Big-step semantics is more abstract, but less precise.

Small-step semantics allows to make difference between non-termination and "getting stuck".

A functional language

\[ t, u ::= x \quad \text{(variable)} \quad | \quad c \quad \text{(constant)} \quad | \quad (t, u) \quad \text{(pair)} \quad | \quad t \ u \quad \text{(application)} \quad | \quad \lambda x. t \quad \text{(abstraction)} \quad | \quad \text{let } x = t \ \text{in } u \quad \text{(let)} \]

Some constant function symbols: fst, snd, ifthenelse, +, * ...

Some constants: true, false, 0, 1, 2, 3 ...

Thus e.g. \( \text{if } t \ \text{then } u \ \text{else } v \) can be defined as ifthenelse\( (t, (u, v)) \).

A program is a closed expression belonging to the previous grammar.

Call-by-value lambda-calculus (big-step semantics)

(Values) \( V, W ::= c \ | \ (V, V) \ | \ \lambda x. t \)

Meaningless expressions such as \((1, 1) 3\) or \(\text{true } 3\) are not considered as values.

\[
\begin{align*}
V \text{ is a value} & \quad t_1 \downarrow V_1 \quad t_2 \downarrow V_2 \quad u \downarrow V \quad r[x/V] \downarrow W \\
& \quad \text{let } x = u \ \text{in } r \downarrow W \\
& \quad t \downarrow \lambda x. r \quad u \downarrow W \\
& \quad \text{let } x = u \ \text{in } r \downarrow W \\
& \quad t \ u \downarrow V \\
& \quad t \downarrow \text{fst} \\
& \quad t \ u \downarrow V_1 \\
& \quad t \downarrow \text{snd} \\
& \quad t \ u \downarrow V_2 \\
\end{align*}
\]

Particular case: closed pure lambda-terms

(Values) \( V ::= \lambda x. t \)

\[
\begin{align*}
& \quad t \downarrow V_1 \\
& \quad t \downarrow \text{ifthenelse} \\
& \quad t \downarrow V_1 \\
& \quad t \downarrow \text{ifthenelse} \\
& \quad t \downarrow V_1 \\
& \quad t \downarrow \text{ifthenelse} \\
& \quad t \downarrow V_1 \\
& \quad t \downarrow \text{ifthenelse} \\
& \quad t \downarrow V_1 \\
& \quad t \downarrow \text{ifthenelse} \\
& \quad t \downarrow V_1 \\
\end{align*}
\]
Call-by-name lambda-calculus (big-step semantics)

\[ t = λf.λx.(x, f x) \text{ and } u = λy.y. \]

\[
\begin{array}{c}
\varepsilon \vdash t u 1 \vdash λx.(x, u x) 1 \vdash (1, u 1) \vdash (1, 1) \\
\varepsilon \vdash t u 1 \vdash λx.(x, u x) 1 \vdash (1, 1) \\
\varepsilon \vdash t u 1 \vdash λx.(x, u x) 1 \vdash (1, 1) \\
\varepsilon \vdash (1, u 1) \vdash λx.(x, u x) 1 \vdash (1, 1) \\
\end{array}
\]

The same example

\[
t = λf.λx.(x, f x) \text{ and } u = λy.y.\]

\[
\begin{array}{c}
t u 1 \vdash \langle x, u x \rangle 1 \vdash (1, 1) \\
t u 1 \vdash \langle x, u x \rangle 1 \vdash (1, 1) \\
t u 1 \vdash \langle x, u x \rangle 1 \vdash (1, 1) \\
t u 1 \vdash \langle x, u x \rangle 1 \vdash (1, 1) \\
\end{array}
\]

Call-by-value lambda calculus (small-step semantics)

\[
lam (λx.λy.(y f)) 1 \vdash \langle 1, u 1 \rangle \vdash \langle 1, 1 \rangle \\
\]

\[
lam (λx.λy.(y f)) 1 \vdash \langle 1, u 1 \rangle \vdash \langle 1, 1 \rangle \\
\]

\[
lam (λx.λy.(y f)) 1 \vdash \langle 1, u 1 \rangle \vdash \langle 1, 1 \rangle \\
\]

\[
lam (λx.λy.(y f)) 1 \vdash \langle 1, u 1 \rangle \vdash \langle 1, 1 \rangle \\
\]

