Type-omega DPLs

Konstantine Arkoudas

AI Memo 2001-027 October 2001

Abstract

Type-ω DPLs (Denotational Proof Languages) are languages for proof presentation and search that offer strong soundness guarantees. LCF-type systems such as HOL offer similar guarantees, but their soundness relies heavily on static type systems. By contrast, DPLs ensure soundness dynamically, through their evaluation semantics; no type system is necessary. This is possible owing to a novel two-tier syntax that separates deductions from computations, and to the abstraction of assumption bases, which is factored into the semantics of the language and allows for sound evaluation.

Every type-ω DPL properly contains a type-α DPL, which can be used to present proofs in a lucid and detailed form, exclusively in terms of primitive inference rules. Derived inference rules are expressed as user-defined *methods*, which are "proof recipes" that take arguments and dynamically perform appropriate deductions. Methods arise naturally via parametric abstraction over type-α proofs. In that light, the evaluation of a method call can be viewed as a computation that carries out a type-α deduction. The type-α proof "unwound" by such a method call is called the *certificate* of the call. Certificates can be checked by exceptionally simple type-α interpreters, and thus they are useful whenever we wish to minimize our trusted base.

Methods are statically closed over lexical environments, but dynamically scoped over assumption bases. They can take other methods as arguments, they can iterate, and they can branch conditionally. These capabilities, in tandem with the bifurcated syntax of type-ω DPLs and their dynamic assumption-base semantics, allow the user to define methods in a style that is disciplined enough to ensure soundness yet fluid enough to permit succinct and perspicuous expression of arbitrarily sophisticated derived inference rules.

We demonstrate every major feature of type-ω DPLs by defining and studying $\mathcal{NDL}_0^\omega$, a higher-order, lexically scoped, call-by-value type-ω DPL for classical zero-order natural deduction—a simple choice that allows us to focus on type-ω syntax and semantics rather than on the subtleties of the underlying logic. We start by illustrating how type-α DPLs naturally lead to type-ω DPLs by way of abstraction; present the formal syntax and semantics of $\mathcal{NDL}_0^\omega$; prove several results about it, including soundness; give numerous examples of methods; point out connections to the $\lambda\phi$-calculus, a very general framework for type-ω DPLs; introduce a notion of computational and deductive cost; define several instrumented interpreters for computing such costs and for generating certificates; explore the use of type-ω DPLs as general programming languages; show that DPLs do not have to be type-less by formulating a static Hindley-Milner polymorphic type system for $\mathcal{NDL}_0^\omega$; discuss some idiosyncrasies of type-ω DPLs such as the potential divergence of proof checking; and compare type-ω DPLs to other approaches to proof presentation and discovery. Finally, a complete implementation of $\mathcal{NDL}_0^\omega$ in SML-NJ is given for users who want to run the examples and experiment with the language. (Note: accompanying software for this paper can be found at `www.ai.mit.edu/projects/dynlangs/dpls/omega`.)

"*A notation is good for what it leaves out.*"
J. Stoy

1.1 Introduction

Extending type-α DPLs

Type-α DPLs were introduced in an earlier paper [5] and were shown to be simple and efficient. These advantages do not come for free. Perhaps the most serious limitation of type-α DPLs is the lack of an abstraction mechanism. In particular, there is no parametric abstraction: there is no way to abstract a given proof over one or more concrete objects in order to formulate a "proof recipe" that can be reused in many different situations by supplying different values for the parameters. This ability is very important in proof engineering for the same reasons that procedural abstraction is important

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**-elimimation)

Figure 1.1: Primitive inference rules

in programming: task decomposition and complexity management. It is particularly important for expressing so-called derived inference rules.

