How to Replace Sharing with Fortune-Telling!

Jennifer Hackett
University of Nottingham
Introduction
Introduction

- This is a talk about cost semantics – mathematical models of the cost of evaluating programs
Introduction

- This is a talk about cost semantics – mathematical models of the cost of evaluating programs
- This is also a talk about evaluation strategies, and how to characterise lazy evaluation
Introduction

- This is a talk about cost semantics – mathematical models of the cost of evaluating programs

- This is also a talk about evaluation strategies, and how to characterise lazy evaluation

- Finally, this is a talk about what counts as a “pure” programming language
Improvement Theory
I work on improvement theory, an *algebraic* approach to reasoning about the *relative* efficiency of programs.
I work on improvement theory, an algebraic approach to reasoning about the relative efficiency of programs.

Essentially the study of the algebraic properties of the improvement relation:
I work on improvement theory, an algebraic approach to reasoning about the relative efficiency of programs.

Essentially the study of the algebraic properties of the improvement relation:

\[ e \succ e' \iff \forall C. C[e] \downarrow_k \Rightarrow C[e'] \downarrow_{\leq k} \]
Improvement Theory

- I work on improvement theory, an *algebraic* approach to reasoning about the *relative* efficiency of programs.

- Essentially the study of the algebraic properties of the *improvement relation*:

  \[ e \geq e' \iff \forall C. C[e] \downarrow_k \Rightarrow C[e'] \downarrow_{\leq k} \]

- We say "*e is improved by e'""
Call-by-Value vs. Call-by-Name
Call-by-Value vs. Call-by-Name

- Two ways to evaluate function application...
Call-by-Value vs. Call-by-Name

- Two ways to evaluate function application...
- Call-by-Value evaluates \( f(x) \) by first reducing \( x \) to some sort of *normal form*, then substitutes it into the body of \( f \)
Call-by-Value vs. Call-by-Name

- Two ways to evaluate function application...

- Call-by-Value evaluates $f(x)$ by first reducing $x$ to some sort of normal form, then substitutes it into the body of $f$

  - Easy to understand and costs are compositional, but has strict semantics, i.e. divergence propagates
Call-by-Value vs. Call-by-Name

- Two ways to evaluate function application...

- Call-by-Value evaluates f(x) by first reducing x to some sort of normal form, then substitutes it into the body of f
  - Easy to understand and costs are compositional, but has strict semantics, i.e. divergence propagates

- Call-by-Name evaluates f(x) by substituting x into the body of f regardless of whether it is in a normal form
Call-by-Value vs. Call-by-Name

- Two ways to evaluate function application...

- Call-by-Value evaluates $f(x)$ by first reducing $x$ to some sort of *normal form*, then substitutes it into the body of $f$
  - Easy to understand and costs are compositional, but has *strict* semantics, i.e. divergence propagates

- Call-by-Name evaluates $f(x)$ by substituting $x$ into the body of $f$ *regardless* of whether it is in a normal form
  - Non-strict, so we can work with divergent terms like any other value, but terms appearing *once* in a program could be evaluated *several times*
Call-by-Need
Call-by-Need

- Combines non-strict semantics of call-by-name with the evaluate-once property of call-by-value
Call-by-Need

- Combines non-strict semantics of call-by-name with the evaluate-once property of call-by-value
  - Achieves this by *sharing* the results of let-bound terms, storing those results in a *heap*
Call-by-Need

- Combines non-strict semantics of call-by-name with the evaluate-once property of call-by-value
  - Achieves this by *sharing* the results of let-bound terms, storing those results in a *heap*
  - Ignoring efficiency, this is equivalent to call-by-name
Call-by-Need

- Combines non-strict semantics of call-by-name with the evaluate-once property of call-by-value
  - Achieves this by *sharing* the results of let-bound terms, storing those results in a *heap*
  - Ignoring efficiency, this is equivalent to call-by-name
  - But if we want to consider costs, we need to reason about impure *heap updates*...
Launchbury’s Semantics

\[ \Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \ (\text{Lam}) \]
Launchbury’s Semantics

\[
\begin{align*}
\Gamma &: \lambda x. e \downarrow \quad \Gamma &: \lambda x. e \quad \text{(Lam)} \\
\Gamma &: e \downarrow \quad \Delta &: \lambda y. e' \quad \Delta &: e'[x/y] \downarrow \quad \Theta &: v \\
\Gamma &: e\ x \downarrow \quad \Theta &: v \quad \text{(App)}
\end{align*}
\]
Launchbury’s Semantics

