Continuations and Monads

Kenneth Lai

Brandeis University

October 31, 2022

Continuations and Monads

Kenneth Lai

Brandeis University

October 31, 2022

Source

Continuations and Monads

Kenneth Lai

Brandeis University

October 31, 2022

Source

Burritos

- Monads are like burritos

Burritos

- Monads are like burritos
- Monads are not like burritos

Announcements

- For Wednesday
  - Read Lipovača Chapter 12
  - HW3 due
- For 11/9
  - Final Project Idea due

Today's Plan

- Final Project Idea: Counterfactual Model Generation
- Continuations
- Monads

Today's Plan

- Final Project Idea: Counterfactual Model Generation
- Continuations
- Monads
- (we'll see how far we get...)

Counterfactual Model Generation

- "If kangaroos had no tails, [then] they would topple over."
- Given a counterfactual sentence, generate a minimal model that represents the counterfactual
  - Set of worlds, accessibility relation
    - Easier: Worlds contain propositions
    - Harder: Worlds contain entities and relations

Counterfactual Model Generation

- Some things you could/should do
  - Modify the (book's) parser to accept the counterfactual syntax
    - Look at `P.hs`
  - Identify what is being presupposed about the actual world, and what is asserted about the counterfactual world, and fill in the worlds appropriately

Counterfactual Model Generation

- Some things to think about
  - Interaction with tense and aspect
    - “If John had gone to New York, Mary would have seen him.”
  - What does the accessibility relation “mean”?
    - Actions by one or more agents
    - Interventions to make certain facts true

  - These will affect how many worlds you generate and how they are related

Counterfactual Model Generation

- Some things to look at
  - van Eijck and Unger Chapter 9
  - Peter Menzies and Helen Beebee, “Counterfactual Theories of Causation”, The Stanford Encyclopedia of Philosophy (Winter 2020 Edition), Edward N. Zalta (ed.)

Intensions

When we want to determine the reference of an expression, we have to consider the context, i.e. reference is not absolute anymore but depends on the context (time, possible worlds, anaphoric potential,. . .).

Two Kinds of Context

- The meaning of an expression is a function of its context

Two Kinds of Context

- The meaning of an expression is a function of its context
  - Extralinguistic context
    - Reference depends on time, possible worlds, etc.
    - “Alonzo greeted the queen of the Netherlands.”

Two Kinds of Context

- The meaning of an expression is a function of its context
  - Extralinguistic context
    - Reference depends on time, possible worlds, etc.
    - "Alonzo greeted the queen of the Netherlands."
  - Linguistic context
    - Reference depends on the rest of the sentence, discourse, etc.
    - "John pushed Mary. She fell."

Continuations

- Suppose we are computing the meaning of “Alice helped Dorothy”
  - Consider the linguistic context of “Alice” to be the rest of the sentence, “_____ helped Dorothy”
    - $\lambda x.\text{Help}(x, d)$

Continuations

- Suppose we are computing the meaning of “Alice helped Dorothy”
  - Consider the linguistic context of “Alice” to be the rest of the sentence, “_____ helped Dorothy”
    - $\lambda x.\mathrm{Help}(x, d)$
  - “During the meaning computation this slot will be filled with the individual constant corresponding to Alice.”
    - $(\lambda x.\mathrm{Help}(x, d))(a)$

Continuations

- Suppose we are computing the meaning of “Alice helped Dorothy”
  - Consider the linguistic context of “Alice” to be the rest of the sentence, “_____ helped Dorothy”
    - $\lambda x.\text{Help}(x, d)$
  - “During the meaning computation this slot will be filled with the individual constant corresponding to Alice.”
    - $(\lambda x.\text{Help}(x, d))(a)$
  - We can abstract over this context to get a function
    - $\lambda P.P(a)$

Continuations

- This is just the meaning of “Alice” as a generalized quantifier!
  - “Alice” denotes a function that takes a predicate and hands it the argument a

Continuations

- This is just the meaning of “Alice” as a generalized quantifier!
  - “Alice” denotes a function that takes a predicate and hands it the argument *a*
- “The treatment of NP denotations as abstractions over NP contexts was devised by Montague in order to give a unified treatment of quantificational and non-quantificational NPs. But we do not need to restrict this strategy to NPs; we can extend it to expressions of other categories as well.”

