Abstract

We will look at the simple programming language used to process evidence for constructive and intuitionistic propositional logic (iPC). We can think of this as a programming language for thought. As we move on to more expressive logics, we will enrich this programming language to include additional operations on evidence.

The programming language for propositional logic allows us to experience the meaning of logical expressions by computing specific examples of how evidence is transformed according to the computational meaning of the logical expression.

As we enrich the logic, we must at some point give up on expecting to decide whether a proposition is provable or not. Provability is already not decidable for first-order logic, a remarkable result discovered by Alonzo Church [1] – putting American logic on the map and eventually computer science as well.\(^1\)

1 A programming language for constructive propositional logic

The salient feature of constructive propositional logic, also called the intuitionistic propositional logic, iPC, is that proofs of implications can be treated as functional programs and proofs of conjunctions and disjunctions create data for these programs. For example, a proof of the proposition \((A \& B) \Rightarrow A\) is a computable function that takes an ordered pair pair\((a; b)\) where \(a\) is evidence for \(A\) and \(b\) is evidence for \(B\) into evidence for \(A\). This is a very simple function that decomposes the pair and outputs the first component. In this section we show in detail how to do this. It requires that we provide a specific syntax for

---

\(^1\)Alonzo Church advised 35 PhD students in logic, many of whom are major contributors to the field including Alan Turing and Stephen Kleene. His students in turn graduated many PhD students. Church now has 4,783 descendants.
functions, pairs, disjunctions, and the empty type. We use the empty type to represent \textit{False}, a proposition with no evidence. Here is another example we will study.

\[(A \& B) \Rightarrow G \Rightarrow (\neg G \Rightarrow \neg (A \& B))\]

A proof of this proposition serves as a program that converts a function \((A \& B) \Rightarrow G\) into a function \((\neg G \Rightarrow \neg (A \& B))\). This kind of program is called a \textit{functional}, but it does not require special syntax.

\section{The propositional programming language}

Evidence semantics for propositional logic is built from the computational meaning of the logical primitives, namely \textit{ordered pairs} for conjunction \((\land \text{ or } \&)\), \textit{disjoint unions} for disjunctions \((\lor \text{ or } \vee)\), and \textit{computable functions} for implications \((\Rightarrow)\). For negation we need the \textit{empty type} to provide the meaning of the logical constant \textit{False}, sometimes written as \(\bot\). The negation of proposition \(A\) is defined as \(A \Rightarrow \bot\). The constant proposition \textit{True}, also called \textit{top} denoted \(\top\), can be interpreted as a type with exactly one element. The \textit{Unit} type also has this property, with \(*\) as its only member. It is convenient to simply use \textit{Unit} for \textit{True}.

```
Evidence semantics for intuitionistic propositional logic requires understanding \textit{ordered pairs}, \textit{pair}(a; b), also written \(<a, b>\). We an operator called \textit{spread} to decompose these pairs, \textit{spread}(p; a, b.exp(a, b)). Using spread, we can easily define simple operations such as \textit{first}(\textit{pair}(a; b)); this is \textit{spread}(p; a, b.a) which reduces to \(a\) and \textit{second}(\textit{pair}(a; b)) as \textit{spread}(p; a, b.b) which reduces to \(b\). The term \textit{pair}(a; b) is the \textit{canonical form} for conjunction.
```

```
The \textit{disjoint union} \(A \lor B\) corresponds to a logical \textit{disjunction}, as in “\(A\text{ or }B\).” The canonical elements of this type are either \textit{inl}(a) where \(a\) is evidence for \(A\) or \textit{inr}(b) where \(b\) is evidence for \(B\). The tags \textit{inl} and \textit{inr} specify which proposition the evidence supports, e.g. \textit{inr} designates the right hand disjunct, and \textit{inl} the left one. The operation for processing this evidence is \textit{decide}(d; l.left(l); r.right(r)). This is a computable operation as we discussed in \textit{Lecture 7}. The operation “\textit{decide}” is like an “\textit{if then else}” construct in natural language.
```

```
Understanding \textit{implication} as in \(A \Rightarrow B\) requires understanding computable functions. The proposition \(A \Rightarrow B\) is a \textit{constructive implication}. It conveys a computational understanding of implication and is quite intuitive. To know \(A \Rightarrow B\) we need a \textit{computable} method to transform evidence \(a\) for \(A\) into evidence for \(B\). We write such a function using lambda notion from Church \([2]\), \(\lambda(x.b(x))\). We are familiar with using functions for such tasks; they transform inputs into outputs. So the evidence for \(A \Rightarrow B\) is a \textit{computable function}. For
```
example, $A \Rightarrow A$ is the type of the identity function $\lambda(x.x)$.

In Lecture 7 we also mentioned notations for functions. In functional programming languages, it is common to follow Church and use some variant of the \textit{lambda calculus}. One version of Church’s notation is $\lambda(x.b(x))$ where the variable $x$ is called the \textit{bound variable}, and $b(x)$ is the \textit{body} of the function. The OCaml programming language writes functions as \texttt{fun x → b(x)}. The bound variable $x$ names the input, and the body $b(x)$ is an expression for computing the output value. Here is a concrete example where the input is an ordered pair, $\lambda(x.first(x))$.

Application of functions is expressed as $ap(f;a)$. The \textit{computation rule} (or reduction rule) is $ap(\lambda(x.exp(x));a)$ reduces to $exp(a)$ which may reduce further depending on $exp$. In the simplest case $ap(\lambda(x.x);a)$ reduces to $a$.

### 1.2 Properties of the propositional programming language

In a typical programming language it is possible to define functions that do not terminate. This is done using recursion. A recursive function can be defined like this: let $f(n) = \text{if } n = 0 \text{ then } 1 \text{ else } (n \text{ times } f(n-1))$. We do not need such functions to compute evidence for logical propositions. This example terminates, but $(\text{let } f(x) = f(n+1))$ does not.

Some expressions in our propositional programming language do not reduce to anything simpler, for example $\lambda(x.x)$. These expressions are called \textit{canonical}. Another example is $\text{pair}(\lambda(x.x);\star)$. What proposition could this pair prove? One answer is:

$$(\text{Unit} \Rightarrow \text{Unit}) \& \text{Unit}.$$ 

Is there another answer?

### 2 Tableaux for Intuitionistic Propositional Calculus and Evidence Semantics

In his book \textit{Intuitionistic Logic Model Theory and Forcing} [3], Fitting gives tableaux style rules for iPC and proves that they are complete for this logic. His proof is not constructive, but it justifies extending Smullyan’s tableaux methods to the iPC, and the proofs are a basis for showing how to find computational evidence for provable propositions.
Here are his tableaux rules.

\[
\begin{align*}
\text{True } \& & S, T(X \& Y) & \quad & \text{False } \& & S, F(X \& Y) \\
& & S, TX, TY & & S, FX \mid S, FY \\
\text{True } \vee & & S, T(X \vee Y) & & \text{False } \vee & & S, (FX \vee Y) \\
& & S, TX \mid S, TY & & S, FX, FY \\
\text{True } \neg & & T \neg X & & \text{False } \neg & & F(\neg X) \\
& & S, FX & & S \mid TX \\
\text{True } \Rightarrow & & S, X \Rightarrow Y & & \text{False } \Rightarrow & & S, F(X \Rightarrow Y) \\
& & S, FX \mid S, TY & & S \mid TX, FY
\end{align*}
\]

References

