Computational Semantics, Type Theory, and Functional Programming

Jan van Eijck

CWI and ILLC, Amsterdam, Uil-OTS, Utrecht

LOLA7 Tutorial, Pecs

August 2002

In this tutorial we link computational semantics with polymorphic type theory and functional programming.

An emerging standard for polymorphically typed, lazy, purely functional programming is Haskell, a language named after Haskell Curry. Haskell is based on (polymorphically typed) lambda calculus, which makes it an excellent tool for computational semantics.

In this tutorial we link computational semantics with polymorphic type theory and functional programming.

An emerging standard for polymorphically typed, lazy, purely functional programming is Haskell, a language named after Haskell Curry. Haskell is based on (polymorphically typed) lambda calculus, which makes it an excellent tool for computational semantics.

Haskell is easy. In fact, you will learn Haskell programming in the course of this tutorial.

In this tutorial we link computational semantics with polymorphic type theory and functional programming.

An emerging standard for polymorphically typed, lazy, purely functional programming is Haskell, a language named after Haskell Curry. Haskell is based on (polymorphically typed) lambda calculus, which makes it an excellent tool for computational semantics.

Haskell is easy. In fact, you will learn Haskell programming in the course of this tutorial.

Haskell is free: grab it from *www.haskell.org*

In the first lecture we give a brief introduction to polymorphic type theory, and we show how Montague Grammarians can be converted into programmers in one easy step.

In the first lecture we give a brief introduction to polymorphic type theory, and we show how Montague Grammarians can be converted into programmers in one easy step.

In the second lecture, we look at the extension of dynamic predicate logic to type theory. This gives rise to a system that is very much like modern compositional versions of discourse representation theory. We present an implementation and comment on its failings.

In the first lecture we give a brief introduction to polymorphic type theory, and we show how Montague Grammarians can be converted into programmers in one easy step.

In the second lecture, we look at the extension of dynamic predicate logic to type theory. This gives rise to a system that is very much like modern compositional versions of discourse representation theory. We present an implementation and comment on its failings.

In the third lecture, we focus on those shortcomings of dynamic Montague grammar or compositional DRT that have to do with the use of dynamic variable binding and destructive assignment. We show how these can be overcome, and we look at issues of updating salience in context and pronoun reference resolution.