Call-by-name lambda-calculus (big-step semantics)

\[
\begin{array}{c}
(\textbf{Lazy Forms}) \quad P ::= c \mid (t, u) \mid λx.t \\
\end{array}
\]

\[
P \text{ is a lazy form } \\
\begin{array}{c}
\varepsilon \vdash P u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]

\[
\begin{array}{c}
\varepsilon \vdash r[x/u] u_n P \\
\varepsilon \vdash r[x/u] u_n P \\
\end{array}
\]
Particular case: closed pure lambda-terms

(Lazy Forms) \( P ::= \lambda x.t \)

\[
\begin{array}{c}
P \Downarrow_n P \\
\hline
\hline
P \Downarrow_n P \\
\hline
\hline
P \Downarrow_n P
\end{array}
\]

Coherence of results

\[
\begin{array}{c}
\text{If } t \Downarrow_n v, \text{ then } u \text{ is a value.} \\
\text{If } t \Downarrow_n u, \text{ then } u \text{ is a lazy form.}
\end{array}
\]
Deterministic properties

- If \( t \Downarrow V \) and \( t \Downarrow V' \), then \( V = V' \).
- If \( t \Downarrow P \) and \( t \Downarrow P' \), then \( P = P' \).
- If \( t \rightarrow^* u \) and \( t \rightarrow^* u' \), then \( u = u' \).
- If \( t \rightarrow^n u \) and \( t \rightarrow^n u' \), then \( u = u' \).

Relating big and small-steps semantics

- If \( t \downarrow V \), then \( t \rightarrow^* V \).
- If \( t \downarrow P \), then \( t \rightarrow^n P \).
- If \( t \rightarrow^* u \) and \( u \) is a value, then \( t \downarrow u \).
- If \( t \rightarrow^n u \) and \( u \) is a lazy form, then \( t \downarrow u \).

Abstract Machines

Call-by-Need
Lazy Evaluation and Call-by-Need

Lazy Evaluation (Wadsworth'71)
- Based on demand-driven computation
- Implements memoization (the first demand-driven function call transforms the argument into a value)
- May manipulate potentially infinite data
- The best of call-by-name and the best of call-by-value
- Modeled by call-by-need calculi (Ariola-Felleisen).

Call-by-need different from call-by-name

Call-by-need is different from call-by-name: first evaluates arguments (like call-by-value).

\[ \text{Twice} (4 + 3) \rightarrow_{\text{cbname}} (4 + 3) + (4 + 3) \rightarrow_{\text{cbname}} 7 + (4 + 3) \rightarrow_{\text{cbname}} 7 + 7 \rightarrow_{\text{cbname}} 14 \]
\[ \text{Twice} (4 + 3) \rightarrow_{\text{cbneed}} \text{Twice} 7 \rightarrow_{\text{cbneed}} 7 + 7 \rightarrow_{\text{cbneed}} 14 \]

where \( \text{Twice} = \lambda x. x + x \).

Call-by-need different from call-by-value

Call-by-need is different from call-by-value: values are only consumed when required

\[ (\lambda x.8)(4 + 3) \rightarrow_{\text{cbvalue}} (\lambda x.8)7 \rightarrow_{\text{cbvalue}} 8 \]
\[ (\lambda x.8)(4 + 3) \rightarrow_{\text{cbneed}} 8 \]

In particular

\[ (\lambda x.8)\Omega \Rightarrow_{\text{cbvalue}} \]
\[ (\lambda x.8)\Omega \rightarrow_{\text{cbneed}} 8 \]
(Syntactical) call-by-need different from (semantical) neededness

**Syntax:** Call-by-need

Evaluation strategy defined syntactically, using a notion of need context and let-constructors.

**Semantics:** Neededness

Evaluation strategy defined semantically, using the residual theory of λ-calculus.

