massachusetts institute of technology — artificial intelligence laboratory

Type-alpha DPLs

Konstantine Arkoudas

AI Memo 2001-025 October 2001

Abstract

This paper introduces *Denotational Proof Languages* (DPLs). DPLs are languages for presenting, discovering, and checking formal proofs. In particular, in this paper we discus *type-*α DPLs—a simple class of DPLs for which termination is guaranteed and proof checking can be performed in time linear in the size of the proof. Type-α DPLs allow for lucid proof presentation and for efficient proof checking, but not for proof search. Type-ω DPLs allow for search as well as simple presentation and checking, but termination is no longer guaranteed and proof checking may diverge. We do not study type-ω DPLs here.

We start by listing some common characteristics of DPLs. We then illustrate with a particularly simple example: a toy type-α DPL called $\mathcal{PAR}$, for deducing parities. We present the abstract syntax of $\mathcal{PAR}$, followed by two different kinds of formal semantics: evaluation and denotational. We then relate the two semantics and show how proof checking becomes tantamount to evaluation. We proceed to develop the proof theory of $\mathcal{PAR}$, formulating and studying certain key notions such as observational equivalence that pervade all DPLs.

We then present $\mathcal{NDL}_0$, a type-α DPL for classical zero-order natural deduction. Our presentation of $\mathcal{NDL}_0$ mirrors that of $\mathcal{PAR}$, showing how every basic concept that was introduced in $\mathcal{PAR}$ resurfaces in $\mathcal{NDL}_0$. We present sample proofs of several well-known tautologies of propositional logic that demonstrate our thesis that DPL proofs are readable, writable, and concise. Next we contrast DPLs to typed logics based on the Curry-Howard isomorphism, and discuss the distinction between pure and augmented DPLs. Finally we consider the issue of implementing DPLs, presenting an implementation of $\mathcal{PAR}$ in SML and one in Athena, and end with some concluding remarks.

1.1 Introduction

DPLs are languages for presenting, discovering, and checking formal proofs [2]. In particular, in this paper we discuss *type-*α DPLs—a simple class of DPLs for which termination is guaranteed and proof checking can be performed in time linear in the size of the proof on average. Type-α DPLs allow for lucid proof presentation and efficient proof checking, but not for proof search. Type-ω DPLs allow for search as well as simple presentation and checking, but termination is no longer guaranteed and proof checking may diverge. Type-ω DPLs will be discussed elsewhere.

Technically, a DPL can be viewed as a $\lambda\phi$ system [2], and the distinction between type-α and type-ω DPLs can be made formally precise in that setting. The $\lambda\phi$-calculus is an abstract calculus that can be viewed as a general unifying framework for DPLs, much as the λ-calculus can be viewed as a general unifying framework for functional programming languages.[1] However, in this paper we will be more concrete; we will attempt to convey the main intuitions behind type-α DPLs informally, by way of examples. Most of our results will be stated without proof. The omitted proofs are usually straightforward structural inductions; those that are more involved can be found elsewhere [2].

We begin by listing some characteristics that are common to all DPLs, both type-α and type-ω. These features will be illustrated shortly in our two sample DPLs.

- A DPL has at least two syntactic categories: *propositions* P and *proofs* D. Proofs are intended to establish propositions. Both are generated by domain-specific grammars:

$$\begin{aligned} P &::= \cdots \\ D &::= \cdots \end{aligned}$$

  By "domain-specific" we mean that the exact specification of the above grammars may vary from DPL to DPL.

[1] The $\lambda\phi$-calculus originally appeared under the name "$\lambda\mu$-calculus" in "Denotational Proof Languages" [2]; the μ operator was subsequently changed to ϕ to avoid confusion with Parigot's $\lambda\mu$-calculus [9].

- Proofs denote propositions. More operationally: proofs produce (or *derive*) propositions.
- Proofs are *evaluated*. Evaluating a proof D will either produce a proposition P, which is viewed as the *conclusion* of D, or else it will fail, indicating that the proof is unsound. Thus the slogan

$$\textit{Evaluation} = \textit{Proof checking}.$$

- A proof is evaluated with respect to a given *assumption base*. An assumption base is a set of propositions that we take for granted (as "premises") for the purposes of a proof. A proof may succeed with respect to some assumption bases and fail with respect to others.
- A DPL always provides a composition (or "cut") mechanism that allows us to join two proofs so that the conclusion of the first proof becomes available as a lemma inside the second proof.

DPLs have been formulated for propositional logic, for first-order predicate logic with equality, for polymorphic multi-sorted first-order logic, for higher-order logic, for sequent logics, unification logics, equational logics, Hindley-Milner type systems, Hoare-Floyd logics, and for several intuitionist, modal, and temporal logics. Many of these have been implemented, and implementations are freely available.

Finally, a note on terminology: some authors draw a distinction between "proof" and "deduction", often using "proof" to refer to a deduction from the empty set of premises. In this paper we will use the two terms interchangeably.

1.2 A simple example

We will now formulate $\mathcal{PAR}$, a simple type-α DPL for inferring the parity of a natural number.

Abstract syntax

The propositions of $\mathcal{PAR}$ are generated by the following abstract grammar:

$$P ::= \mathit{Even}(n) \mid \mathit{Odd}(n)$$

where n ranges over the natural numbers. Thus $\mathit{Odd}(0), \mathit{Even}(24)$, and $\mathit{Even}(103)$ are sample propositions.

Deductions are generated by the following simple grammar:

$$D ::= \textit{Prim-Rule}\ P_1, \ldots, P_n \mid D_1; D_2 \tag{1.1}$$

where

$$\textit{Prim-Rule} ::= \textbf{zero-even} \mid \textbf{even-next} \mid \textbf{odd-next}.$$

The rule **zero-even** is nullary (takes no arguments), while **even-next** and **odd-next** are unary. We will write $\textbf{Prop}[\mathcal{PAR}]$ and $\textbf{Ded}[\mathcal{PAR}]$ for the sets of all propositions and deductions of $\mathcal{PAR}$, respectively. When it is clear that we are talking about $\mathcal{PAR}$, we will simply write **Prop** and **Ded**.

A deduction of the form $\textit{Prim-Rule}\ P_1, \ldots, P_n$, $n \geq 0$, is called a *primitive rule application*, or simply a "rule application".[2] We say that $P_1, \ldots, P_n$ are the *arguments* supplied to *Prim-Rule*. A deduction of the form $D_1; D_2$ is called a *composition*.

Compositions are *complex proofs*, meaning that they are recursively composed from smaller component proofs. Rule applications, on the other hand, are *atomic* proofs; they have no internal structure.

[2] In type-ω DPLs the term "method" is used instead of "rule".

$$\frac{}{\beta \vdash \textbf{zero-even} \rightsquigarrow \mathit{Even}(0)}\ [\mathit{ZeroEven}]$$

$$\frac{}{\beta \cup \{\mathit{Odd}(n)\} \vdash \textbf{even-next}\ \mathit{Odd}(n) \rightsquigarrow \mathit{Even}(n+1)}\ [\mathit{EvenNext}]$$

$$\frac{}{\beta \cup \{\mathit{Even}(n)\} \vdash \textbf{odd-next}\ \mathit{Even}(n) \rightsquigarrow \mathit{Odd}(n+1)}\ [\mathit{OddNext}]$$

$$\frac{\beta \vdash D_1 \rightsquigarrow P_1 \qquad \beta \cup \{P_1\} \vdash D_2 \rightsquigarrow P_2}{\beta \vdash D_1; D_2 \rightsquigarrow P_2}\ [\mathit{Comp}]$$

Figure 1.1: Formal evaluation semantics of $\mathcal{PAR}$.

This distinction is reflected in the definition of $SZ(D)$, the size of a given D:

$$SZ(\textit{Prim-Rule}\ P_1, \ldots, P_n) = 1$$
$$SZ(D_1; D_2) = SZ(D_1) + SZ(D_2)$$

A deduction D will be called *well-formed* iff every application of **zero-even** in D has zero arguments, every application of **odd-next** has exactly one proposition of the form $\mathit{Even}(n)$ as its argument, and every application of **even-next** has exactly one proposition of the form $\mathit{Odd}(n)$ as an argument. It is trivial to check that a proof is well-formed. Unless we explicitly say otherwise, in what follows we will only be concerned with well-formed proofs.

Finally, we stipulate that the composition operator associates to the right, so that $D_1; D_2; D_3$ stands for $D_1; (D_2; D_3)$.

Evaluation semantics

The formal semantics of $\mathcal{PAR}$ are given in the style of Kahn [7] by the five rules shown in Figure 1.1. These rules establish judgments of the form $\beta \vdash D \rightsquigarrow P$, where β is an *assumption base*. An assumption base β is a finite set of propositions, say $\{\mathit{Even}(45), \mathit{Odd}(97)\}$. Intuitively, the elements of an assumption base are *premises*—propositions that are taken to hold for the purposes of a proof. In particular, a judgment of the form $\beta \vdash D \rightsquigarrow P$ states that "With respect to the assumption base β, proof D derives the proposition P"; or "Relative to β, D produces P"; or "D evaluates to P in the context of β"; etc. We write $\mathbf{AB}[\mathcal{PAR}]$ for the set of all assumption bases of $\mathcal{PAR}$ (or simply **AB** when $\mathcal{PAR}$ is tacitly understood).

Here is an informal explanation of the rules of Figure 1.1:

1. The axiom [*ZeroEven*] can be used to derive the proposition $\mathit{Even}(0)$ in *any* assumption base β. This postulates that 0 is even.

2. [*EvenNext*] captures the following rule: if we know that n is odd, then we may conclude that $n + 1$ is even. More precisely, [*EvenNext*] says that if the assumption base contains the proposition $\mathit{Odd}(n)$, then applying the rule **even-next** to $\mathit{Odd}(n)$ will produce the conclusion

$Even(n+1)$. For example, if the assumption base contains $Odd(5)$, then the primitive application **even-next** $Odd(5)$ will result in $Even(6)$. So, for instance,

$$\{Odd(5)\} \vdash \textbf{even-next}\ Odd(5) \rightsquigarrow Even(6). \tag{1.2}$$

Likewise,

$$\{Even(873), Odd(5), Odd(99)\} \vdash \textbf{even-next}\ Odd(5) \rightsquigarrow Even(6). \tag{1.3}$$

Both 1.2 and 1.3 are instances of [*EvenNext*], and thus hold by virtue of it.