\[
\begin{align*}
\Gamma : \lambda x. e & \Downarrow \quad \Gamma : \lambda x. e \quad \text{(Lam)} \\
\Gamma : e & \Downarrow \quad \Delta : \lambda y. e' \quad \Delta : e'[x/y] \Downarrow \quad \Theta : v \\
\Gamma : e \, x & \Downarrow \quad \Theta : v \\
\Gamma, x \mapsto e_1 : e_2 & \Downarrow \quad \Delta : v \\
\Gamma : \text{let } x = e_1 \text{ in } e_2 & \Downarrow \quad \Delta : v \\
\end{align*}
\]
Launchbury’s Semantics

\[
\begin{align*}
\Gamma : \lambda x. e & \Downarrow \quad \Gamma : \lambda x. e \quad \text{(Lam)} \\
\Gamma, x \mapsto e_1 : e_2 & \Downarrow \quad \Delta : v \\
\Gamma : \text{let } x = e_1 \text{ in } e_2 & \Downarrow \quad \Delta : v \\
\end{align*}
\]

\[
\begin{align*}
\Gamma : e & \Downarrow \quad \Delta : \lambda y. e' \quad \Delta : e'[x/y] \Downarrow \quad \Theta : v \\
\Gamma : e \ x & \Downarrow \quad \Theta : v \\
\end{align*}
\]

\[
\begin{align*}
\Gamma : e & \Downarrow \quad \Delta : v \\
\Gamma, x \mapsto e : x & \Downarrow \quad \Delta, x \mapsto v : \hat{v} \\
\end{align*}
\]

(Var)
Launchbury’s Semantics

\[ \Gamma : \lambda x. e \Downarrow \quad \Gamma : \lambda x. e \quad \text{(Lam)} \]

\[ \frac{\Gamma : e \Downarrow \quad \Delta : \lambda y. e' \quad \Delta : e'[x/y] \Downarrow \quad \Theta : v}{\Gamma : e \ x \Downarrow \quad \Theta : v} \quad \text{(App)} \]

\[ \frac{\Gamma, x \mapsto e_1 : e_2 \Downarrow \quad \Delta : v}{\Gamma : \text{let } x = e_1 \ \text{in} \ e_2 \Downarrow \quad \Delta : v} \quad \text{(Let)} \]

\[ \frac{\Gamma : e \Downarrow \quad \Delta : v}{\Gamma(x \mapsto e) : x \Downarrow \quad \Delta, x \mapsto v : \hat{v}} \quad \text{(Var)} \]

Note the heap update!
Launchbury’s Semantics

\[\Gamma : \lambda x.e \Downarrow \_ \Gamma : \lambda x.e \quad \text{(Lam)}\]

\[\Gamma : e \Downarrow_c \Delta : \lambda y.e' \quad \Delta : e'[x/y] \Downarrow_d \Theta : v\]

\[\frac{\Delta : v}{\Gamma : e x \Downarrow_{c+d+1} \Theta : v} \quad \text{(App)}\]

\[\frac{\Gamma, x \mapsto e_1 : e_2 \Downarrow_c \Delta : v}{\Gamma : \text{let } x = e_1 \text{ in } e_2 \Downarrow_{c+1} \Delta : v} \quad \text{(Let)}\]

\[\frac{\Gamma : e \Downarrow_c \Delta : v}{\Gamma, x \mapsto e : x \Downarrow_{c+1} \Delta, x \mapsto v : \hat{v}} \quad \text{(Var)}\]
Can we “purify” laziness?
Can we “purify” laziness?

- On the face of it, lazy evaluation is *inherently* impure; the updates are an integral part.
Can we “purify” laziness?

- On the face of it, lazy evaluation is inherently impure; the updates are an integral part.
- But the actual results hide that impurity from us…
Can we “purify” laziness?

- On the face of it, lazy evaluation is inherently impure; the updates are an integral part.
- But the actual results hide that impurity from us…
- …so maybe we can construct a cost-equivalent semantics that “feels purer”?
Should we care?
Should we care?

