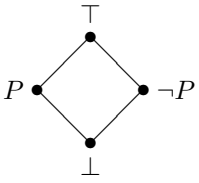
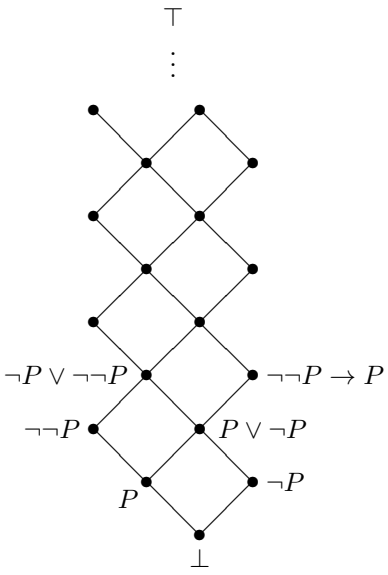
1 A Digression on Heyting Algebra

As discussed last time, there are fewer formulas that are considered intuitionistically valid than classically valid. The law of double negation ($\neg\neg\varphi \to \varphi$), the law of excluded middle ($\varphi \vee \neg\varphi$), and proof by contradiction or reductio ad absurdum are no longer accepted.

Boolean algebra is to classical logic as *Heyting algebra* is to intuitionistic logic. A Heyting algebra is an algebraic structure of the same signature as Boolean algebra, but satisfying only those equations that are provable intuitionistically. Whereas the free Boolean algebra on n generators has 2^{2^n} elements, the free Heyting algebra on one generator has infinitely many elements.



Free Boolean algebra on one generator



Free Heyting algebra on one generator

The picture on the right is sometimes called the *Rieger–Nishimura ladder*.

2 Uninhabited Types

Since the proposition $\bot$ is not provable, it follows that if it corresponds to a type 0, that type must be uninhabited: there is no term with that type. Of course, $\bot$ is not the only uninhabited type; for example, the type $\forall\alpha\,.\,\alpha$ also corresponds to logical falsity and is uninhabited as well.

Note that we can produce terms with these types if we have recursive functions, as in the following term with type 0:

$$(\mathsf{rec}\ f : \mathsf{int} \to 0.\lambda x : int.\, f(x))\, 42$$

However, the typing rule for recursive functions corresponds to a logic rule that makes the logic inconsistent: it assumes what it wants to prove!

$$\frac{\Gamma,\, y : \tau \to \tau',\, x : \tau \vdash e : \tau'}{\Gamma \vdash (\mathsf{rec}\ y : \tau \to \tau'.\lambda x : \tau\,.\,e) : \tau \to \tau'} \qquad \frac{\Gamma,\, \varphi \Rightarrow \varphi',\, \varphi \vdash \varphi'}{\Gamma \vdash \varphi \Rightarrow \varphi'}$$

Thus, we can think of 0 as the type of a term that does not actually return to its surrounding context.

3 Continuations and Negation

What is the significance of negation? We know that logically $\neg\varphi$ is equivalent to $\varphi \Rightarrow \bot$, which suggests that we can think of $\neg\varphi$ as corresponding to a function $\tau \to 0$. We have seen functions that accept a type and do not return a value before: continuations have that behavior. If φ corresponds to τ, a reasonable interpretation of $\neg\varphi$ is as a continuation expecting a τ. Negation corresponds to turning outputs into inputs.

As we saw above with currying and uncurrying, meaning-preserving program transformations can have interesting logical interpretations. What about conversion to continuation-passing style? We represent a continuation k expecting a value of type τ as a function with type $\tau \to 0$.