3. Similarly, [*OddNext*] says that if we know n to be even, i.e., if $Even(n)$ is in the assumption base, then we may conclude that $n+1$ is odd. Thus, for example, [*OddNext*] ensures that the following judgment holds:

$$\{Even(46) \vdash \textbf{odd-next}\ Even(46) \rightsquigarrow Odd(47).$$

4. Finally, [*Comp*] allows for proof composition. The rule is better read backwards: to evaluate $D_1; D_2$ in some assumption base β, we first evaluate D_1 in β, getting some conclusion P_1 from it, and then we evaluate D_2 in $\beta \cup \{P_1\}$, i.e., in β *augmented with* P_1. Thus the result of D_1 serves as a lemma within D_2. The result of D_2 is the result of the entire composition.

As an example, let

$$D \quad = \quad \begin{array}{l} \textbf{zero-even}; \\ \textbf{odd-next}\ Even(0); \\ \textbf{even-next}\ Odd(1). \end{array}$$

The following derivation shows that $\emptyset \vdash D \rightsquigarrow Even(2)$:

1. $\emptyset \vdash \textbf{zero-even} \rightsquigarrow Even(0)$ [*ZeroEven*]
2. $\{Even(0)\} \vdash \textbf{odd-next}\ Even(0) \rightsquigarrow Odd(1)$ [*OddNext*]
3. $\{Even(0), Odd(1)\} \vdash \textbf{even-next}\ Odd(1) \rightsquigarrow Even(2)$ [*EvenNext*]
4. $\{Even(0)\} \vdash \textbf{odd-next}\ Even(0); \textbf{even-next}\ Odd(1) \rightsquigarrow Even(2)$ 2, 3, [*Comp*]
5. $\emptyset \vdash \textbf{zero-even}; \textbf{odd-next}\ Even(0); \textbf{even-next}\ Odd(1) \rightsquigarrow Even(2)$ 1, 4, [*Comp*]

The following is an important result. It can be proved directly by induction on D, but it also follows from Theorem 1.6 below.

Theorem 1.1 (Conclusion Uniqueness) *If* $\beta_1 \vdash D \rightsquigarrow P_1$ *and* $\beta_2 \vdash D \rightsquigarrow P_2$ *then* $P_1 = P_2$.

Denotational semantics: a $\mathcal{PAR}$ interpreter

For a denotational semantics for $\mathcal{PAR}$, let *error* be an object that is distinct from all propositions. We define a meaning function

$$\mathcal{M} : \textbf{Ded} \to \textbf{AB} \to \textbf{Prop} \cup \{error\}$$

that takes a deduction D followed by an assumption base β and produces an element of $\textbf{Prop} \cup \{error\}$ that is regarded as "the meaning" of D with respect to β. The definition of $\mathcal{M}[\![D]\!]$ is driven by the

structure of D. We use the notation $e_1\,? \rightarrow e_2,\, e_3$ to mean "if e_1 then e_2 else e_3".

$$\begin{aligned}
\mathcal{M}[\![\textbf{zero-even}]\!]\,\beta &= \mathit{Even}(0) \\
\mathcal{M}[\![\textbf{even-next}\ \mathit{Odd}(n)]\!]\,\beta &= \mathit{Odd}(n) \in \beta\,? \rightarrow \mathit{Even}(n+1),\ \mathit{error} \\
\mathcal{M}[\![\textbf{odd-next}\ \mathit{Even}(n)]\!]\,\beta &= \mathit{Even}(n) \in \beta\,? \rightarrow \mathit{Odd}(n+1),\ \mathit{error} \\
\mathcal{M}[\![D_1; D_2]\!]\,\beta &= (\mathcal{M}[\![D_1]\!]\,\beta) = \mathit{error}\,? \rightarrow \mathit{error},\ \mathcal{M}[\![D_2]\!]\,\beta \cup (\mathcal{M}[\![D_1]\!]\,\beta)
\end{aligned}$$

or, equivalently,

$$\begin{aligned}
\mathcal{M}[\![\textbf{zero-even}]\!] &= \lambda\,\beta\,.\,\mathit{Even}(0) \\
\mathcal{M}[\![\textbf{even-next}\ \mathit{Odd}(n)]\!] &= \lambda\,\beta\,.\,\mathit{Odd}(n) \in \beta\,? \rightarrow \mathit{Even}(n+1),\ \mathit{error} \\
\mathcal{M}[\![\textbf{odd-next}\ \mathit{Even}(n)]\!] &= \lambda\,\beta\,.\,\mathit{Even}(n) \in \beta\,? \rightarrow \mathit{Odd}(n+1),\ \mathit{error} \\
\mathcal{M}[\![D_1; D_2]\!] &= \lambda\,\beta\,.\,(\mathcal{M}[\![D_1]\!]\,\beta) = \mathit{error}\,? \rightarrow \mathit{error},\ \mathcal{M}[\![D_2]\!]\,\beta \cup (\mathcal{M}[\![D_1]\!]\,\beta)
\end{aligned}$$

The latter formulation emphasizes more directly that the meaning of a proof is *a function over assumption bases*, a key characteristic of DPLs.

The above equations can be readily transcribed into an algorithm that takes a deduction D and an assumption base β and either produces a proposition P or else generates an error. In fact we can look at $\mathcal{M}$ itself as an interpreter for $\mathcal{PAR}$. In that light, it is easy to see that $\mathcal{M}$ always terminates. Furthermore, assuming that looking up a proposition in an assumption base takes constant time on average (e.g., by implementing assumption bases as hash tables), $\mathcal{M}$ terminates in $O(SZ(D))$ time in the average case. The worst-case complexity is $O(n \cdot \log n)$, where $n = SZ(D)$, achieved by implementing assumption bases as balanced trees, which guarantees logarithmic look-up time in the worst case. We refer to the process of obtaining $\mathcal{M}[\![D]\!]\,\beta$ as *evaluating D in β*.

Theorem 1.2 *Evaluating a deduction D in a given assumption base takes $O(n)$ time on average, and $O(n \cdot \log n)$ time in the worst case, where n is the size of D.*

The following result relates the denotational semantics to the evaluation semantics of Figure 1.1. If we view $\mathcal{M}$ as an interpreter for $\mathcal{PAR}$, the theorem below becomes a correctness result for the interpreter: it tells us that $\mathcal{M}$ will produce a result P for a given D and β iff the judgment $\beta \vdash D \rightsquigarrow P$ holds according to the evaluation semantics.

Theorem 1.3 (Coincidence of the two semantics) *For all D, β, and P:*

$$\beta \vdash D \rightsquigarrow P \ \ \textit{iff}\ \ \mathcal{M}[\![D]\!]\,\beta = P.$$

Hence, by termination, $\mathcal{M}[\![D]\!]\,\beta = \mathit{error}$ iff there is no P such that $\beta \vdash D \rightsquigarrow P$.

Metatheory

We will call a proposition of the form $\mathit{Even}(n)$ ($\mathit{Odd}(n)$) *true* iff n is even (odd). A proposition that is not true will be called *false*. Further, we will say that a proposition P *follows from* a set of propositions β, written $\beta \models P$, iff P is true whenever every element of β is true. So, for instance, we vacuously have $\{\mathit{Odd}(0)\} \models \mathit{Even}(1)$ by virtue of the fact that $\mathit{Odd}(0)$ is false. Note that $\emptyset \models P$ iff P is true. We have:

Theorem 1.4 (Soundness) *If $\beta \vdash D \rightsquigarrow P$ then $\beta \models P$. In particular, if $\emptyset \vdash D \rightsquigarrow P$ then P is true.*

Theorem 1.5 (Completeness) *If P is true then there is a D such that $\beta \vdash D \rightsquigarrow P$ for all β.*

Basic proof theory

We define $\mathcal{C}(D)$, the *conclusion* of a proof D, as follows:

$$\begin{aligned} \mathcal{C}(\textbf{zero-even}) &= \mathit{Even}(0) \\ \mathcal{C}(\textbf{even-next } \mathit{Odd}(n)) &= \mathit{Even}(n+1) \\ \mathcal{C}(\textbf{odd-next } \mathit{Even}(n)) &= \mathit{Odd}(n+1) \\ \mathcal{C}(D_1; D_2) &= \mathcal{C}(D_2) \end{aligned}$$

This definition can be used as a recursive algorithm for computing $\mathcal{C}(D)$. Computing $\mathcal{C}(D)$ is quite different—much easier—than evaluating D in a given β. For example, if D is of the form $D_1; D_2; \ldots; D_{99}; D_{100}$, we can completely ignore the first ninety-nine deductions and simply compute $\mathcal{C}(D_{100})$, since, by definition, $\mathcal{C}(D) = \mathcal{C}(D_{100})$. The catch, of course, is that D might fail to establish its conclusion (in a given assumption base); evaluation is the only way to find out. However, it is easy to show that *if* $\beta \vdash D \rightsquigarrow P$ then $P = \mathcal{C}(D)$. Hence, in terms of the interpreter $\mathcal{M}$, either $\mathcal{M}[\![D]\!]\,\beta = \mathit{error}$ or $\mathcal{M}[\![D]\!]\,\beta = \mathcal{C}(D)$, for any D and β. Loosely paraphrased: a proof succeeds iff it produces its purported conclusion.

Theorem 1.6 *If $\beta \vdash D \rightsquigarrow P$ then $P = \mathcal{C}(D)$. Therefore, by Theorem 1.3, either $\mathcal{M}[\![D]\!]\,\beta = \mathcal{C}(D)$ or else $\mathcal{M}[\![D]\!]\,\beta = \mathit{error}$.*

Next, we define a function $\mathit{FA} : \textbf{Ded} \to \mathcal{P}_\infty(\textbf{Prop})$, where $\mathcal{P}_\infty(\textbf{Prop})$ denotes the set of all finite subsets of **Prop**. We call the elements of $\mathit{FA}(D)$ the *free assumptions* of D. Intuitively, a free assumption of D is a proposition that D uses without proof. It is a "premise" of D—a necessary assumption that D takes for granted. The main result of this section is that D can be successfully evaluated only in an assumption base that contains its free assumptions. In particular, we define:

$$\begin{aligned} \mathit{FA}(\textbf{zero-even}) &= \emptyset \\ \mathit{FA}(\textbf{even-next } \mathit{Odd}(n)) &= \{\mathit{Odd}(n)\} \\ \mathit{FA}(\textbf{odd-next } \mathit{Even}(n)) &= \{\mathit{Even}(n)\} \\ \mathit{FA}(D_1; D_2) &= \mathit{FA}(D_1) \cup (\mathit{FA}(D_2) - \{\mathcal{C}(D_1)\}) \end{aligned}$$

Theorem 1.7 *$\beta \vdash D \rightsquigarrow P$ iff $\mathit{FA}(D) \subseteq \beta$. Accordingly, $\mathcal{M}[\![D]\!]\,\beta = \mathit{error}$ iff there is a P such that $P \in \mathit{FA}(D)$ and $P \notin \beta$.*

The above result suggests an alternative way of evaluating a proof D in a given β: compute $\mathit{FA}(D)$ and check whether each of its elements appears in β. If so, output $\mathcal{C}(D)$; otherwise, output *error*.