- Consider common subexpression elimination:

  \[ M[N] \rightarrow \text{let } x = N \text{ in } M[x] \]
Should we care?

- Consider common subexpression elimination:
  - $M[N] \rightarrow \text{let } x = N \text{ in } M[x]$

- Seems obvious that this transformation should never degrade time performance…
Should we care?

- Consider common subexpression elimination:
  - $M[N] \longrightarrow \text{let } x = N \text{ in } M[x]$

- Seems obvious that this transformation should never degrade time performance...

- But it’s actually pretty hard to prove this!
Should we care?

- Consider common subexpression elimination:
  - \( M[N] \rightarrow \text{let } x = N \text{ in } M[x] \)

- Seems obvious that this transformation should never degrade time performance...

- But it’s actually pretty hard to prove this!

(Image from Schmidt-Schauß and Sabel, 2017)
Should we care?
Should we care?

Another problem: if we want a denotational semantics that captures cost information, we have to thread state through everything we do.
Should we care?

- Another problem: if we want a denotational semantics that captures cost information, we have to thread state through everything we do.

- This makes reasoning about our semantics more difficult.
A Visit from St. Nicholas...
A Visit from St. Nicholas...
A Visit from St. Nicholas...

¬ “I see you when you're sleeping! I know when you're awake!

¬ I know which terms you'll evaluate, so do those early, for goodness' sake!”

(Photo by Huhu Uet)
Who Needs Determinism, Anyway?
Who Needs Determinism, Anyway?

- Nondeterministic automata can be thought of as automata that can “guess” the right path to take.
Who Needs Determinism, Anyway?

- Nondeterministic automata can be thought of as automata that can “guess” the right path to take
- By analogy, we can implement our foresight idea with a nondeterministic semantics
Who Needs Determinism, Anyway?

- Nondeterministic automata can be thought of as automata that can “guess” the right path to take

- By analogy, we can implement our foresight idea with a nondeterministic semantics

- In other words, we try both eager evaluation and discarding, and pick the path that leads to the cheapest successful evaluation
Clairvoyant Call-by-Value

\[ \text{Lam} \]
\[ \Gamma : \lambda x . e \Downarrow_0 \Gamma : \lambda x . e \]  

\[ \text{App} \]
\[ \Gamma : e \Downarrow_c \Delta : \lambda y . e' \quad \Delta : e'[x/y] \Downarrow_d \Theta : v \]
\[ \Gamma : e \Downarrow_{c+d+1} \Theta : v \]

\[ \text{Let} \]
\[ \Gamma, x \mapsto e_1 : e_2 \Downarrow_c \Delta : v \]
\[ \Gamma : \text{let } x = e_1 \text{ in } e_2 \Downarrow_{c+1} \Delta : v \]

\[ \text{Var} \]
\[ \Gamma : e \Downarrow_c \Delta : v \]
\[ \Gamma, x \mapsto e : x \Downarrow_{c+1} \Delta, x \mapsto v : \hat{v} \]
Clairvoyant Call-by-Value

\[
\begin{align*}
\Gamma : \lambda x . e \Downarrow_0 & \Gamma : \lambda x . e \quad \text{(Lam)} \\
\Gamma : e \Downarrow_c \Delta : \lambda y . e' & \Delta : e'[x/y] \Downarrow_d \Theta : \nu \\
\Gamma : e \Downarrow_{c+d} \Theta : \nu & \\
\Gamma : e_1 \Downarrow_c \Delta : \nu_1 & \Delta, x \mapsto \nu_1 : e_2 \Downarrow_d \Theta : \nu_2 \\
\Gamma : \text{let } x = e_1 \text{ in } e_2 \Downarrow_{c+d+1} \Theta : \nu_2 & \\
\Gamma : e \Downarrow_c \Delta : \nu & \\
\Gamma, x \mapsto e : x \Downarrow_{c+1} \Delta, x \mapsto \nu : \hat{\nu} & \quad \text{(Var)}
\end{align*}
\]
Clairvoyant Call-by-Value

\[ \Gamma : \lambda x . e \downarrow_{0} \Gamma : \lambda x . e \quad \text{(Lam)} \]

\[ \Gamma : e \downarrow_{c} \Delta : \lambda y . e' \quad \Delta : e'[x/y] \downarrow_{d} \Theta : v \]
\[ \quad \frac{}{\Gamma : e \ x \downarrow_{c+d+1} \Theta : v} \quad \text{(App)} \]