Continuations

- Consider the linguistic context of “helped” to be the rest of the sentence, “Alice ______ Dorothy”
  - $\lambda P.P(d)(a)$
    - We will write d before a, because transitive verbs combine with their direct objects before their subjects

Continuations

- Consider the linguistic context of “helped” to be the rest of the sentence, “Alice ______ Dorothy”
  - $\lambda P.P(d)(a)$
    - We will write d before a, because transitive verbs combine with their direct objects before their subjects
- During the meaning computation this slot will be filled with the predicate corresponding to “helped”.
  - $(\lambda P.P(d)(a))(\lambda y \lambda x.\text{Help}(x, y))$, or $(\lambda P.P(d)(a))(\text{Help})$
    - $\text{Help}(d)(a) = \text{Help}(a, d)$

Continuations

- Consider the linguistic context of “helped” to be the rest of the sentence, “Alice ______ Dorothy”
  - $\lambda P.P(d)(a)$
    - We will write d before a, because transitive verbs combine with their direct objects before their subjects
- During the meaning computation this slot will be filled with the predicate corresponding to “helped”.
  - $(\lambda P.P(d)(a))(\lambda y \lambda x.\text{Help}(x,y))$, or $(\lambda P.P(d)(a))(\text{Help})$
    - $\text{Help}(d)(a) = \text{Help}(a,d)$
- We can abstract over this context to get a function
  - $\lambda \mathcal{P}.\mathcal{P}(\text{Help})$

Continuations

- **Exercise 11.1** In a similar manner, we can look at the meaning of *helped Dorothy*. Its linguistic context is *Alice* ______ _______. Specify the denotation of *helped Dorothy* and its type along the above lines, and do the same also for *Alice helped*.

Continuations

- "We will call the meaning of the linguistic context of an expression its continuation
  - The notion stems from computer science. There continuations are used for giving a compositional semantics of programs that exhibit apparent non-compositional side effects like jumps and exceptions.
  - It serves the same purpose we used it for: it captures the context of an expression and provides it to the computation as a function.

Continuations

- "We will call the meaning of the linguistic context of an expression its continuation
  - The notion stems from computer science. There continuations are used for giving a compositional semantics of programs that exhibit apparent non-compositional side effects like jumps and exceptions.
  - It serves the same purpose we used it for: it captures the context of an expression and provides it to the computation as a function.
- The general type of the continuation of an expression of type τ is a function of type $\tau \rightarrow r$, with r being the type of result values."

Continuations

- “We will call the meaning of the linguistic context of an expression its continuation
  - The notion stems from computer science. There continuations are used for giving a compositional semantics of programs that exhibit apparent non-compositional side effects like jumps and exceptions.
  - It serves the same purpose we used it for: it captures the context of an expression and provides it to the computation as a function.
- The general type of the continuation of an expression of type τ is a function of type $\tau \to r$, with r being the type of result values.”
- If the linguistic context is the rest of the sentence (or clause, “next enclosing sentence”), then the result values are sentence denotations
  - $r = t$

Continuations

- “We specified the denotation of an expression as a function from the continuation of that expression to the result type t. We will call such functions from continuations to result values computations.”
  - The meaning of “Alice” is the computation of “Alice”, $\lambda P.P(a)$

Continuations

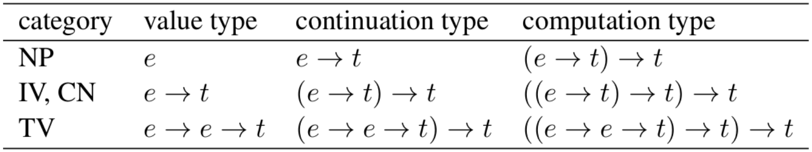
- "We specified the denotation of an expression as a function from the continuation of that expression to the result type t. We will call such functions from continuations to result values computations."
    - The meaning of "Alice" is the computation of "Alice", $\lambda P.P(a)$
    - "This computation tells us what role the sub-expression *Alice* plays in the whole expression, i.e. what is going to be done with it when computing the meaning of the whole expression (in this case it is plugged into a VP-meaning).
    - Under this view, a continuation is an instruction of what to do next."