Corollary 1.8 *If $\mathit{FA}(D) = \emptyset$ then $\beta \vdash D \rightsquigarrow \mathcal{C}(D)$ for all β.*

Next, let us say that two proofs D_1 and D_2 are *observationally equivalent with respect to an assumption base* β, written $D_1 \approx_\beta D_2$, whenever

$$\beta \vdash D_1 \rightsquigarrow P \quad \text{iff} \quad \beta \vdash D_2 \rightsquigarrow P \tag{1.4}$$

for all P. For instance, for $\beta_0 = \{\mathit{Even}(4), \mathit{Odd}(5)\}$ we have

$$\textbf{odd-next } \mathit{Even}(4); \textbf{even-next } \mathit{Odd}(5) \approx_{\beta_0} \textbf{even-next } \mathit{Odd}(5).$$

These two proofs are observationally equivalent in β_0 because both of them yield the conclusion $\mathit{Even}(6)$ when we evaluate them in β_0. Likewise, we have

$$\textbf{even-next } \mathit{Odd}(35) \approx_{\beta_0} \textbf{odd-next } \mathit{Even}(56)$$

because both of these claims will produce *error* when we evaluate them in β_0. We clearly have:

Theorem 1.9 *$D_1 \approx_\beta D_2$ iff $\mathcal{M}[\![D_1]\!]\,\beta = \mathcal{M}[\![D_2]\!]\,\beta$.*

Thus D_1 and D_2 are observationally equivalent in a given β iff when we evaluate them in β we obtain the same result: either the same proposition, or an error in both cases. It follows that the relation $\approx_\beta$ is decidable.

If we have $D_1 \approx_\beta D_2$ for *all* β then we write $D_1 \approx D_2$ and say that D_1 and D_2 are *observationally equivalent*. The reader will verify that $\approx$ is an equivalence relation.

Theorem 1.10 *If $D_1 \approx D_1'$ and $D_2 \approx D_2'$ then $D_1; D_2 \approx D_1'; D_2'$.*

Theorem 1.11 *$(D_1; D_2); D_3 \not\approx D_1; (D_2; D_3)$.*

Proof: Take $D_1 =$ **even-next** $\mathit{Odd}(5)$, $D_2 =$ **zero-even**, $D_3 =$ **odd-next** $\mathit{Even}(6)$, and consider any β that contains $\mathit{Odd}(5)$ but not $\mathit{Even}(6)$. ■

At first glance it might appear that $\approx$ is undecidable, as it is defined by quantifying over all assumption bases. However, the following result shows that two proofs are observationally equivalent iff they have the same conclusion and the same free assumptions. Since both of these are computable, it follows that $\approx$ is decidable:

Theorem 1.12 *$D_1 \approx D_2$ iff $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ and $\mathit{FA}(D_1) = \mathit{FA}(D_2)$.*

For instance, for D_1, D_2 and D_3 as defined in the proof of Theorem 1.11, we have

$$\mathcal{C}(D_1; (D_2; D_3)) = \mathcal{C}((D_1; D_2); D_3) = \mathit{Odd}(7)$$

but

$$\mathit{FA}(D_1; (D_2; D_3)) = \{\mathit{Odd}(5)\} \neq \{\mathit{Odd}(5), \mathit{Even}(6)\} = \mathit{FA}((D_1; D_2); D_3)$$

and this is why the two compositions are not observationally equivalent.

1.3 A DPL for classical propositional logic

In this section we will formulate and study $\mathcal{NDL}_0$, a type-α DPL for classical zero-order natural deduction. (Since this paper presents no risk of confusion, we will often write $\mathcal{NDL}$ instead of $\mathcal{NDL}_0$.)

Abstract syntax

We will use the letters P, Q, $R, \ldots$, to designate arbitrary *propositions*. Propositions are built from the following abstract grammar:

$$P ::= A \mid \mathbf{true} \mid \mathbf{false} \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid P \Leftrightarrow Q$$

where A ranges over a countable set of atomic propositions ("atoms") which we need not specify in detail. The letters A, B, and C will be used as typical atoms. We stipulate that the connective $\neg$ has the highest precedence, followed by $\wedge$ and $\vee$ (both of which have equal precedence and associate to the right), followed by $\Rightarrow$ and $\Leftrightarrow$ (where, again, both $\Rightarrow$ and $\Leftrightarrow$ have equal precedence and are right-associative). Thus, for instance, $\neg A \vee B \wedge C \Rightarrow B$ stands for $[(\neg A) \vee (B \wedge C)] \Rightarrow B$. We will not

Prim-Rule	::=	**claim**	(reiteration)
	|	**modus-ponens**	($\Rightarrow$-introduction)
	|	**modus-tollens**	($\neg$-introduction)
	|	**double-negation**	($\neg$-elimination)
	|	**both**	($\wedge$-introduction)
	|	**left-and**	($\wedge$-elimination)
	|	**right-and**	($\wedge$-elimination)
	|	**left-either**	($\vee$ -introduction)
	|	**right-either**	($\vee$ -introduction)
	|	**constructive-dilemma**	($\vee$ -elimination)
	|	**equivalence**	($\Leftrightarrow$ -introduction)
	|	**left-iff**	($\Leftrightarrow$ -elimination)
	|	**right-iff**	($\Leftrightarrow$ -elimination)
	|	**true-intro**	(**true**-introduction)
	|	**absurd**	(**false**-introduction)
	|	**false-elim**	(**false**-elimination)

Figure 1.2: Primitive inference rules

rely too much on these conventions, however; parsing ambiguities will be resolved mostly by using parentheses and brackets.

The proofs (or "deductions") of $\mathcal{NDL}$ have the following abstract syntax:

$$D ::= \textit{Prim-Rule}\ P_1, \ldots, P_n \mid D_1; D_2 \mid \textbf{assume}\ P\ \textbf{in}\ D \tag{1.5}$$

where *Prim-Rule* ranges over a collection of primitive inference rules such as modus ponens. The decision of exactly which rules to choose as primitives has important ramifications for the metatheory of the language (it affects soundness and completeness, in particular), but has little bearing on the abstract syntax of the language or on the core semantic framework. This is similar to the distinction between the core syntax and semantics of the λ-calculus, on the one hand, and the δ-reduction rules for the constants, on the other. For instance, if we take as a primitive rule one that derives $P \wedge Q$ from P, we will clearly have an unsound language; and if we leave out certain rules such as $\Leftrightarrow$-elimination, then we will have an incomplete language. The rules shown in Figure 1.2 allow for a sound and complete natural deduction system. As before, we will write **Prop**$[\mathcal{NDL}_0]$ and **Ded**$[\mathcal{NDL}_0]$ for the sets of all propositions and all deductions, respectively, or simply **Prop** and **Ded** when $\mathcal{NDL}_0$ is understood.

The reader will notice one omission from Figure 1.2: an introduction rule for $\Rightarrow$. This has traditionally been the most troublesome spot for classical deduction systems. As we will see shortly, in $\mathcal{NDL}$ conditionals are introduced via the language construct **assume**, in a manner that avoids most of the usual problems.

As before, a proof will be called a *rule application* if it is of the form *Prim-Rule* $P_1, \ldots, P_n$, and a *composition* if it is of the form $D_1; D_2$. Deductions of the form **assume** P **in** D will be called *hypothetical* or *conditional*. In a hypothetical deduction of the above form, P and D are called the *hypothesis* and the *body* of the deduction, respectively. The body represents the *scope* of the corresponding hypothesis. Compositions and hypothetical proofs are called *complex*, because they have recursive structure; while rule applications are *atomic* proofs. The following equations define

$$\frac{\beta \vdash D_1 \rightsquigarrow P_1 \qquad \beta \cup \{P_1\} \vdash D_2 \rightsquigarrow P_2}{\beta \vdash D_1; D_2 \rightsquigarrow P_2} \quad [Comp]$$

$$\frac{\beta \cup \{P\} \vdash D \rightsquigarrow Q}{\beta \vdash \textbf{assume } P \textbf{ in } D \rightsquigarrow P \Rightarrow Q} \quad [Assume]$$

Figure 1.3: Evaluation semantics for complex $\mathcal{NDL}$ proofs..

$SZ(D)$, the *size* of D, by structural recursion:

$$\begin{aligned} SZ(\textit{Prim-Rule } P_1, \ldots, P_n) &= 1 \\ SZ(D_1; D_2) &= SZ(D_1) + SZ(D_2) \\ SZ(\textbf{assume } P \textbf{ in } D) &= SZ(D) + 1 \end{aligned}$$

Grammar 1.5 specifies the abstract syntax of proofs; it cannot serve as a concrete syntax because it is ambiguous. For instance, it is not clear whether

$$\textbf{assume } P \wedge Q \textbf{ in true}; \textbf{ left-and } P \wedge Q$$

is a hypothetical deduction with the composition **true**; **left-and** $P \wedge Q$ as its body, or a composition consisting of a hypothetical deduction followed by an application of **left-and**. We will use **begin-end** pairs or parentheses to remove any such ambiguity. Finally, as before, we stipulate that the composition operator ; associates to the right.

Evaluation semantics

The evaluation semantics of $\mathcal{NDL}$ are given by a collection of rules, shown in Figure 1.3 and Figure 1.4, that establish judgments of the form $\beta \vdash D \rightsquigarrow P$, where, just as in $\mathcal{PAR}$, β is an assumption base, i.e., a finite set of propositions (where the definition of proposition, of course, is now different). A judgment of this form should be read just as in $\mathcal{PAR}$: "In the context of β, D produces the conclusion P".

The evaluation rules are partitioned into two groups: those dealing with complex proofs, shown in Figure 1.3; and those dealing with atomic proofs, i.e., with rule applications, shown in Figure 1.4. Our first result about this semantics is the analogue of Theorem 1.1 for $\mathcal{PAR}$. This can also be proved directly, or as an immediate corollary of Theorem 1.16 or Theorem 1.19 below.