We can then define CPS conversion as a type-preserving translation $[\![\Gamma \vdash e : \tau]\!]$. Here we include the entire type derivation $\Gamma \vdash e : \tau$ inside the $[\![\cdot]\!]$ because types are not unique and the translation depends on the typing. The translation is type-preserving in the sense that a well-typed source term ($\Gamma \vdash e : \tau$) translates to a well-typed target term:

$$\mathcal{G}[\![\Gamma]\!] \vdash [\![\Gamma \vdash e : \tau]\!] : \mathcal{T}[\![\tau]\!].$$

The translation of the typing context $\mathcal{G}[\![\Gamma]\!]$ simply translates all the contained variables:

$$\mathcal{G}[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] = x_1 : \mathcal{T}[\![\tau_1]\!], \ldots, x_n : \mathcal{T}[\![\tau_n]\!].$$

The soundness of the translation can be seen by induction on the typing derivation.

$$\begin{array}{rcl}
[\![\Gamma,\, x : \tau \vdash x : \tau]\!] & = & \lambda k : \mathcal{T}[\![\tau]\!] \to 0.\, k\, x \\
[\![\Gamma \vdash (\lambda x : \tau.\, e) : \tau \to \tau']\!] & = & \lambda k : \mathcal{T}[\![\tau \to \tau']\!] \to 0.\, k\, (\lambda k' : \mathcal{T}[\![\tau']\!] \to 0.\, \lambda x : \mathcal{T}[\![\tau]\!].\, [\![\Gamma,\, x : \tau \vdash e : \tau']\!] k') \\
[\![\Gamma \vdash (e_0\, e_1) : \tau']\!] & = & \lambda k : \mathcal{T}[\![\tau']\!] \to 0.\, [\![\Gamma \vdash e_0 : \tau \to \tau']\!]\, (\lambda f : \mathcal{T}[\![\tau \to \tau']\!].\, [\![\Gamma \vdash e_1 : \tau]\!]\, (\lambda v : \mathcal{T}[\![\tau]\!].\, f\, k\, v))
\end{array}$$

To make this type-check, we define the type translation $\mathcal{T}[\![\cdot]\!]$ as follows:

$$\begin{array}{rcl}
\mathcal{T}[\![B]\!] & = & B \\
\mathcal{T}[\![\tau \to \tau']\!] & = & (\mathcal{T}[\![\tau']\!] \to 0) \to (\mathcal{T}[\![\tau]\!] \to 0)
\end{array}$$

Note that the logical interpretation of the translation of a function type corresponds to the use of the contrapositive: $(\varphi \Rightarrow \psi) \;\Rightarrow\; (\neg\psi \Rightarrow \neg\varphi)$.

By induction on the typing derivation, we can see that CPS conversion converts a source term of type τ into a target term of type $(\mathcal{T}[\![\tau]\!] \to 0) \to 0$. Since programs correspond to proofs, CPS conversion shows how to convert a proof of proposition φ into a proof of proposition $\neg\neg\varphi$. In other words, CPS conversion proves the admissibility in constructive logic of the rule for introducing double negation:

$$\frac{\varphi}{\neg\neg\varphi}$$

However, we are unable to invert CPS translation, and similarly we are unable (constructively) to *remove* double negation.

4 Extracting Computational Content

Many automated deduction systems, such as NuPrl and Coq, are based on constructive logic. Automatic programming was a significant research direction that motivated the development of these systems. The idea

was that a constructive proof of the existence of a function would automatically yield a program to compute it: the statement asserting the existence of the function is a type, and a constructive proof yields a λ-term inhabiting that type. For example, to obtain a program computing square roots, one merely has to give a constructive proof of the statement $\forall x \geq 0\ \exists y\ y^2 = x$.

5 Other Directions

If second-order constructive predicate logic corresponds to System F, do other logics correspond to new kinds of programming language features? This has been an avenue of fruitful exploration over the last couple of decades, with programming-language researchers deriving insights from classical logic, higher-order, and linear logics that help guide the design of useful language features.

For example, *linear logic* is a logic that keeps track of resources. One may only use an assumption in the application of a rule once; the assumption is consumed and may not be reused. This corresponds to functions that consume their arguments, and hence is a possible model for systems with bounded resources.