Consider, for example, the type-α DPL $\mathcal{NDL}_0$ [5], for classical natural deduction:

$$D ::= \mathit{Prim\text{-}Rule}\ P_1, \ldots, P_n \mid D_1; D_2 \mid \textbf{assume}\ P\ \textbf{in}\ D \tag{1.1}$$

where propositions $P, Q, \ldots$ are generated by the following abstract grammar:

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

with $A, B, C, \ldots$ ranging over some unspecified collection of atoms and *Prim-Rule* as specified in Figure 1.1. Now consider a proof of the tautology $A \Rightarrow \neg\neg A$, for some atom A:[1]

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

It is clear here that the identity of A is immaterial. The deduction will succeed even if we replace the atom A by an arbitrary proposition P. Accordingly, we should be able to abstract this deduction over A in order to obtain a derived inference rule, call it *foo*, that can be applied to any given proposition. For instance, applying *foo* to $A \vee B$ would *derive* the proposition $(A \vee B) \Rightarrow \neg\neg(A \vee B)$.

We might thus proceed to introduce an abstraction operator, call it ϕ, which we can use to define *foo* as follows:

foo $= \phi\, P\,.$ **assume** P **in**
                suppose-absurd $\neg P$ **in**
                  absurd $P, \neg P$

[1]Recall the desugaring of **suppose-absurd** in terms of **assume**, primitive rules, and deduction composition [5].

The letter P is used here as a parameter; it does not designate any one specific proposition. In general, we can use ϕ with any number of parameters $P_1, \ldots, P_n$, arriving at *methods*[2] of the form $\phi\, P_1, \ldots, P_n \,.\, D$, where $P_1, \ldots, P_n$ are the parameters and D is the body of the method. For now the term "method" can be understood simply as a shorthand for "derived inference rule", although later we will see that methods are more general and powerful than what are commonly understood as derived inference rules.

Once we have a method we can *apply* it to appropriate arguments. That will simply evaluate the body of the method with the formal parameters replaced by the actual arguments. For instance, applying *foo* to the argument $A \vee B$ will evaluate the body of *foo* with $A \vee B$ in place of P, thereby deriving the conclusion $(A \vee B) \Rightarrow \neg\neg(A \vee B)$.

This all sounds fairly innocuous and reasonable, but in fact a slew of difficult questions are immediately raised. The setting of type-α DPLs was very simple: we only had one syntactic category, deductions, and it was very clear what deductions were meant to do: derive propositions. Now we propose to have methods as well. But a method by itself is not a deduction; it's a deduction recipe. A deduction is what one gives in order to establish a proposition. A method does not establish any one proposition. It is the *application* of a method that derives a proposition, and indeed we will see that method applications will be regarded as deductions. Method themselves, however, are not deductions. But if they are not deductions, what are they? Alternatively, what do methods *denote*? In type-α DPLs we know what the denotable values are: propositions. Every deduction denotes a proposition. But here things are not so simple. In the presence of naming, specifying clearly what the denotable values are is very important because it affects a whole range of issues. For instance, can a method take another method as a parameter? Can a method return a method? Also, can a method call itself? Does it make sense to admit recursive methods?

Note that if we treat methods simply as functions that return propositions, then we will need a static type system to weed out unsound methods. That approach—essentially the approach of LCF-type systems—is fundamentally incompatible with the dynamic character of assumption bases, which enforce soundness at evaluation time. If we want a smooth generalization of type-α DPLs that allows for abstraction and proof search while conservatively extending the assumption-base semantics of type-α DPLs, we must look beyond type systems.

There are many more questions. Consider, for example, a method M of the following form:

$$M = \phi\, P, Q \,.\, \textbf{assume}\ P \vee Q\ \textbf{in}\ \cdots$$

that takes two propositions P and Q and performs some hypothetical deduction with the disjunction $P \vee Q$ as the hypothesis. How do we apply M to two arguments, say to A and $\neg B$? Proceeding as usual by way of substitution, we have to evaluate the body of M with A and $\neg B$ substituted in place of P and Q, respectively. The body of M is a hypothetical deduction, so the first thing we need to do is add the hypothesis to the assumption base. But the hypothesis is not quite there yet! We have to form it, by *applying* the disjunction operator $\vee$ to A and $\neg B$. But in what sense are we "applying" $\vee$? Certainly not in the same sense in which we apply methods, such as **modus-ponens**. The disjunction operator $\vee$ is not a method; it does not perform any kind of inference; it does not at all interact with the assumption base. But if $\vee$ is not a method, what is it?