Theorem 1.13 (Conclusion Uniqueness) *If $\beta_1 \vdash D \rightsquigarrow P_1$ and $\beta_2 \vdash D \rightsquigarrow P_2$ then $P_1 = P_2$.*

We end this section by introducing some useful syntax sugar. Although we listed **modus-tollens** as an introduction rule for negation, in practice negations in $\mathcal{NDL}$ are usually introduced via deductions of the form

$$\textbf{suppose-absurd } P \textbf{ in } D \tag{1.6}$$

that perform reasoning by contradiction. A proof of the form 1.6 seeks to establish the negation of P by deriving a contradiction. Thus, operationally, the evaluation of 1.6 in an assumption base β proceeds as follows: we add the hypothesis P to β and evaluate the body D in the augmented

$$\beta \cup \{P\} \vdash \textbf{claim } P \rightsquigarrow P$$
$$\beta \cup \{P \Rightarrow Q, P\} \vdash \textbf{modus-ponens } P \Rightarrow Q, P \rightsquigarrow Q$$
$$\beta \cup \{P \Rightarrow Q, \neg Q\} \vdash \textbf{modus-tollens } P \Rightarrow Q, \neg Q \rightsquigarrow \neg P$$
$$\beta \cup \{\neg\neg P\} \vdash \textbf{double-negation } \neg\neg P \rightsquigarrow P$$
$$\beta \cup \{P_1, P_2\} \vdash \textbf{both } P_1, P_2 \rightsquigarrow P_1 \wedge P_2$$
$$\beta \cup \{P_1 \wedge P_2\} \vdash \textbf{left-and } P_1 \wedge P_2 \rightsquigarrow P_1$$
$$\beta \cup \{P_1 \wedge P_2\} \vdash \textbf{right-and } P_1 \wedge P_2 \rightsquigarrow P_2$$
$$\beta \cup \{P_1\} \vdash \textbf{left-either } P_1, P_2 \rightsquigarrow P_1 \vee P_2$$
$$\beta \cup \{P_2\} \vdash \textbf{right-either } P_1, P_2 \rightsquigarrow P_1 \vee P_2$$
$$\beta \cup \{P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q\} \vdash \textbf{constructive-dilemma } P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q \rightsquigarrow Q$$
$$\beta \cup \{P_1 \Rightarrow P_2, P_2 \Rightarrow P_1\} \vdash \textbf{equivalence } P_1 \Rightarrow P_2, P_2 \Rightarrow P_1 \rightsquigarrow P_1 \Leftrightarrow P_2$$
$$\beta \cup \{P_1 \Leftrightarrow P_2\} \vdash \textbf{left-iff } P_1 \Leftrightarrow P_2 \rightsquigarrow P_1 \Rightarrow P_2$$
$$\beta \cup \{P_1 \Leftrightarrow P_2\} \vdash \textbf{right-iff } P_1 \Leftrightarrow P_2 \rightsquigarrow P_2 \Rightarrow P_1$$
$$\beta \vdash \textbf{true-intro} \rightsquigarrow \textbf{true}$$
$$\beta \cup \{P, \neg P\} \vdash \textbf{absurd } P, \neg P \rightsquigarrow \textbf{false}$$
$$\beta \vdash \textbf{false-elim} \rightsquigarrow \neg\textbf{false}$$

Figure 1.4: Evaluation axioms for rule applications.

assumption base. If and when D produces a contradiction (the proposition **false**), we output the negation $\neg P$. This is justified because P led us to a contradiction—hence we must have $\neg P$.[3] More formally, the semantics of 1.6 are given by the following rule:

$$\frac{\beta \cup \{P\} \vdash D \rightsquigarrow \textbf{false}}{\beta \vdash \textbf{suppose-absurd } P \textbf{ in } D \rightsquigarrow \neg P} \tag{1.7}$$

However, we do not take proofs of the form 1.6 as primitive because their intended behavior is expressible in terms of **assume**, primitive inference rules, and composition. In particular, 1.6 is defined as an abbreviation for the deduction

assume P **in** D;
false-elim;
modus-tollens $P \Rightarrow$ **false**, $\neg$**false**

It is readily shown that this desugaring results in the intended semantics 1.7:

Theorem 1.14 *If* $\beta \cup \{P\} \vdash D \rightsquigarrow$ **false** *then* $\beta \vdash$ **suppose-absurd** P **in** $D \rightsquigarrow \neg P$.

Denotational semantics: an interpreter for $\mathcal{NDL}$

For a denotational semantics for $\mathcal{NDL}$, we again introduce an element *error* distinct from all propositions. We define a meaning function

$$\mathcal{M} : \textbf{Ded} \rightarrow \textbf{AB} \rightarrow \textbf{Prop} \cup \{\mathit{error}\}$$

[3]Note that if P was not necessary in deriving the contradiction then β must have already been inconsistent, in which case $\neg P$ follows vacuously, since every proposition follows from an inconsistent assumption base. Hence, in either case, $\neg P$ follows logically from β.

$$
\begin{array}{rcl}
\mathcal{M}[\![D_1; D_2]\!] & = & \lambda\,\beta\,.\,(\mathcal{M}[\![D_1]\!]\,\beta) = \mathit{error}\,?\,{\to}\,\mathit{error},\ \mathcal{M}[\![D_2]\!]\,\beta \cup (\mathcal{M}[\![D_1]\!]\,\beta) \\
\mathcal{M}[\![\textbf{assume } P \textbf{ in } D]\!] & = & \lambda\,\beta\,.\,(\mathcal{M}[\![D]\!]\,\beta \cup \{P\}) = \mathit{error}\,?\,{\to}\,\mathit{error},\ P \Rightarrow (\mathcal{M}[\![D]\!]\,\beta \cup \{P\}) \\
\mathcal{M}[\![\textbf{claim } P]\!] & = & \lambda\,\beta\,.\,P \in \beta\,?\,{\to}\,P,\ \mathit{error} \\
\mathcal{M}[\![\textbf{both } P, Q]\!] & = & \lambda\,\beta\,.\,\{P, Q\} \subseteq \beta\,?\,{\to}\,P \wedge Q,\ \mathit{error} \\
\mathcal{M}[\![\textbf{left-and } P \wedge Q]\!] & = & \lambda\,\beta\,.\,P \wedge Q \in \beta\,?\,{\to}\,P,\ \mathit{error} \\
& \vdots &
\end{array}
$$

Figure 1.5: Alternative denotational semantics for $\mathcal{NDL}$.

(where $\mathbf{AB} = \mathcal{P}_\infty(\mathbf{Prop})$) as follows:

$$
\begin{array}{rcl}
\mathcal{M}[\![P]\!]\,\beta & = & P \in \beta \cup \{\mathbf{true}, \neg\mathbf{false}\}\,?\,{\to}\,P,\ \mathit{error} \\
\mathcal{M}[\![\textbf{assume } P \textbf{ in } D]\!]\,\beta & = & (\mathcal{M}[\![D]\!]\,\beta \cup \{P\}) = \mathit{error}\,?\,{\to}\,\mathit{error},\ P \Rightarrow (\mathcal{M}[\![D]\!]\,\beta \cup \{P\}) \\
\mathcal{M}[\![D_1; D_2]\!]\,\beta & = & (\mathcal{M}[\![D_1]\!]\,\beta) = \mathit{error}\,?\,{\to}\,\mathit{error},\ \mathcal{M}[\![D_2]\!]\,\beta \cup (\mathcal{M}[\![D_1]\!]\,\beta).
\end{array}
$$

The equations for rule applications are given by a case analysis of the rule. We illustrate with the equations for **modus-ponens**, **left-and**, and **both**; the rest can be similarly translated from Figure 1.4.

$$
\begin{array}{rcl}
\mathcal{M}[\![\textbf{modus-ponens } P \Rightarrow Q, P]\!]\,\beta & = & \{P \Rightarrow Q, P\} \subseteq \beta\,?\,{\to}\,Q,\ \mathit{error} \\
\mathcal{M}[\![\textbf{left-and } P \wedge Q]\!]\,\beta & = & P \wedge Q \in \beta\,?\,{\to}\,P,\ \mathit{error} \\
\mathcal{M}[\![\textbf{both } P, Q]\!]\,\beta & = & \{P, Q\} \subseteq \beta\,?\,{\to}\,P \wedge Q,\ \mathit{error}
\end{array}
$$

Equivalently, we can choose to emphasize that the denotation of a proof is a function over assumption bases by recasting the above equations in the form shown in Figure 1.5.

As in the case of $\mathcal{PAR}$, we can view $\mathcal{M}$ as an interpreter for $\mathcal{NDL}$. It takes a deduction D and an assumption base β and produces "the meaning" of D with respect to β: either a proposition P, which we regard as the conclusion of D, or the token *error*. In that light, we can see that $\mathcal{M}$ always terminates promptly:

Theorem 1.15 *Computing $\mathcal{M}[\![D]\!]\,\beta$, i.e., evaluating D in β, requires $O(n)$ time in the average case and $O(n \cdot \log n)$ time in the worst case, where n is the size of D.*

The following result relates the interpreter $\mathcal{M}$ to the evaluation semantics; it is the analogue of the $\mathcal{PAR}$ Theorem 1.3.

Theorem 1.16 (Coincidence of the two semantics) *For all D, β, and P:*

$$\beta \vdash D \rightsquigarrow P \ \ \textit{iff}\ \ \mathcal{M}[\![D]\!]\,\beta = P.$$

Hence, by termination, $\mathcal{M}[\![D]\!]\,\beta = \mathit{error}$ iff there is no P such that $\beta \vdash D \rightsquigarrow P$.