\[ \Gamma : e_1 \downarrow_{c} \Delta : v_1 \quad \Delta, x \mapsto v_1 : e_2 \downarrow_{d} \Theta : v_2 \]
\[ \quad \frac{}{\Gamma : \text{let } x = e_1 \ \text{in } e_2 \downarrow_{c+d+1} \Theta : v_2} \quad \text{(Let)} \]

\[ \Gamma, x \mapsto v : x \downarrow_{1} \Gamma, x \mapsto v : \hat{v} \quad \text{(Var)} \]
Clairvoyant Call-by-Value

\[ \Gamma : \lambda x . e \Downarrow_0 \Gamma : \lambda x . e \quad \text{(Lam)} \]

\[
\begin{align*}
\Gamma & : e \Downarrow_c \Delta : \lambda y . e' & \Delta : e'[x/y] \Downarrow_d \Theta : v \\
\Gamma & : e \ x \Downarrow_{c+d+1} \Theta : v
\end{align*}
\]

\quad \text{(App)}

\[
\begin{align*}
\Gamma & : e_1 \Downarrow_c \Delta : v_1 & \Delta, x \mapsto v_1 : e_2 \Downarrow_d \Theta : v_2 \\
\Gamma & : \text{let } x = e_1 \text{ in } e_2 \Downarrow_{c+d+1} \Theta : v_2
\end{align*}
\]

\quad \text{(Let)}

\[
\begin{align*}
\Gamma, x \mapsto v : x \Downarrow_1 & \Gamma, x \mapsto v : \hat{v} \\
\text{(Var)}
\end{align*}
\]

\[
\begin{align*}
\Gamma, e_2 \Downarrow_c \Delta : v \\
\Gamma & : \text{let } x = e_1 \text{ in } e_2 \Downarrow_{c+1} \Delta : v
\end{align*}
\]

\quad \text{(Skip)}
Clairvoyant Call-by-Value

\[ \Gamma : \lambda x. e \Downarrow_0 \Gamma : \lambda x. e \quad \text{(Lam)} \]

\[ \frac{\Gamma : e \Downarrow_c \Delta : \lambda y. e' \quad \Delta : e'[x/y] \Downarrow_d \Theta : v}{\Gamma : e \ x \Downarrow_{c+d+1} \Theta : v} \quad \text{(App)} \]

\[ \frac{\Gamma : e_1 \Downarrow_c \Delta : v_1 \quad \Delta, x \mapsto v_1 : e_2 \Downarrow_d \Theta : v_2}{\Gamma : \text{let } x = e_1 \ \text{in} \ e_2 \Downarrow_{c+d+1} \Theta : v_2} \quad \text{(Let)} \]

\[ \frac{\Gamma, e_2 \Downarrow_c \Delta : v}{\Gamma : \text{let } x = e_1 \ \text{in} \ e_2 \Downarrow_{c+1} \Delta : v} \quad \text{(Skip)} \]

Ambiguous...
Clairvoyant Call-by-Value

\[ \Gamma : \lambda x. e \Downarrow_0 \quad \Gamma : \lambda x. e \quad \text{(Lam)} \]

\[ \frac{\Gamma : e \Downarrow_c \quad \Delta : \lambda y. e'}{\Gamma : e \ x \Downarrow_{c+d+1} \quad \Theta : v} \quad \text{(App)} \]

\[ \frac{\Gamma : e_1 \Downarrow_c \quad \Delta : v_1 \quad \Delta, x \mapsto v_1 : e_2 \Downarrow_d \quad \Theta : v_2}{\Gamma : \text{let } x = e_1 \ \text{in} \ e_2 \Downarrow_{c+d+1} \quad \Theta : v_2} \quad \text{(Let)} \]

\[ \frac{\Gamma, e_2 \Downarrow_c \quad \Delta : v}{\Gamma : \text{let } x = e_1 \ \text{in} \ e_2 \Downarrow_{c+1} \quad \Delta : v} \quad \text{(Skip)} \]

...but no more updates
Equivalence to Call-by-Need
Equivalence to Call-by-Need

- Given a term $e$ and a heap $\Gamma$ that consists only of values:
Equivalence to Call-by-Need