Continuations

- In general, if α is the type of a value, then $\alpha \to t$ is the type of its continuation, and $(\alpha \to t) \to t$ is the type of its computation

Continuations

- In general, if α is the type of a value, then $\alpha \to t$ is the type of its continuation, and $(\alpha \to t) \to t$ is the type of its computation



category	value type	continuation type	computation type
NP	e	$e \to t$	$(e \to t) \to t$
IV, CN	$e \to t$	$(e \to t) \to t$	$((e \to t) \to t) \to t$
TV	$e \to e \to t$	$(e \to e \to t) \to t$	$((e \to e \to t) \to t) \to t$

Continuations

- “Now we will turn to systematically lifting the denotation of expressions from the type they had in Chapter 7 to the type of a computation.
  - The resulting semantics is called continuation passing style semantics, because during the whole meaning computation we will keep passing continuations around.
  - The lifting is called continuation passing style transformation, or CPS transformation for short.”

Continuations

- "Now we will turn to systematically lifting the denotation of expressions from the type they had in Chapter 7 to the type of a computation.
  - The resulting semantics is called continuation passing style semantics, because during the whole meaning computation we will keep passing continuations around.
  - The lifting is called continuation passing style transformation, or CPS transformation for short."
- Like with intensionalization, we can express the CPS transformation process in terms of functional programming tools, in this case, monads

Monads

```
class (Applicative M) = > Monad M where
    return :: a -> M a

    (>>=) :: M a -> (a -> M b) -> M b

    (>>) :: M a -> M b -> M b
    x >> y = x >>= \_ -> y

    fail :: String -> M a
    fail msg = error msg
```

Monads

```
class (Applicative M) = > Monad M where
    return :: a -> M a

    (>>=) :: M a -> (a -> M b) -> M b

    (>>) :: M a -> M b -> M b
    x >> y = x >>= \_ -> y

    fail :: String -> M a
    fail msg = error msg
```

- `return` is just like `pure` for applicative functors

Monads

- To understand (>>=) (pronounced bind), it may help to think in terms of its flipped version, (=<<)

```
(>>=) :: M a -> (a -> M b) -> M b
(=<<) = flip (>>=)
```

Monads

- To understand (>>=) (pronounced bind), it may help to think in terms of its flipped version, (=<<)

```
(>>=) :: M a -> (a -> M b) -> M b
(=<<) = flip (>>=)
```

- Let us compare (=<<) with some other functions

```
(=<<) ::   (a -> M b) -> M a -> M b
(<*>) :: F (a ->   b) -> F a -> F b
fmap  ::   (a ->   b) -> F a -> F b
```

Monads

- To understand (>>=) (pronounced bind), it may help to think in terms of its flipped version, (=<<)

```
(>>=) :: M a -> (a -> M b) -> M b
(=<<) = flip (>>=)
```

- Let us compare (=<<) with some other functions

```
(=<<) ::   (a -> M b) -> M a -> M b
(<*>) :: F (a ->   b) -> F a -> F b
fmap  ::   (a ->   b) -> F a -> F b
```

- (=<<) (and (>>=)) are maps for monadic functions
  - Functions that create their own boxes

Monads

- To understand (>>=) (pronounced bind), it may help to think in terms of its flipped version, (=<<)

```
(>>=) :: M a -> (a -> M b) -> M b
(=<<) = flip (>>=)
```

- Let us compare (=<<) with some other functions

```
(=<<) ::   (a -> M b) -> M a -> M b
(<*>) :: F (a ->   b) -> F a -> F b
fmap  ::   (a ->   b) -> F a -> F b
```

- (=<<) (and (>>=)) are maps for monadic functions
  - Functions that create their own context