Examples

In this section we present $\mathcal{NDL}$ proofs of some well-known tautologies of propositional logic. The reader is invited to read the first few of these and tackle the rest on his own.

$$\boxed{P \Rightarrow \neg\neg P}$$

Proof:

assume P **in**
  suppose-absurd $\neg P$ **in**
    absurd $P, \neg P$

$$\boxed{(P \Rightarrow Q) \Rightarrow [(Q \Rightarrow R) \Rightarrow (P \Rightarrow R)]}$$

Proof:

assume $P \Rightarrow Q$ **in**
  assume $Q \Rightarrow R$ **in**
    assume P **in**
      begin
        modus-ponens $P \Rightarrow Q, P$;
        modus-ponens $Q \Rightarrow R, Q$
      end

$$\boxed{[P \Rightarrow (Q \Rightarrow R)] \Rightarrow [Q \Rightarrow (P \Rightarrow R)]}$$

Proof:

assume $P \Rightarrow (Q \Rightarrow R)$ **in**
  assume Q **in**
    assume P **in**
      begin
        modus-ponens $P \Rightarrow (Q \Rightarrow R), P$;
        modus-ponens $Q \Rightarrow R, Q$
      end

$$\boxed{[P \Rightarrow (Q \Rightarrow R)] \Rightarrow [(P \wedge Q) \Rightarrow R]}$$

Proof:

assume $P \Rightarrow (Q \Rightarrow R)$ **in**
  assume $P \wedge Q$ **in**
    begin
      left-and $P \wedge Q$;
      modus-ponens $P \Rightarrow (Q \Rightarrow R), P$;
      right-and $P \wedge Q$;
      modus-ponens $Q \Rightarrow R, Q$
    end

$$\boxed{[(P \wedge Q) \Rightarrow R] \Rightarrow [P \Rightarrow (Q \Rightarrow R)]}$$

Proof:
assume $(P \wedge Q) \Rightarrow R$ **in**
assume P **in**
assume Q **in**
begin
both P, Q;
modus-ponens $(P \wedge Q) \Rightarrow R, P \wedge Q$
end

$$\boxed{(P \Rightarrow Q) \Rightarrow (\neg Q \Rightarrow \neg P)}$$

Proof:
assume $P \Rightarrow Q$ **in**
assume $\neg Q$ **in**
modus-tollens $P \Rightarrow Q, \neg Q$;

$$\boxed{(\neg Q \Rightarrow \neg P) \Rightarrow (P \Rightarrow Q)}$$

Proof:
assume $\neg Q \Rightarrow \neg P$ **in**
assume P **in**
begin
suppose-absurd $\neg Q$ **in**
begin
modus-ponens $\neg Q \Rightarrow \neg P, \neg Q$;
absurd $P, \neg P$
end;
double-negation $\neg\neg Q$
end

$$\boxed{(P \vee Q) \Rightarrow (Q \vee P)}$$

Proof:
assume $P \vee Q$ **in**
begin
assume P **in**
right-either Q, P;
assume Q **in**
left-either Q, P;
constructive-dilemma $P \vee Q, P \Rightarrow Q \vee P, Q \Rightarrow Q \vee P$
end

$$\boxed{(\neg P \vee Q) \Rightarrow (P \Rightarrow Q)}$$

Proof:
assume $\neg P \vee Q$ **in**
assume P **in**
begin
suppose-absurd $\neg Q$ **in**
begin
assume $\neg P$ **in**
absurd $P, \neg P$;
assume Q **in**
absurd $Q, \neg Q$;
constructive-dilemma $\neg P \vee Q, \neg P \Rightarrow \textbf{false}, Q \Rightarrow \textbf{false}$
end;
double-negation $\neg\neg Q$
end

$$\boxed{\neg(P \vee Q) \Rightarrow (\neg P \wedge \neg Q)}$$

Proof:
assume $\neg(P \vee Q)$ **in**
begin
suppose-absurd P **in**
begin
left-either P, Q;
absurd $P \vee Q, \neg(P \vee Q)$
end;
suppose-absurd Q **in**
begin
right-either P, Q;
absurd $P \vee Q, \neg(P \vee Q)$
end;
both $\neg P, \neg Q$
end

$$\boxed{(\neg P \vee \neg Q) \Rightarrow \neg(P \wedge Q)}$$

Proof:
assume $\neg P \vee \neg Q$ **in**
suppose-absurd $P \wedge Q$ **in**
begin
left-and $P \wedge Q$;
right-and $P \wedge Q$;
assume $\neg P$ **in**
absurd $P, \neg P$;
assume $\neg Q$ **in**
absurd $Q, \neg Q$;

constructive-dilemma $\neg P \vee \neg Q, \neg P \Rightarrow \textbf{false}, \neg Q \Rightarrow \textbf{false}$
end

$$\boxed{(\neg P \wedge \neg Q) \Rightarrow \neg(P \vee Q)}$$

Proof:
assume $\neg P \wedge \neg Q$ **in**
begin
left-and $\neg P \wedge \neg Q$;
right-and $\neg P \wedge \neg Q$;
assume P **in**
absurd $P, \neg P$;
assume Q **in**
absurd $Q, \neg Q$;
suppose-absurd $P \vee Q$ **in**
constructive-dilemma $P \vee Q, P \Rightarrow \textbf{false}, Q \Rightarrow \textbf{false}$
end

$$\boxed{P \Leftrightarrow [P \wedge (P \vee Q)]}$$

Proof:
assume P **in**
begin
left-either P, Q;
both $P, P \vee Q$
end;
assume $P \wedge (P \vee Q)$ **in**
left-and $P \wedge (P \vee Q)$;
equivalence $P \Rightarrow [P \wedge (P \vee Q)], [P \wedge (P \vee Q)] \Rightarrow P$

$$\boxed{[(P \wedge Q) \vee (\neg P \wedge \neg Q)] \Rightarrow (P \Leftrightarrow Q)}$$

Proof:
assume $(P \wedge Q) \vee (\neg P \wedge \neg Q)$ **in**
begin
assume $P \wedge Q$ **in**
begin
left-and $P \wedge Q$;
right-and $P \wedge Q$;
assume P **in**
claim Q;
assume Q **in**
claim P;
equivalence $P \Rightarrow Q, Q \Rightarrow P$
end;
assume $\neg P \wedge \neg Q$ **in**

```
    begin
      left-and ¬P ∧ ¬Q;
      right-and ¬P ∧ ¬Q;
      assume P in
        begin
          suppose-absurd ¬Q in
            absurd P, ¬P;
          double-negation ¬¬Q
        end;
      assume Q in
        begin
          suppose-absurd ¬P in
            absurd Q, ¬Q;
          double-negation ¬¬P
        end;
      equivalence P ⇒ Q, Q ⇒ P
    end;
  constructive-dilemma (P ∧ Q) ∨ (¬P ∧ ¬Q),(P ∧ Q) ⇒ (P ⇔ Q),
                                           (¬P ∧ ¬Q) ⇒ (P ⇔ Q);
end
```

Metatheory

We will view the set of all propositions as the free term algebra formed by the five propositional constructors over the set of atoms, treating the latter as variables. Any Boolean algebra $\mathcal{B}$, say the one with carrier $\{0,1\}$, can be seen as a $\{\mathbf{true}, \mathbf{false}, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$-algebra with respect to the expected realizations ($\wedge^{\mathcal{B}}(x,y) = \min\{x,y\}$, $\vee^{\mathcal{B}}(x,y) = \max\{x,y\}$, $\mathbf{false}^{\mathcal{B}} = 0$, etc.). Now by an *interpretation* $\mathcal{I}$ we will mean a function from the set of atoms to $\{0,1\}$. We will write $\widehat{\mathcal{I}}$ for the unique homomorphic extension of $\mathcal{I}$ to the set of all propositions. Since $\widehat{\mathcal{I}}$ is completely determined by $\mathcal{I}$, we may, for most purposes, conflate the two.

If $\mathcal{I}(P) = 1$ ($\mathcal{I}(P) = 0$) we say that $\mathcal{I}$ *satisfies* (*falsifies*) P, or that it is a *model* of it. This is written as $\mathcal{I} \models P$. If $\mathcal{I} \models P$ for every $P \in \Phi$ then we write $\mathcal{I} \models \Phi$ and say that $\mathcal{I}$ satisfies (or is a model of) Φ. We write $\Phi_1 \models \Phi_2$ to indicate that every model of Φ_1 is also a model of Φ_2. If this is the case we say that the elements of Φ_2 are *logical consequences* of (or are "logically implied by") the propositions in Φ_1. A single proposition P may appear in place of either Φ_1 or Φ_2 as an abbreviation for the singleton $\{P\}$.

We are now ready to state the soundness and completeness theorems for $\mathcal{NDL}$ (proofs of these results can be found elsewhere [2]).

Theorem 1.17 (Soundness) *If $\beta \vdash D \rightsquigarrow P$ then $\beta \models P$.*

Theorem 1.18 (Completeness) *If $\beta \models P$ then there is a D such that $\beta \vdash D \rightsquigarrow P$.*

Basic proof theory

We will say that a proof is *well-formed* iff every rule application in it has one of the forms shown in Figure 1.4. Thus, loosely put, a deduction is well-formed iff the right number and kind of arguments are supplied to every application of a primitive rule, so that, for instance, there are no applications

$$
\begin{array}{rcll}
FA(D_1; D_2) & = & FA(D_1) \cup (FA(D_2) - \{\mathcal{C}(D_1)\}) & (1.8) \\
FA(\textbf{assume } P \textbf{ in } D) & = & FA(D) - \{P\} & (1.9) \\
FA(\textbf{left-either } P_1, P_2) & = & \{P_1\} & (1.10) \\
FA(\textbf{right-either } P_1, P_2) & = & \{P_2\} & (1.11) \\
FA(\textit{Prim-Rule } P_1, \ldots, P_n) & = & \{P_1, \ldots, P_n\} & (1.12)
\end{array}
$$

Figure 1.6: Definition of $FA(D)$ for $\mathcal{NDL}$ ($\textit{Prim-Rule} \notin \{\textbf{left-either}, \textbf{right-either}\}$ in 1.12).

such as **modus-ponens** $A \wedge B, B$ or **left-and true**. Clearly, checking a deduction to make sure that it is well-formed is a trivial matter, and from now on we will take well-formedness for granted. The *conclusion* of a deduction D, denoted $\mathcal{C}(D)$, is defined by structural recursion. For complex D,

$$
\begin{array}{rcl}
\mathcal{C}(D_1; D_2) & = & \mathcal{C}(D_2) \\
\mathcal{C}(\textbf{assume } P \textbf{ in } D) & = & P \Rightarrow \mathcal{C}(D)
\end{array}
$$

while for atomic rule applications we have:

$$
\begin{array}{rcl}
\mathcal{C}(\textbf{claim } P) & = & P \\
\mathcal{C}(\textbf{modus-ponens } P \Rightarrow Q, P) & = & Q \\
\mathcal{C}(\textbf{modus-tollens } P \Rightarrow Q, \neg Q) & = & \neg P \\
\mathcal{C}(\textbf{double-negation } \neg\neg P) & = & P \\
\mathcal{C}(\textbf{both } P, Q) & = & P \wedge Q \\
\mathcal{C}(\textbf{left-and } P \wedge Q) & = & P \\
\mathcal{C}(\textbf{right-and } P \wedge Q) & = & Q \\
\mathcal{C}(\textbf{left-either } P, Q) & = & P \vee Q
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{C}(\textbf{right-either } P, Q) & = & P \vee Q \\
\mathcal{C}(\textbf{cd } P_1 \vee P_2, P_1 \Rightarrow Q, P_2 \Rightarrow Q) & = & Q \\
\mathcal{C}(\textbf{equivalence } P \Rightarrow Q, Q \Rightarrow P) & = & P \Leftrightarrow Q \\
\mathcal{C}(\textbf{left-iff } P \Leftrightarrow Q) & = & P \Rightarrow Q \\
\mathcal{C}(\textbf{right-iff } P \Leftrightarrow Q) & = & Q \Rightarrow P \\
\mathcal{C}(\textbf{true-intro}) & = & \textbf{true} \\
\mathcal{C}(\textbf{absurd } P, \neg P) & = & \textbf{false} \\
\mathcal{C}(\textbf{false-elim}) & = & \neg\textbf{false}
\end{array}
$$

(writing **cd** as an abbreviation for **constructive-dilemma**). Note that the defining equation for compositions is the same as in $\mathcal{PAR}$. Also, just as in $\mathcal{PAR}$, the above equations can be used as a straightforward recursive algorithm for computing $\mathcal{C}(D)$. The next result is the analogue of Theorem 1.6:

Theorem 1.19 *If $\beta \vdash D \rightsquigarrow P$ then $P = \mathcal{C}(D)$. Hence, by Theorem 1.16, either $\mathcal{M}[\![D]\!]\,\beta = \mathcal{C}(D)$ or else $\mathcal{M}[\![D]\!]\,\beta = error$.*

Figure 1.6 defines $FA(D)$, the set of free assumptions of a proof D. The intuitive meaning of $FA(D)$ is the same as in $\mathcal{PAR}$: the elements of $FA(D)$ are propositions that D uses as premises. In the present setting, note in particular the case of hypothetical deductions **assume** P **in** D: the free assumptions here are those of the body D *minus* the hypothesis P.