The answer is that $\vee$ is a function. It is not a particularly interesting function—it is just a constructor that takes two already formed propositions P_1 and P_2 and builds their disjunction $P_1 \vee P_2$. But it is a function nevertheless—it performs *computation* rather than *inference.* It constructs both valid and invalid propositions, such as **false** $\vee$ **false**, without caring in the least about consistency

[2]We are not using the term "method" in any sense related to object-oriented programming.

with the assumption base. Indeed, as far as a function such as $\vee$ is concerned, the assumption base might as well not be there at all.

In fact, if we are to have even a remotely sophisticated proof language then we have to incorporate computation into our framework. Many times in informal proofs an author might say

$$\textbf{assume } f(\cdots) \textbf{ in } D$$

where $f(\cdots)$ is a *computation*, potentially of arbitrary complexity and duration, that will eventually produce some proposition P. It is only after $f(\cdots)$ returns P that we proceed in the usual manner by adding P to the assumption base and continuing with the evaluation of the body D. Likewise, it is common to express inference rule applications in the form

$$\textbf{modus-ponens } X, Y$$

where X and Y are descriptions of computational processes that run and eventually produce propositions of the form $P \Rightarrow Q$ and P, at which point we again proceed as usual by checking that both of these propositions are in the assumption base and, if so, returning Q.

So at this point we seem compelled to admit a different kind of evaluation into the picture; not deductive evaluation of the type-α sort, but rather conventional, algorithmic computation. The question now is how to formally distinguish these two notions, computation and deduction, while also integrating them in a sound and practical manner. What are their respective semantics, and how do they interact?

Other approaches such as that of LCF [16] or HOL [17], or of Boyer-Moore [7], do not explicitly distinguish between deduction and computation. In HOL, for instance, everything that a user writes down is ML code. The distinction is implicit, insofar it is only reflected in the type system, which puts deductions apart as special kinds of computations, namely, computations that return values of some type "theorem". DPLs, as we will shortly see, drive a much deeper wedge between the two notions. To begin with, computations and deductions are teased apart at the syntactic level. There are two distinct syntax layers, *expressions* for computation and *deductions* for inference. This syntactic separation is central to the approach of type-ω DPLS. Once we have sequestered the syntax of computations and deductions, their dynamic semantics can be formally distinguished in terms of how their evaluation interacts with the assumption base.

To make things concrete we will illustrate with $\mathcal{NDL}_0^\omega$, a type-ω DPL that extends $\mathcal{NDL}_0$. In the next three sections we will define the syntax and discuss the informal semantics of this language; present its formal semantics and prove some elementary results about its proof theory; and finally we will introduce an interpreter for it. We will subsequently extend $\mathcal{NDL}_0^\omega$ by introducing some additional syntax forms, thereby arriving at a larger language, $\overline{\mathcal{NDL}_0^\omega}$. The new forms will be theoretically superfluous, in the sense that they could in principle be defined as syntax sugar in terms of core $\mathcal{NDL}_0^\omega$ constructs. Nevertheless, they are very useful in practice and it is worthwhile to consider them as primitives—especially constructs such as pattern matching, whose desugarings are not trivial. Later on when we speak of "$\mathcal{NDL}_0^\omega$" we will not bother to specify whether we are referring to the kernel language of Section 1.2 or to the extended language of Section 1.6, since in most cases the distinction will be immaterial. When we have to differentiate the two, we will refer to the former as "core $\mathcal{NDL}_0^\omega$" and to the latter as "full $\mathcal{NDL}_0^\omega$", or as $\overline{\mathcal{NDL}_0^\omega}$.