But....

Observational equivalence, formally

Let $R$ be a reduction relation with an associated notion of result (e.g. value). We write $t \Downarrow_R$ if the term $t$ converges to some result.

Two terms $t$ and $u$ are said to be observationally equivalent for $R$, written $t \equiv_R u$, if for every context $C$,

$$C[t] \Downarrow_R \text{ if and only if } C[u] \Downarrow_R.$$

- Terms that cannot be distinguished by any context.
- Often used to compare two different implementations or protocols.
- Difficult to reason because of the universal quantification of contexts.

Theorem

Given a program $t$, the call-by-name interpreter on $t$ stops in a value if and only if the call-by-need interpreter on $t$ stops in an answer.

Said differently,

Theorem

Given $t$ and $u$ we have that $t \equiv_{cbname} u$ if and only if $t \equiv_{cbneed} u$. 
Call-by-need observationally equivalent to call-by-name

Theorem
Given a program $t$, the call-by-name interpreter on $t$ stops in a value if and only if the call-by-need interpreter on $t$ stops in an answer.

Said differently,
Theorem
Given $t$ and $u$ we have that $t \cong_{cbname} u$ if and only if $t \cong_{cbneed} u$.

Defining weak call-by-need

Explicit binding of arguments

First Reduction Rule:

$$\lambda x. t \rightarrow_B t[x/u]$$

Explicit binding of arguments

First Reduction Rule:

$$\lambda x. t \rightarrow_B t[x/u]$$
Defining weak call-by-need

Explicit binding of arguments

\[(\lambda x . id) (\lambda y . y)\]

\[\rightarrow\]

\[\text{let } x = (\lambda y . y) \text{ in } id\]

(also written \(id[x \ (\lambda y . y)]\))

First Reduction Rule:

\[(\lambda x . t) u \rightarrow_B t[x \ u]\]

Explicit binding of arguments again

\[(\lambda x . id) [y \Omega] (\lambda y . y)\]

\[\rightarrow\]

\[id[x \ (\lambda y . y)] [y \Omega]\]

First (Revised) Reduction Rule:

\[(\lambda x . t) [x_1 \ u_1] \ldots [x_n \ u_n] u \rightarrow_B t[x \ u] [x_1 \ u_1] \ldots [x_n \ u_n]\]

Substituting values when needed

\[x [x \ y]\]

\[\rightarrow\]

\[(\lambda y . y) [x \ y]\]

Second Reduction Rule:

\[N[[x \ V]] [x / V] \rightarrow_{lsv} N[[V]] [x / V]\]

In the example \(N = \Box\), but what is \(N\) in general??
**Defining weak call-by-need**

### Substituting values when needed

<table>
<thead>
<tr>
<th>In the example ( N = \square ), but what is ( N ) in general?</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N[[x]] ) ( [x/V] ) ( \mapsto_{1sv} ) ( N[[V]] ) ( [x/V] )</td>
</tr>
</tbody>
</table>

### Second Reduction Rule:

\[
N[\[x\]] \[x/V\] \mapsto_{1sv} N[\[V\]] \[x/V\]
\]

In the example \( N = \square \), but what is \( N \) in general??
Sharing while Substituting

\[
\begin{align*}
\lambda x.(\lambda y. y)[z/\text{id}] & \rightarrow \\
\lambda x.\lambda y. y[z/\text{id}] & \rightarrow \\
(\lambda y.)(\lambda y. y)[z/\text{id}] & \rightarrow
\end{align*}
\]

Second (Revised) Reduction Rule:

\[
N[[\lambda x.\lambda y. y]/V\ [x_1/\text{id}]\ldots[x_n/\text{id}]] \rightarrow_{\text{lsv}} \\
N[[V]/V\ [x_1/\text{id}]\ldots[x_n/\text{id}]]
\]

Reduction Rules:

- **(Beta reduction)**
  \[
  \lambda x.t\ [x_1/\text{id}]\ldots[x_n/\text{id}] \rightarrow_{s} \ [x\ [x_1/\text{id}]\ldots[x_n/\text{id}]]
  \]