The computation of $FA(D)$ is not trivial, meaning that it cannot proceed in a local manner down the abstract syntax tree of D using only a fixed amount of memory: a variable amount of state must be maintained in order to deal with clauses 1.9 and 1.8. The latter clause, in particular, is especially computation-intensive because it also calls for the computation of $\mathcal{C}(D_1)$. The reader should reflect on what would be involved in computing, for instance, $FA(D)$ for a D of the form $D_1; \cdots; D_{100}$. In fact the next result shows that the task of evaluating D in a given β can be reduced to the computation of $FA(D)$. It is the analogue of Theorem 1.7 in $\mathcal{PAR}$.

Theorem 1.20 *$\beta \vdash D \rightsquigarrow P$ iff $FA(D) \subseteq \beta$. Accordingly, $\mathcal{M}[\![D]\!]\,\beta = error$ iff there is a P such that $P \in FA(D)$ and $P \notin \beta$.*

As Theorem 1.7 did in the case of $\mathcal{PAR}$, the above result captures the sense in which evaluation in $\mathcal{NDL}$ can be reduced to the computation of *FA*: to compute $\mathcal{M}[\![D]\!]\,\beta$, for any given D and β, simply compute $FA(D)$ and $\mathcal{C}(D)$: if $FA(D) \subseteq \beta$, output $\mathcal{C}(D)$; otherwise output *error*. Intuitively, this reduction is the reason why the computation of $FA(D)$ cannot be much easier than the evaluation of D.

As before, we will say that two proofs D_1 and D_2 are *observationally equivalent with respect to an assumption base* β, written $D_1 \approx_\beta D_2$, whenever

$$\beta \vdash D_1 \rightsquigarrow P \quad \text{iff} \quad \beta \vdash D_2 \rightsquigarrow P \tag{1.13}$$

for all P. For instance,

$$D_1 = \textbf{left-either}\ A, B \quad \text{and} \quad D_2 = \textbf{right-either}\ A, B$$

are observationally equivalent with respect to any assumption base that either contains both A and B or neither of them.

Theorem 1.21 *$D_1 \approx_\beta D_2$ iff $\mathcal{M}[\![D_1]\!]\,\beta = \mathcal{M}[\![D_2]\!]\,\beta$.*

Thus D_1 and D_2 are observationally equivalent in a given β iff when we evaluate them in β we obtain the same result: either the same proposition, or an error in both cases. It follows that $\approx_\beta$ is decidable.

If we have $D_1 \approx_\beta D_2$ for all β then we write $D_1 \approx D_2$ and say that D_1 and D_2 are *observationally equivalent*. For a simple example, we have

$$\textbf{left-iff}\ A \Leftrightarrow A \ \approx\ \textbf{right-iff}\ A \Leftrightarrow A$$

as well as

$$\begin{array}{lcl} \textbf{left-and}\ A \wedge B; & & \textbf{right-and}\ A \wedge B; \\ \textbf{right-and}\ A \wedge B; & \approx & \textbf{left-and}\ A \wedge B; \\ \textbf{both}\ B, A & & \textbf{both}\ B, A \end{array}$$

Again, $\approx$ is an equivalence relation. We can also show again that composition is not associative:

Theorem 1.22 *$D_1; (D_2; D_3) \not\approx (D_1; D_2); D_3$.*

Proof: Take $D_1 = \textbf{double-negation}\ \neg\neg A$, $D_2 = \textbf{true}$, $D_3 = A$, and consider any β that contains $\neg\neg A$ but not A. ■

We note, however, that composition is idempotent (i.e., $D \approx D; D$) and that associativity does hold in the case of claims:

$$\textbf{claim}\ P_1; (\textbf{claim}\ P_2; \textbf{claim}\ P_3) \approx (\textbf{claim}\ P_1; \textbf{claim}\ P_2); \textbf{claim}\ P_3.$$

Commutativity fails in all cases. A certain kind of distributivity holds between the constructor **assume** and the composition operator (as well as between **suppose-absurd** and composition), in the following sense:

Theorem 1.23 *$\textbf{assume}\ P\ \textbf{in}\ (D_1; D_2) \approx D_1; \textbf{assume}\ P\ \textbf{in}\ D_2$ whenever $P \notin FA(D_1)$.*

This result forms the basis for a "hoisting" transformation in the theory of $\mathcal{NDL}$ optimization. We also note that $\approx$ is compatible with the abstract syntax constructors of $\mathcal{NDL}$, a result that is analogous to Theorem 1.10 in the case of $\mathcal{PAR}$.

Theorem 1.24 *If $D \approx D'$ then*

$$\textbf{assume } P \textbf{ in } D \approx \textbf{assume } P \textbf{ in } D'.$$

In addition, if $D_1 \approx D_1'$ and $D_2 \approx D_2'$ then $D_1; D_2 \approx D_1'; D_2'$.

Finally, we derive decidable necessary and sufficient conditions for observational equivalence, as in Theorem 1.12:

Theorem 1.25 *$D_1 \approx D_2$ iff $\mathcal{C}(D_1) = \mathcal{C}(D_2)$ and $FA(D_1) = FA(D_2)$.*

1.4 Contrast with Curry-Howard systems

The reader will notice a similarity between [*Assume*], our evaluation rule for hypothetical deductions, and the typing rule for abstractions in the simply typed λ-calculus:

$$\frac{\beta[x \mapsto \sigma] \vdash E : \tau}{\beta \vdash \lambda\, x : \sigma\, .\, E : \sigma \to \tau}$$

where β in this context is a type environment, i.e., a finite function from variables to types (and $\beta[x \mapsto \sigma]$ is the extension of β with the binding $x \mapsto \sigma$). Likewise, the reader will notice a similarity between [*Comp*], our rule for compositions, and the customary typing rule for **let** expressions:

$$\frac{\beta \vdash E_1 : \tau_1 \qquad \beta[x \mapsto \tau_1] \vdash E_2 : \tau_2}{\beta \vdash \textbf{let } x = E_1 \textbf{ in } E_2 : \tau_2}$$

Indeed, it is straightforward to pinpoint the correspondence between $\mathcal{NDL}$ and the simply typed λ-calculus:

- Deductions correspond to terms, in accordance with the "proofs-as-programs" motto [6, 4, 10]. In particular, **assume**s correspond to λs and compositions correspond to **let**s.
- Propositions correspond to types: e.g., $A \Rightarrow B$ corresponds to $A \to B$, $A \wedge B$ corresponds to $A \times B$, etc.
- Assumption bases correspond to type environments.
- Evaluation in $\mathcal{NDL}$ corresponds to type checking in the λ-calculus.
- Observational equivalence corresponds to type cohabitation in the λ-calculus.
- Optimization via proof transformation in $\mathcal{NDL}$ corresponds to normalization in the λ-calculus.

The correspondence extends to results: conclusion uniqueness in DPLs corresponds to type uniqueness theorems in type systems, and so on.

However, in the case of DPLs this correspondence is only a similarity, *not* an isomorphism. This is so even in the particularly simple case of $\mathcal{NDL}$. The reason is that $\mathcal{NDL}$ has no variables, and hence infinitely many distinct terms of the typed λ-calculus collapse to the same DPL proof, destroying the bijection. For instance, consider the two terms

$$\lambda\, x : A\, .\, \lambda\, y : A\, .\, x \tag{1.14}$$

and

$$\lambda\, x : A \,.\, \lambda\, y : A \,.\, y \tag{1.15}$$

These are two distinct terms (they are not alphabetically equivalent), and under the Curry-Howard isomorphism they represent two distinct proofs of $A \Rightarrow A \Rightarrow A$. To see why, one should keep in mind Heyting's constructive interpretation of conditionals [4]. In 1.14, we are given a proof x of A, then we are given another proof y of it, and we finally decide to *use the first proof*, which might well have significant computational differences from the second one (e.g., one might be much more efficient than the other). By contrast, in 1.15 we are given a proof of A, then another proof of it, and finally we decide to *use the second proof*, disregarding the first. But in mathematical practice these two proofs would—and usually should—be conflated, because one *postulates propositions, not proofs of propositions*. The customary mode of hypothetical reasoning is

"Assume that A holds. Then $\cdots A \cdots$."

not

"Assume we have a proof Π of A. Then $\cdots \Pi \cdots$."

Clearly, to assume that something is true is not the same as to assume that we have a proof of it. Constructivists will argue against such a distinction, but regardless of whether or not the distinction is philosophically justified, it is certainly common in mathematical practice.

DPLs reflect mathematical practice more closely. Both 1.14 and 1.15 get conflated to the single deduction

$$\begin{array}{l}\textbf{assume } A \textbf{ in} \\ \quad \textbf{assume } A \textbf{ in} \\ \quad\quad \textbf{claim } A\end{array} \tag{1.16}$$

By "common mathematical practice" we mean that if we ask a random mathematician or computer scientist to prove the tautology $A \Rightarrow A \Rightarrow A$, the odds are that we will receive something like 1.16 as an answer, not 1.14 or 1.15.