Notation

We will write $\langle a, b\rangle$ for the ordered pair having a and b as its first and second elements, respectively; and $[a_1, \ldots, a_n]$ for the *list* of $a_1, \ldots, a_n$. For any given list L and element a, we write $a{::}L$ for the list obtained by prepending ("consing") a onto L; $\overleftarrow{L}$ for the reverse of L; and $\oplus$ for the list concatenation operator. For any set S, the expression $\mathcal{P}_\infty(S)$ will denote the set of all finite subsets of S.

In the sequel we will define several interpreters using functional pseudo-code, mostly in the style of SML [27]. As a notational convention made in the interest of brevity, we will sometimes write

$$\mathit{let}\ P = f(\cdots)\ \ \mathit{in}\ \ \cdots$$

to mean that the result of the call $f(\cdots)$ should be a *proposition* P. In general, whenever we bind the result of a function call $f(\cdots)$ to a variable x that has been designated to range over a certain set S, the binding should be understood to mean "If $f(\cdots)$ produces an element of S, let that element be x; otherwise generate an error." Similarly, we will write $\mathit{let}\ [V_1, \ldots, V_n] = f(\cdots)\ \ \mathit{in}\ \ \cdots$ to mean that the result of the function call $f(\cdots)$ should be a list of n values, to be named $V_1, \ldots, V_n$ within the body of the *let*; an error should occur if $f(\cdots)$ produces anything other than a list of n values. Occasionally, when it is more convenient to do so, our pseudocode will use the *where* construct (e.g., as found in Caml [13]) instead of *let*.

1.2 Syntax and informal semantics

We now assume that we are given a primitive syntactic domain of *identifiers* I. Further, we have a collection of *constants* c that are distinct from the identifiers. The language has two main syntactic categories: deductions D and expressions E. A *phrase* F is simply either a deduction or an expression. These domains are specified by the following grammar:

$$\begin{array}{rcl} D & ::= & \mathbf{!}E\ \ F_1 \cdots F_n \ \mid\ \mathbf{assume}\ E\ \mathbf{in}\ D \\ E & ::= & c \mid I \mid \phi\, I_1, \ldots, I_n \,.\, D \mid \lambda\, I_1, \ldots, I_n \,.\, E \mid E\ \ F_1 \cdots F_n \\ F & ::= & E \mid D \end{array}$$

As constants c we will take all the propositional atoms A along with **true** and **false**, the five propositional connectives, the propositional equality function $\equiv$, and the primitive methods shown in Figure 1.1. Accordingly,

$$c\ ::=\ A \mid \mathbf{true} \mid \mathbf{false} \mid \neg \mid \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \mid \equiv \mid \mathbf{claim} \mid \mathbf{modus\text{-}ponens} \mid \cdots$$

We will often write **mp**, **dn**, and **cd** as abbreviations for **modus-ponens**, **double-negation**, and **constructive-dilemma**, respectively.

Expressions of the form $\lambda\, I_1, \ldots, I_n \,.\, E$ and $\phi\, I_1, \ldots, I_n \,.\, D$ are called *functions* and *methods*, respectively. The identifiers $I_1, \ldots, I_n$ are the *formal parameters* of the respective function or method; they must all be distinct, i.e., $I_{j_1} \neq I_{j_2}$ whenever $1 \leq j_1 < j_2 \leq n$. We could have $n = 0$ formal parameters, and in that case, for readability purposes, the corresponding function or method will be written as $\lambda\,()\,.\,E$ or $\phi\,()\,.\,D$, instead of $\lambda\ .\,E$ or $\phi\ .\,D$.