- Given a term $e$ and a heap $\Gamma$ that consists only of values:
  - If $\Gamma : e \Downarrow_k$ in call-by-need then $\Gamma : e \Downarrow_k$ in clairvoyant call-by-value
Equivalence to Call-by-Need

- Given a term $e$ and a heap $\Gamma$ that consists only of values:
  - If $\Gamma : e \Downarrow_k$ in call-by-need then $\Gamma : e \Downarrow_k$ in clairvoyant call-by-value

  - Proof sketch: Traverse the derivation of $\Gamma : e \Downarrow_k$ replacing the CBN Let rules with CCBV Let or Skip. If the bound variable is used later, splice in the subtree from there as a premise of the CCBV Let; this splicing is safe because the subtree will only use variables in scope at the point of the let binding. Otherwise, just use Skip.
Equivalence to Call-by-Need

Given a term $e$ and a heap $\Gamma$ that consists only of values:

- If $\Gamma : e \downarrow_k$ in call-by-need then $\Gamma : e \downarrow_k$ in clairvoyant call-by-value

  Proof sketch: Traverse the derivation of $\Gamma : e \downarrow_k$ replacing the CBN Let rules with CCBV Let or Skip. If the bound variable is used later, splice in the subtree from there as a premise of the CCBV Let; this splicing is safe because the subtree will only use variables in scope at the point of the let binding. Otherwise, just use Skip.

- If $\Gamma : e \downarrow_k$ in clairvoyant call-by-value then there is some $k' \leq k$ such that $\Gamma : e \downarrow_{k'}$ in call-by-need
Equivalence to Call-by-Need

- Given a term \( e \) and a heap \( \Gamma \) that consists only of values:
  - If \( \Gamma : e \Downarrow_k \) in call-by-need then \( \Gamma : e \Downarrow_k \) in clairvoyant call-by-value
    - Proof sketch: Traverse the derivation of \( \Gamma : e \Downarrow_k \) replacing the CBN Let rules with CCBV Let or Skip. If the bound variable is used later, splice in the subtree from there as a premise of the CCBV Let; this splicing is safe because the subtree will only use variables in scope at the point of the let binding. Otherwise, just use Skip.
  - If \( \Gamma : e \Downarrow_k \) in clairvoyant call-by-value then there is some \( k' \leq k \) such that \( \Gamma : e \Downarrow_{k'} \) in call-by-need
    - Proof sketch: Similar, but in reverse.
Equivalence to Call-by-Need, continued...
Equivalence to Call-by-Need, continued...

- What this means is that the call-by-need cost is the \textit{minimal} clairvoyant call-by-value cost
Equivalence to Call-by-Need, continued…

▪ What this means is that the call-by-need cost is the 
  *minimal* clairvoyant call-by-value cost

▪ As a corollary, we know that \( \forall C.C[e] \downarrow_k \Rightarrow C[e'] \downarrow_{\leq k} \) in call-
  by-need if and only if \( \forall C.C[e] \downarrow_k \Rightarrow C[e'] \downarrow_{\leq k} \) in clairvoyant 
  call-by-value
Equivalence to Call-by-Need, continued...

- What this means is that the call-by-need cost is the _minimal_ clairvoyant call-by-value cost.

- As a corollary, we know that \( \forall C. C[e] \downarrow_k \Rightarrow C[e'] \downarrow_{\leq k} \) in call-by-need if and only if \( \forall C. C[e] \downarrow_k \Rightarrow C[e'] \downarrow_{\leq k} \) in clairvoyant call-by-value.

- They have the same _contextual improvement_ relation.
Common Subexpression Elimination, Revisited
Common Subexpression Elimination, Revisited

Now it is much easier to justify common subexpression elimination:

\[ \text{M}[\text{N}] \longrightarrow \text{let } x = \text{N} \text{ in } \text{M}[x] \]
Common Subexpression Elimination, Revisited

- Now it is much easier to justify common subexpression elimination:
  \[ M[N] \rightarrow \text{let } x = N \text{ in } M[x] \]

- If the context hole is not evaluated, we can just use Skip rule to get into the same position; should cost the same
Common Subexpression Elimination, Revisited

- Now it is much easier to justify common subexpression elimination:
  - \( M[N] \rightarrow \text{let } x = N \text{ in } M[x] \)

- If the context hole is not evaluated, we can just use Skip rule to get into the same position; should cost the same

- Otherwise, the Let rule allows us to share work, replacing potentially multiple evaluations of \( N \) with only one; this can never degrade performance
But Wait! There’s More!
But Wait! There’s More!