The difference is not accidental. It reflects a deep divergence between DPLs and Curry-Howard systems on a foundational issue: constructivism. In a DPL one only cares about truth, or more narrowly, about what is in the assumption base. On the other hand, logical frameworks [5, 3] based on the Curry-Howard isomorphism put the focus on *proofs* of propositions rather than on propositions *per se*. This is due to the constructive bent of these systems, which dictates that a proposition should not be supposed to hold unless we have a proof for it. There are types, but more importantly, there are variables that have those types, and those variables range over proofs. A DPL such as $\mathcal{NDL}$ stops at types—there are no variables. Another way to see this is to consider the difference between assumption bases and type environments: an assumption base is simply a set of propositions (types), whereas a type environment is a set of *pairs* of variables and types. Pushing the correspondence, when we write in $\mathcal{NDL}$

$$\begin{array}{l}\textbf{assume } \textbf{false} \wedge \textbf{true} \textbf{ in} \\ \quad \textbf{left-and } \textbf{false} \wedge \textbf{true}\end{array}$$

it is as if we are saying, in the λ-calculus,

$$\lambda\, \mathbf{int} \times \mathbf{bool} \,.\, \mathit{left}\ (\mathbf{int} \wedge \mathbf{bool})$$

which is nonsensical. In a way, type-α DPLs such as $\mathcal{NDL}$ compute with types, i.e., with constants. There are no variables, and hence there is no meaningful notion of abstraction.

By avoiding the explicit manipulation of proofs, DPLs are more parsimonious: an inference rule R is applied directly to propositions $P_1, \ldots, P_n$, not to *proofs* of propositions $P_1, \ldots, P_n$; a hypothetical deduction assumes simply that we are given a hypothesis P, not that we are given *a proof* of P; and so on. Shaving away all the talk about proofs results in shorter and cleaner deductions. Curry-Howard systems have more populated ontologies because they explicitly posit proof entities. But for most *practical* purposes, the fact that both $\lambda\, x : A \,.\, \lambda\, y : A \,.\, x$ and $\lambda\, x : A \,.\, \lambda\, y : A \,.\, y$ would exist and be considered distinct proofs of $A \Rightarrow A \Rightarrow A$ amounts to unnecessary multiplication of entities.

Another important difference lies in the representation of syntactic categories such as propositions, proofs, etc. Curry-Howard systems typically rely on higher-order abstract syntax for representing such objects. While this has some advantages (most notably, alphabetically identical expressions at the object level are represented by alphabetically identical expressions at the meta level, and object-level syntactic operations such as substitution become reduced to β-reduction at the meta level), it also has serious drawbacks: higher-order expressions are less readable and writable, they are considerably larger on account of the extra abstractions and the explicit presence of type information, and complicate issues such as matching, unification, and reasoning by structural induction. A more detailed discussion of these issues, along with relevant examples, can be found elsewhere [2].

1.5 Pure and augmented DPLs

Observe that the grammar of $\mathcal{NDL}$ proofs (1.5) is a proper extension of the corresponding grammar of $\mathcal{PAR}$ (1.1): the first two productions, rule applications and compositions, are the same in both languages. DPLs in which deductions are generated by grammar 1.1, i.e., in which every deduction is either a rule application or a composition, are called *pure*. DPLs such as $\mathcal{NDL}$ that feature additional syntactic forms for proofs, over and above rule applications and compositions, are called *augmented*. Formally, these correspond to pure and augmented $\lambda\phi$ systems [2], respectively.

Many logics are sufficiently simple so that every proof can be depicted as a tree, where leaves correspond to premises and internal nodes correspond to rule applications, with the node at the root producing the ultimate conclusion. Pure DPLs capture precisely that class of logics, albeit with greater compactness and clarity than trees owing to the use of assumption bases and the cut (whereas trees, being acyclic, duplicate a lot of structure; see Chapter 7 of "Denotational Proof Languages" [2]). Most sophisticated logics, however, fall outside this category because they employ inference mechanisms such as hypothetical and parametric reasoning that cannot be captured by simple inference rules that just take a number of propositions as premises and return another proposition as the conclusion.

Such logics need to be formalized as augmented DPLs, with special proof forms that capture those modes of inference that are difficult or impossible to model with simple inference rules. Loosely speaking, special forms are needed whenever the mere presence of some premise Q in the assumption base is not sufficient to warrantee the desired conclusion, and additional constraints on *how* that premise was derived need to be imposed. In a sense, such modes of reasoning can only be captured by "higher-order" inference rules whose premises are not simple propositions but rather evaluation judgments themselves. In DPLs this is typically handled by introducing a recursive special form "**kwd** $\cdots$ **in** D", flagged by some keyword token **kwd** and having an arbitrary deduction D as its *body*, where D is intended to produce the premise in question, Q. Then when we come to state the formal semantics of this form, we impose the necessary caveats on how Q should be derived by appropriately constraining the recursive evaluation of the body D. Take hypothetical reasoning as an example. If we want to establish a conditional $P \Rightarrow Q$ in some β, knowing that P or Q or any

combination thereof is in β will not help much. What we need to know is that Q is derivable from β *and* the hypothesis P. Therefore, hypothetical reasoning cannot be captured by a simple inference rule and we are led to introduce the form "**assume** P **in** D", which can derive the desired conclusion $P \Rightarrow Q$ in some β only provided that the judgment $\beta \cup \{P\} \vdash D \rightsquigarrow Q$ holds.

As another example, consider the necessitation rule of modal logic: in order to infer $\Box P$ we need to know not just that P is in the assumption base, but rather that P is a tautology, i.e., derivable from the empty assumption base. So in a DPL formulation of modal logic we typically model necessitation inferences with a special form "**nec** D", whose semantics are succinctly stated as follows:

$$\frac{\emptyset \vdash D \rightsquigarrow P}{\beta \vdash \textbf{nec}\ D \rightsquigarrow \Box P}$$

As a final example, consider the parametric reasoning of universal generalization rules in predicate logic, where in order to infer the conclusion $(\forall x)\, P$ we need to know not simply that P is in the assumption base, but rather that it is derivable from an assumption base that does not contain any free occurrences of x. We thus introduce a special form "**generalize-over** x **in** D" with the following semantics:

$$\frac{\beta \vdash D \rightsquigarrow P}{\beta \vdash \textbf{generalize-over}\ x\ \textbf{in}\ D \rightsquigarrow (\forall x)\, P} \quad \text{provided } x \notin FV(\beta)$$

In general, the syntactic specification of DPL deductions via recursive abstract grammars $D ::= \cdots D \cdots$ in tandem with their inductive evaluation semantics based on assumption-base judgments make it easy to introduce such recursive syntax forms and to specify their semantics perspicuously.

1.6 Implementing DPLs

There are two choices for implementing a DPL L. One is to implement it directly from scratch. The other is to leverage an existing implementation of a sufficiently rich type-ω DPL L' and implement L within L'. In this section we illustrate both alternatives by offering two implementations of $\mathcal{PAR}$, one from scratch and one in Athena, a powerful type-ω DPL.

Implementing a DPL from scratch

Implementing a type-α DPL from scratch is less difficult than it sounds. A complete implementation of the abstract syntax and semantics of $\mathcal{PAR}$ appears in Figure 1.7 in less than one page of SML code. Typically, the following components must be built:

1. An abstract syntax module that models the grammatic structure of propositions and proofs by appropriate data structures.

2. A parser that takes text input and builds an abstract syntax tree. (This of course presupposes that we have settled on a concrete syntax choice.)

3. A module that implements assumption bases. As an abstract data type, assumption bases are collections of propositions which, at a minimum, must support look-ups and insertions. They can be implemented with varying degrees of efficiency as simple lists, balanced search trees, etc; and they can be parameterized over propositions for reusability. In the interest of simplicity, our implementation here is rather naive: it uses simple lists.

```
(* ABSTRACT SYNTAX *)

structure AbSyntax =
struct

datatype prop = even of int | odd of int;

fun propToString(even(n)) = "Even("^Int.toString(n)^")"
  | propToString(odd(n)) = "Odd("^Int.toString(n)^")";

datatype rule = zeroEven | oddNext | evenNext;

datatype proof = ruleApp of rule * prop list | comp of proof * proof;

end;

(* ASSUMPTION BASES *)

structure AssumBase =
struct

type assum_base = AbSyntax.prop list;

val empty_ab = [];

fun member(P,L:assum_base) = List.exists (fn Q => Q = P) L;

val insert = op::;

end;

(* SEMANTICS *)

structure Semantics =
struct

exception Error of string;

structure A = AbSyntax;

fun eval (A.ruleApp (A.zeroEven,[])) ab = A.even(0)
  | eval (A.ruleApp (A.oddNext,[premise as A.even(n)])) ab =
       if AssumBase.member(premise,ab) then A.odd(n+1)
       else raise Error("Invalid application of odd-next")
  | eval (A.ruleApp(A.evenNext,[premise as A.odd(n)])) ab =
       if AssumBase.member(premise,ab) then A.even(n+1)
       else raise Error("Invalid application of even-next")
  | eval (A.ruleApp _) _ = raise Error("Ill-formed deduction")
  | eval (A.comp(D1,D2)) ab = eval D2 (AssumBase.insert(eval D1 ab,ab));

end;
```

Figure 1.7: SML implementation of $\mathcal{PAR}$.

4. A module that implements the evaluation semantics of the DPL. This is the module responsible for implementing the DPL interpreter, i.e., the proof checker: a procedure that will take the abstract syntax tree of a proof and an assumption base and will either produce a proposition, validating the proof, or generate an error, rejecting the proof.

The SML structures in Figure 1.7 implement all of the aforementioned modules for $\mathcal{PAR}$, except for the parser. With automated tools such as Lex and Yacc (or in our case, ML-Lex and ML-Yacc), building lexers and parsers is quite straightforward. For $\mathcal{PAR}$, appropriate ML-Lex and ML-Yacc files could be put together in less than one page of combined code. In total, our entire implementation of

```
(structure Nat
  zero
  (succ Nat))

(declare Even (-> (Nat) Boolean))

(declare Odd (-> (Nat) Boolean))

(primitive-method (zero-even)
  (Even zero))

(primitive-method (odd-next P)
  (match P
    ((Even n) (check ((holds? P) (Odd (succ n)))))))

(primitive-method (even-next P)
  (match P
    ((Odd n) (check ((holds? P) (Even (succ n)))))))
```

Figure 1.8: Athena implementation of $\mathcal{PAR}$.

the language would be roughly two pages of code. (Of course the code would grow if we decided to add more bells and whistles, e.g., positional error checking.) $\mathcal{NDL}$ can likewise be implemented in no more than 2-3 pages of code.