Deductions of the form $\mathbf{!}E\ \ F_1 \cdots F_n$ are called *method applications*, or method calls. The exclamation mark is used simply to distinguish method applications from function applications, which are written as $E\ \ F_1, \ldots, F_n$. (Of course this is abstract syntax; if we try to use it as concrete syntax, to write down phrases as linear strings of symbols, then we will need some form of parenthesization to resolve parsing ambiguities. In what follows we will use parentheses liberally for that purpose.) The expression E is the method and the phrases $F_1, \ldots, F_n$ are the arguments of the application. Note that even though method applications are deductions, methods themselves are expressions.

Expressions and deductions are syntactically distinct, and this is a key feature of type-ω DPLs. It is immediately evident by simple inspection whether a given phrase F is an expression or a deduction. If we look at F as a parse tree, then all we have to do is check the root node: if it is either the exclamation mark or the keyword **assume**, then we have a deduction; otherwise F is an expression. Intuitively, expressions are intended to represent computations, while deductions represent logical demonstrations. In particular, *a deduction D will always produce a proposition*, if it produces anything at all. We thus preserve the viewpoint of type-α DPLs: deductions return propositions; and the proposition returned by some D will be viewed as the *conclusion* of D. To ensure soundness, we must also preserve the type-α guarantee that if D produces a conclusion P in some assumption base β, then P will in fact be a logical consequence of β. In symbols, if $\beta \vdash D \leadsto P$ then $\beta \models P$. We will prove that this holds soon.

By contrast, an expression can return anything—a proposition, a function, a method, etc. Furthermore, we will see that there are no constraints between the result of an expression E and the contents of the assumption base β. Evaluating E in a given β might well produce a proposition P that is inconsistent with β. But precisely because P is obtained from an *expression*, we make no guarantees about its soundness. We only make such guarantees about the results of deductions. Thus, loosely speaking, expressions are given free rein to compute in any way they see fit; they can ride roughshod over the assumption base. Deductions, on the other hand, are much more restricted; they have to play inside the sandbox of the assumption base.

Some additional noteworthy points:

- Methods are abstractions of logical derivations, and therefore the body of a method must be a deduction D, not an expression E. Thus something like $\phi\, g, x \,.\, g \;\; x$ is nonsense at the syntactic level. It is not a well-formed phrase because the body $g \;\; x$ is an expression, when it should instead be a deduction. By contrast, functions are abstractions of computations, and therefore the body of a function must be an expression E, not a deduction D. Hence, symmetrically, something like $\lambda\, M, P \,.\, \mathbf{!}M \;\; P$ is syntactically ill-formed.

- There is no special form for deduction composition. In $\mathcal{NDL}_0$, we had the form $D_1; D_2$, which was used to compose deductions so that the result of D_1 became available as a lemma within D_2. Here there is no need to posit such a mechanism as primitive. We can achieve the same effect by nesting method applications. For now we simply observe from the syntactic form of method calls, $\mathbf{!}E \;\; F_1 \cdots F_n$, that the arguments F_i of a method call are *phrases*, which means that they can be either expressions or deductions. If an argument F_i is a deduction, then the conclusion we obtain from it will be added to the assumption base *before* we apply the method E to the values of $F_1, \ldots, F_n$. This provision will be captured in the formal semantics by the so-called "cut rule", and will be used to implement inference composition. Consider, for instance, the unary method application

$$\mathbf{!double\text{-}negation} \;\; (\mathbf{!right\text{-}and} \;\; A \wedge \neg\neg B) \tag{1.2}$$

  whose only argument is itself a deduction, namely an application of **right-and** to $A \wedge \neg\neg B$. Suppose we are to evaluate 1.2 in the assumption base $\beta = \{A \wedge \neg\neg B\}$. First we evaluate the argument ($\mathbf{!right\text{-}and} \;\; A \wedge \neg\neg B$), obtaining the conclusion $\neg\neg B$. Then we *add* this conclusion to β and proceed to apply **double-negation** to $\neg\neg B$. Thus, the application of **double-negation** will take place in the assumption base $\{A \wedge \neg\neg B, \neg\neg B\}$ and will successfully produce the final conclusion B. Therefore, 1.2 is essentially equivalent to the $\mathcal{NDL}_0$ deduction

$$\mathbf{right\text{-}and} \;\; A \wedge \neg\neg B; \mathbf{double\text{-}negation} \;\; \neg\neg B.$$