- **(Partial substitution)**
  \[
  N[[\lambda y.\lambda z.\text{id} y]/V\ [x_1/\text{id}]\ldots[x_n/\text{id}]] \rightarrow_{1sv} N[[V]/V\ [x_1/\text{id}]\ldots[x_n/\text{id}]]
  \]

Closed by need contexts \( N \)

A full example:

\[
(\lambda x.\text{id} \ (x\ \text{id})) \ (\lambda y.\lambda z.\text{id} y)
\]
A Call-by-Need Calculus (Small-Step Semantics)

**Closed by need contexts**

A full example:

\[
(\lambda x . \text{id} (x \text{id})) (\lambda y . \lambda z . \text{id} y) \rightarrow_B
\]

**Reduction Rules:**

1. **Partial substitution**
   \[
   \frac{N \mid x \midV [x_1 \ldots x_n]}{N \mid x \midV [x_1 \ldots x_n] \rightarrow_{1s} N \mid x \midV [x_1 \ldots x_n]}
   \]

2. **Beta reduction**
   \[
   \frac{N \mid x \mid \beta [t], x \mid \beta [t]}{N \mid x \mid \beta [t], x \mid \beta [t]}
   \]

A full example:

\[
(\lambda x . \text{id} (x \text{id})) (\lambda y . \lambda z . \text{id} y) \rightarrow_B
\]

\[
(\lambda x_1 . x_1) (x \text{id}) [x / \lambda y . \lambda z . \text{id} y]
\]
A Call-by-Need Calculus (Small-Step Semantics)

- **Reduction Rules:**
  - (Beta reduction)
    \[(\lambda x . t_1)[x_1 \ldots x_n] \rightarrow_\beta t_1[x_1 \ldots x_n] \]
  - (Partial substitution)
    \[N \ll [V \ll [x_1 \ldots x_n] \rightarrow_{1sv} N \ll [V \ll [x_1 \ldots x_n]] \]

- Closed by need contexts \( N \)

A full example:

\[
(\lambda x . id (x \, id)) (\lambda y . z . id \, y) \rightarrow_B \\
(\lambda x_1 . x_1) (x \, id) [x / \lambda y . z . id \, y] \rightarrow_B \\
w[w / x \, id][x / \lambda y . z . id \, y] \rightarrow_{1sv}
\]

A Call-by-Need Calculus (Small-Step Semantics)

- **Reduction Rules:**
  - (Beta reduction)
    \[(\lambda x . t_1)[x_1 \ldots x_n] \rightarrow_\beta t_1[x_1 \ldots x_n] \]
  - (Partial substitution)
    \[N \ll [V \ll [x_1 \ldots x_n] \rightarrow_{1sv} N \ll [V \ll [x_1 \ldots x_n]] \]

- Closed by need contexts \( N \)

A full example:

\[
(\lambda x . id (x \, id)) (\lambda y . z . id \, y) \rightarrow_B \\
(\lambda x_1 . x_1) (x \, id) [x / \lambda y . z . id \, y] \rightarrow_B \\
w[w / x \, id][x / \lambda y . z . id \, y] \rightarrow_{1sv}
\]
Reduction Rules:

(Beta reduction)

\((\lambda x. t_1 \ldots \lambda x_n u_t) \xrightarrow{\beta} \lambda x_1 \ldots x_n. t_1 \ldots t_n u_t)\)

(Partial substitution)

\(N \vdash x \xrightarrow{\text{sub}} V [x/x_1 \ldots x_n] \xrightarrow{\text{isv}} N[V] \vdash x \xrightarrow{\text{or}} V [x/x_1 \ldots x_n] \)

Closed by need contexts \(N\)

A full example:

\((\lambda x. \text{id} (x \text{id})) (\lambda y. \lambda z. \text{id} y) \xrightarrow{\beta} (\lambda y. \lambda z. \text{id} y)\)

\((\lambda x_1. x_1) (x \text{id}) \xrightarrow{\beta} (x/\lambda y. \lambda z. \text{id} y)\)