Implementing a DPL in another DPL

A pure type-α DPL L, i.e., one in which every proof is a rule application or a composition, can be expediently implemented in a DPL such as Athena. Specifically:

1. The propositions of L are represented by Athena propositions, by introducing appropriate predicate symbols (a predicate symbol in Athena is just a function symbol whose range is `Boolean`).
2. Each primitive inference rule of L is modelled by an Athena `primitive-method`.
3. The deductions of L are represented by Athena deductions. In particular, rule applications in L become primitive method applications in Athena, and compositions in L become Athena compositions.

In the case of $\mathcal{PAR}$ this process is illustrated in Figure 1.8.

1. First, we must represent the propositions of $\mathcal{PAR}$; we must be able to make assertions such as *Even*(14) and *Odd*(85) in Athena. To that end, we introduce two relation symbols `Even` and `Odd`, each of which is predicated of a natural number. Now we can write down propositions such as `(Even zero)`, `(Odd (succ zero))`, and so on.
2. Second, we must model the inference rules of $\mathcal{PAR}$. Accordingly, we introduce a nullary primitive method `zero-even` and two unary primitive methods `even-next` and `odd-next`. The `zero-even` method simply outputs the conclusion `(Even zero)` whenever it is called, regardless of the contents of the assumption base. The `odd-next` method takes a proposition P of the form `(Even` n`)` and checks to ensure that P is in the current assumption base [4] (an error occurs if the given P

[4] The Athena construct `(check ((`E_1 E'_1`)` $\cdots$ `(`E_n E'_n`)))` is similar to Scheme's `cond`. Each E_i is evaluated in turn until some E_j produces `true`, and then the result of E'_j becomes the result of the whole `check`. It is an error if no E_i evaluates to `true`. Further, the top-level function `holds?` takes a proposition P and returns `true` or `false` depending on whether or not P is in the current assumption base.

is not of the form `(Even` n`)`). If the premise `(Even` n`)` holds, then the conclusion `(Odd (succ n))` is returned; otherwise an error occurs (by the semantics of `check`, since we do not list any other alternatives). Finally, `even-next` captures the **even-next** rule in a similar manner.

We can now map $\mathcal{PAR}$ proofs into Athena proofs. An application such as **even-next** $Odd(1)$ becomes a method application `(!even-next (Odd (succ zero)))`; while a $\mathcal{PAR}$ composition turns into an Athena composition using `dbegin` or `dlet`—or perhaps even just a nested method application. For instance, here is the Athena counterpart of the $\mathcal{PAR}$ proof of $Even(2)$ given in page 4:

```
(dbegin
  (!zero-even)
  (!odd-next  (Even zero))
  (!even-next (Odd (succ zero))))
```

The point here is that we do not have to choose a concrete syntax for deductions—we simply use Athena's syntax. Thus we do not have to write any code for lexical analysis and parsing. More importantly, we do not have to implement assumption bases, or take care of the semantics of compositions, or even implement a proof interpreter at all, because we get all that from Athena for free. We do not even have to handle error checking!

In addition, we can take advantage of Athena's versatility in structuring proofs and present our deductions in whatever style better suits our needs or taste. For instance, instead of using `dbegin` for composition, we can use `dlet`, which offers naming as well composition:

```
(dlet ((P0 (!zero-even))
       (P1 (!odd-next P0)))
  (!even-next P1))
```

For enhanced readability, we may use conclusion-annotated style:

```
(dlet ((P0 ((Even zero) BY (!zero-even)))
       (P1 ((Odd (succ zero)) BY (!odd-next P0))))
  ((Even (succ (succ zero))) BY (!even-next P1)))
```

Or if we want compactness, we can express the whole proof with just one nested method call:

```
(!even-next (!odd-next (!zero-even))).
```

The semantics of Athena ensure that the assumption base is properly threaded in every case.

A second alternative for an Athena implementation is to model the inference rules of L not by introducing new primitive Athena methods, but rather by postulating axioms. Theorems could then be derived using Athena's own primitive methods, namely, the customary introduction and elimination rules of first-order logic: universal instantiation, modus ponens, and so forth.

That amounts to casting L as a first-order theory. Take $\mathcal{PAR}$ as an example. After defining `Nat` and declaring the relation symbols `Even` and `Odd`, we can model the three rules of $\mathcal{PAR}$ by the following axioms:

```
(define zero-even-axiom (Even zero))

(define even-next-axiom
  (forall ?n
    (if (Odd ?n)
        (Even (succ ?n)))))
```

```
(define odd-next-axiom
  (forall ?n
    (if (Even ?n)
        (Odd (succ ?n)))))

(assert zero-even-axiom even-next-axiom odd-next-axiom)
```

The effect of the last assertion is to add the three axioms to the top-level assumption base. We can now derive the above theorem `(Even (succ (succ zero)))` with classical first-order reasoning:

```
(dlet ((P0 (!claim zero-even-axiom))
       (P1 (!mp (!uspec odd-next-axiom zero) P0)))
  (!mp (!uspec even-next-axiom (succ zero)) P1))
```

In fact with this approach we can use Athena to implement not just pure DPLs, but any logic L that can be formulated as a multi-sorted first-order theory. Conditionals $P \Rightarrow Q$ in L will be introduced by Athena's `assume` construct, universal quantifications $(\forall\, x : S)\, P$ will be introduced by Athena's `pick-any` construct, and so on. A great many logics can be formulated as multi-sorted first-order theories,[5] so this makes Athena an expressive logical framework. And, because Athena is a type-ω DPL, it offers proof methods as well, which means that it can be used not only for writing proofs down and checking them, but also for theorem proving—discovering proofs via search. The advantage of writing a theorem prover as an Athena method rather than as a conventional algorithm is that it is guaranteed to produce sound results, by virtue of the semantics of methods. Moreover, evaluating any method call can automatically produce a *certificate* [1]: an expanded proof that uses nothing but primitive methods. This greatly minimizes our trusted base, since we no longer have to trust the proof search; all we have to trust is our primitive methods.

1.7 Conclusions

This paper introduced type-α DPLs. We discussed some general characteristics of DPLs and illustrated with two concrete examples, $\mathcal{PAR}$ and $\mathcal{NDL}$. We demonstrated that type-α DPLs allow for readable, writable, and compact proofs that can be checked very efficiently. We introduced several special syntactic forms for proofs, such as compositions and hypothetical deductions, and showed how the abstraction of assumption bases allows us to give intuitive evaluation semantics to these syntactic constructs, in such a way that evaluation becomes tantamount to proof checking. Assumption-base semantics were also seen to facilitate the rigorous analysis of DPL proofs.

We have briefly contrasted DPLs with typed logical frameworks, and we have seen that the Curry-Howard isomorphism does not apply to DPLs: there is no isomorphism between a typed λ-calculus and a type-α DPL because such DPLs do not have variables. This disparity can be traced to a foundational difference on the issue of constructivism. In a sense, the main denotable values of a typed Curry-Howard logic are proofs of propositions (e.g., the variables of interest range over proofs), whereas in a DPL the denotable values are simply propositions. By doing away with the layer of explicit proof manipulation, DPLs allow for shorter, cleaner, and more efficient deductions. We also argued that readability and writability are enhanced by the fact that DPLs use first-order representations of syntactic categories such as propositions and proofs, whereas Curry-Howard systems typically rely on higher-order abstract syntax, which tends to be notationally cluttered and unintuitive.

[5] Peano arithmetic with induction, the theory of lists, vector spaces, partial orders and lattices, groups, programming language semantics, ZF set theory, the logics used for proof-carrying code [8], etc.

Differences with Curry-Howard frameworks become even more magnified in the setting of type-ω DPLs. Type-ω DPLs integrate computation and deduction. A DPL of this kind is essentially a programming language and a deduction language rolled into one. Although the two can be seamlessly intertwined, computations and deductions are syntactically distinct and have different assumption-base semantics. Type-ω DPLs allow for proof abstraction via *methods*. Methods are to deductions what procedures are to computations. They can be used to conveniently formulate derived inference rules, as well as arbitrarily sophisticated proof-search tactics. Soundness is guaranteed by the fundamental theorem of the $\lambda\phi$-calculus [2], which ensures that if and when a deduction returns a proposition, that proposition is derivable solely by virtue of the primitive rules and the propositions in the assumption base. Type-ω DPLs retain the advantages of type-α DPLs for proof presentation and checking, while offering powerful theorem-proving capabilities in addition.

For instance, in the implementation of $\mathcal{PAR}$ in Athena, we cannot only write down type-α proofs such as those in page 25, but we can also implement a *proof method*, call it `infer-parity`, that takes an arbitrary number n and proves a theorem of the form `(Even` n`)` or `(Odd` n`)`:

```
(define (infer-parity n)
  (dmatch n
    (zero (!zero-even))
    ((succ k) (dlet ((P (!infer-parity k)))
                (dmatch P
                  ((Even _) (!odd-next  P))
                  ((Odd _)  (!even-next P)))))))
```

This method will deduce the parity of any given number n, using nothing but the three primitive methods `zero-even`, `odd-next`, and `even-next`.

The versatile proof-search mechanisms afforded by type-ω DPLs make them ideal vehicles for *certified computation* [1], whereby a computation does not only produce a result r, but also a correctness certificate, which is a formal proof that r is correct.

We have also discussed implementation techniques for DPLs. We illustrated how to implement a type-α DPL from scratch by offering a sample implementation of $\mathcal{PAR}$ in SML. We also presented ways of implementing such a DPL within a more powerful DPL that has already been implemented, and illustrated that approach by expressing $\mathcal{PAR}$ in Athena. Specifically, we considered two distinct ways of implementing a DPL L in Athena: by casting the inference rules of L as primitive methods of Athena, a technique that is readily applicable to pure DPLs; and by formulating L as a multi-sorted first-order Athena theory, a technique that is applicable to a wide variety of first-order DPLs.

Bibliography

[1] K. Arkoudas. Certified Computation. MIT AI memo 2001-07.

[2] K. Arkoudas. Denotational Proof Languages. PhD thesis, MIT, 2000.

[3] N. G. De Brujin. The Automath checking project. In P. Braffort, editor, *Proceedings of Symposium on APL*, Paris, France, December 1973.

[4] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[5] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[6] W. A. Howard. The formulae-as-types notion of construction. In J. Hindley and J. R. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms*, pages 479–490. Academic Press, 1980.

[7] G. Kahn. Natural semantics. In *Proceedings of Theoretical Aspects of Computer Science*, Passau, Germany, February 1987.

[8] G. Necula and P. Lee. Proof-carrying code. Computer Science Technical Report CMU-CS-96-165, CMU, September 1996.

[9] M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proc. Int. Conf. Log. Prog. Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, 1992.

[10] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, Cambridge, England, 1996.