- Functions and methods take multiple arguments rather than being curried. This is quite important for methods because a method application, being a deduction, is always expected to return a proposition; it cannot return another method. As we have already emphasized, this viewpoint of a deduction D as something that produces a proposition is central in DPLs. Of course this viewpoint could be preserved even if methods were unary, by packaging up all the necessary arguments of a method into a single list. However, methods would still have to return propositions, and hence currying would still not be possible in any meaningful sense.

- The fact that the arguments of a method application can be arbitrary phrases suggests that a method can take any type of value as input, including another method. So even though methods have to return propositions, they are higher-order in the sense that they can be given other methods as arguments.
- Expressions and deductions are generated by mutually recursive grammars. Accordingly,
  1. Expressions can appear within deductions. This was motivated in the introduction, where it was pointed out that oftentimes it is necessary to intersperse proof evaluation with conventional algorithmic computation. Observe, for instance, the syntax of hypothetical deductions: **assume** E **in** D. The hypothesis is an expression E, which can thus be an arbitrary computation that eventually produces a proposition; while the body is a deduction D comprising the *scope* of the hypothesis E. Note that this is an entirely different notion of scope from the lexical scope of identifiers induced by λ and ϕ.
  2. Deductions can appear within expressions. In particular, the arguments of a function call $E\ F_1, \ldots, F_n$ can be arbitrary phrases, thus possibly deductions. The reason is that deductions produce propositions, and a proposition can clearly be given as an argument to a function, regardless of whether it was logically derived or simply computed. Thus it is sensible to say, for instance, "apply the negation constructor to the result of the deduction D", i.e., $\neg\ D$, where $\neg$ takes an arbitrary proposition P and outputs $\neg P$. Since deductions that appear as arguments to function calls are used for computational rather than deductive purposes, we will refer to them as "phantom" deductions.

  Nevertheless, the interweaving of expressions and deductions is completely orthogonal. If an expression E does not contain any deductions, then it looks and behaves exactly like a conventional lambda-calculus expression. In addition, the evaluation of such an expression will incur zero run-time penalty because the assumption base β will never be touched: nothing will be looked up in β and nothing will be inserted in it. Conversely, if our deductions contain only trivial expressions in them, such as constants, then we are essentially barring computation and evaluation will proceed almost exclusively by applying primitive methods and manipulating the assumption base; we will then be reverting to type-α deductions. This will be clarified further when we come to discuss pure deductions and pure expressions.

Finally, free and bound occurrences of identifiers are defined as one would expect: in a method $\phi\, I_1, \ldots, I_n \,.\, D$, the abstraction operator ϕ binds $I_1, \ldots, I_n$ within D, just as λ binds $I_1, \ldots, I_n$ within E in a function $\lambda\, I_1, \ldots, I_n \,.\, E$. Specifically, the set of identifiers that have free occurrences in a phrase F, denoted $FV(F)$, is defined as follows:

$$\begin{array}{rcl}
FV(\mathbf{!}E\ F_1 \cdots F_n) & = & FV(E) \cup FV(F_1) \cup \cdots \cup FV(F_n) \\
FV(\mathbf{assume}\ E\ \mathbf{in}\ D) & = & FV(E) \cup FV(D) \\
FV(c) & = & \emptyset \\
FV(I) & = & \{I\} \\
FV(\phi\, I_1, \ldots, I_n \,.\, D) & = & FV(D) - \{I_1, \ldots, I_n\} \\
FV(\lambda\, I_1, \ldots, I_n \,.\, E) & = & FV(E) - \{I_1, \ldots, I_n\} \\
FV(E\ F_1, \ldots, F_n) & = & FV(E) \cup FV(F_1) \cup \cdots \cup FV(F_n)
\end{array}$$