\(w[w/x. \text{id}] [x/\lambda y. \lambda z. \text{id} y] \xrightarrow{\text{isv}} \)

\(w[w/(\lambda y. \lambda z. \text{id} y) \text{id}] [x/\lambda y. \lambda z. \text{id} y] \xrightarrow{\beta} \)

\(w[w/(\lambda z. \text{id} y) \text{id}] [x/\lambda y. \lambda z. \text{id} y] \)
Given any reduction step \( t \leadsto_B t' \):
- If \( t' \notin S N_B \), then \( t \notin S N_B \).
- If \( t' \in S N_B \), then \( t \) is not necessarily in \( S N_B \), e.g. \( t = (\lambda z.I)(\Delta \Delta) \leadsto_B z \).

A reduction strategy \( \leadsto \) for the \( \lambda \)-calculus is perpetual iff \( \leadsto \) and \( t \notin S N_B \) implies \( t' \notin S N_B \) (equiv. \( t' \in S N_B \) implies \( t \in S N_B \)).

Call-by-name is not perpetual: \((\lambda x.I)(\Delta \Delta) \leadsto_n I \) and \((\lambda x.I)(\Delta \Delta) \notin S N_B \) but \( I \in S N_B \).
(Strong) call-by-value is not perpetual: \((\lambda x.I)(\Delta \Delta) \leadsto_v I \) and \((\lambda x.I)(\lambda z.\Delta \Delta) \notin S N_B \) but \( I \in S N_B \).
A Perpetual Strategy for the $\lambda$-calculus

Theorem (Perpetuality Theorem)
The reduction strategy of the previous frame is perpetual, i.e. if $t \rightsquigarrow t'$ and $t \not\in SN_{\beta}$, then $t' \not\in SN_{\beta}$.

Theorem (Perpetuality and SN)
The term $t$ is $\rightsquigarrow$-terminating if and only if $t \in SN_{\beta}$.

Example: $(\lambda x. I)(\Delta \Delta) \rightsquigarrow (\lambda x. I)(\Delta \Delta)$.

Abstract Machines
Lambda-calculus is too abstract.

Semantics of programming languages needs an implementation of substitution.

Abstract machines fill the gap between high-level specifications of languages and real implementations.

Abstract machines: virtual machine, a model of computation,

Abstract machines: states (term, stack, environment) and deterministic transitions.

Some references:

- SECD machine for call-by-value (Landin 1964)
- KAM machine for call-by-name (Krivine 2007)
- ZINC machine for (right-to-left) call-by-value (Leroy 1990)
- CEK machine for (left-to-right) call-by-value (Felleisen-Friedman 1987)
- Call-by-need (Sestoft 1997)
- Strong call-by-name (Cregut 2007)

The Krivine Abstract Machine (KAM)

Main ingredients:

- An environment is a list of elements of the form \([x]\ c\), where \(c\) is a closure.
- A closure is a pair term and environment.
- A state of the KAM is a 3-uple Term | Environment | Stack.

The transitions between states:

- \( \lambda x.t | e |\ c :: \pi \mapsto t | [x]\ c :: e |\ pi \)
- \( \lambda w.w \mapsto \emptyset \)
- \( x \mapsto t | e' |\ pi \) where \( e(x) = (t, e') \)
- \( \lambda x.x \mapsto \emptyset \)
- \( \lambda z.z \mapsto \emptyset \)
- \( \lambda \emptyset \emptyset \mapsto \emptyset \)

Example

\((\lambda x.x)(\lambda z.z)(\lambda w.w)\ |\ \text{nil} | \text{...} \mapsto \)

\( x \ |\ [x]\ (\lambda z.z, \text{nil}) \ | \ (\lambda w.w, \text{nil}) \mapsto \)

\( \lambda z.z \ |\ \text{nil} \ | \ (\lambda w.w, \text{nil}) \mapsto \)

\( \lambda \emptyset \emptyset \ | \ \text{nil} \ | \ \text{empty} \mapsto \)