- We can use this new operational semantics for clairvoyant call-by-value to give a *denotational* semantics for call-by-need that avoids heap nastiness...
But Wait! There’s More!

- We can use this new operational semantics for clairvoyant call-by-value to give a *denotational* semantics for call-by-need that avoids heap nastiness...

- We interpret terms in the domain $D = (\omega^{op} \times (D \rightarrow D))_\bot$
But Wait! There’s More!

- We can use this new operational semantics for clairvoyant call-by-value to give a denotational semantics for call-by-need that avoids heap nastiness…

- We interpret terms in the domain $D = (\omega^{op} \times (D \rightarrow D))_\perp$

  - $\omega^{op}$ captures costs
But Wait! There’s More!

- We can use this new operational semantics for clairvoyant call-by-value to give a denotational semantics for call-by-need that avoids heap nastiness...

- We interpret terms in the domain $D = (\omega^{op} \times (D \rightarrow D))_\perp$
  
  - $\omega^{op}$ captures costs
  
  - $\perp$ captures nontermination
But Wait! There’s More!

- We can use this new operational semantics for clairvoyant call-by-value to give a denotational semantics for call-by-need that avoids heap nastiness...

- We interpret terms in the domain $D = (\omega^{op} \times (D \rightarrow D))_\bot$
  
  - $\omega^{op}$ captures costs
  
  - $\bot$ captures nontermination

- We interpret let-binding as the join of discarding and eager evaluation
Soundness and Adequacy
Soundness and Adequacy

- We can relate our denotational and operational semantics with *soundness* and *adequacy* theorems.
Soundness and Adequacy

- We can relate our denotational and operational semantics with *soundness* and *adequacy* theorems.
- Trick here is to find the right statements:
Soundness and Adequacy

- We can relate our denotational and operational semantics with soundness and adequacy theorems.

- Trick here is to find the right statements:

  - Soundness: If $\Gamma : e \downarrow_k \Delta : v$ then $\llbracket e \rrbracket_{\llbracket \Gamma \rrbracket} \geq k \triangleright \llbracket v \rrbracket_{\llbracket \Delta \rrbracket}$
Soundness and Adequacy

- We can relate our denotational and operational semantics with *soundness* and *adequacy* theorems.

- Trick here is to find the right statements:
  - **Soundness**: If $\Gamma : e \Downarrow_k \Delta : v$ then $\left\llbracket e \right\rrbracket_{\Gamma} \geq k \gg \left\llbracket v \right\rrbracket_{\Delta}$
  - **Adequacy**: If $\left\llbracket e \right\rrbracket_{\Gamma} = (k, f)$ then $\Gamma : e \Downarrow_{\leq k}$
Compositionality and Abstraction
Compositionality and Abstraction

- We also can show that contexts act monotonically:

\[
[e] \leq [e'] \Rightarrow [C[e]] \leq [C[e']] 
\]
Compositionality and Abstraction

- We also can show that contexts act monotonically:
  - $\left[ e \right] \leq \left[ e' \right] \Rightarrow \left[ C[e] \right] \leq \left[ C[e'] \right]$

- As a corollary of all this, we get that our model is sound for contextual improvement:
  - $\left[ e \right] \leq \left[ e' \right] \Rightarrow \forall C. C[e] \downarrow_k \Rightarrow C[e'] \downarrow_{\leq k}$
Conclusion
Conclusion

- We don’t need updates to reason about sharing
Conclusion

- We don’t need updates to reason about sharing
- By constructing an update-free operational semantics, we get a cleaner theory that remains faithful to efficiency
Conclusion

- We don’t need updates to reason about sharing
- By constructing an update-free operational semantics, we get a cleaner theory that remains faithful to efficiency
- This also leads to a nicer denotational semantics
So, that’s good for time...
So, that’s good for time…

Father Christmas moonlights as old Father Time...
...but what about space?
...but what about space?

Enough room to fit?
...but what about space?

- Not clear how to capture space usage in the same way; are heaps essential here?

Enough room to fit?
Questions?