Phrases which differ only in their bound identifiers are called *alphabetic variants*, and will be identified. That is, we will consider two phrases to be identical iff they are alphabetically convertible. For distinct identifiers $I_1, \ldots, I_k$, we write $F[F_1/I_1, \ldots, F_k/I_k]$ for the phrase obtained from F by simultaneously replacing every free occurrence of I_j by F_j, $i = 1, \ldots, k$. In general this replacement can result in an ill-formed phrase (consider for instance, $\lambda\, x\, .\, y[\mathbf{!both}\ \ A\ \ B/y]$), but in our discussion all replacements will be of the form $[E/I]$, which substitutes an expression (E) for another expression (I) and is thus always valid. Also, substitutions can result in variable capture, but since alphabetic variants are considered identical we can always preclude that outcome by renaming the bound identifiers of F.

1.3 Formal semantics

First we single out certain expressions as *propositions*:

$$P \ ::= \ A \mid \mathbf{true} \mid \mathbf{false} \mid \neg P \mid \wedge\ P_1\ P_2 \mid \vee\ P_1\ P_2 \mid\ \Rightarrow\ P_1\ P_2 \mid \Leftrightarrow P_1\ P_2 \tag{1.3}$$

Thus every proposition is an expression: either a constant (an atom A, or **true**, or **false**), or an application (of one of the five constructors to the appropriate number of propositions). We write $\mathbf{Prop}[\mathcal{NDL}_0^\omega]$ for the set of all such propositions. By an *assumption base* we will mean a finite set of such propositions. As a convention, we will often write propositions in infix notation, e.g. writing $A \wedge (B \Rightarrow C)$ instead of $\wedge\ A\ (\ \Rightarrow\ \ B\ C)$. We will also use infix syntax for the equality function $\equiv$, writing $P \equiv Q$ instead of $\equiv\ P\ \ Q$. We will continue to use the letters P and Q to designate propositions, and A, B and C for atoms.

Next we single out a larger set of expressions as *values*. Intuitively, a value V represents an expression which, for our purposes, cannot be simplified any further and may thus be regarded as atomic and indivisible. The process of evaluation that will be induced by our semantics can be viewed as one of mapping phrases to values.

$$V \ ::= \ c \mid P \mid \lambda\, I_1, \ldots, I_n\, .\, E \mid \phi\, I_1, \ldots, I_n\, .\, D$$

Thus an expression is a value iff it is either a constant, or a proposition, or a function, or a method. We will use the letter V to range over the set of values.

The semantics are given by rules that establish judgments of the form $\beta \vdash_{\mathcal{NDL}_0^\omega} F \rightsquigarrow V$, which can be read as "With respect to the assumption base β, phrase F evaluates to V", or "In β, F produces the value V", etc. (We will drop the subscript $\mathcal{NDL}_0^\omega$ from the symbol $\vdash_{\mathcal{NDL}_0^\omega}$ whenever it is easily deducible from the context or irrelevant.) The rules are divided into two groups, the core rules, shown in Figure 1.2, and the rules for the constants, shown in Figure 1.3. The latter depicts only a few sample rules for some of the primitive methods; the rules for the remaining primitive methods can be found elsewhere [5].

Most of the core rules are straightforward. Rule $[R_1]$ specifies the semantics of primitive method applications, i.e., those method applications $\mathbf{!}E\ \ F_1 \cdots F_n$ in which E produces a primitive method c (recall that primitive methods are represented as constants). This rule is meant to be used in combination with the constant rules that specify the semantics of the primitive methods. In particular, the last premise of $[R_1]$, namely $\beta \vdash \mathbf{!}c\ \ V_1 \cdots V_n \rightsquigarrow P$, will have to be established by a rule such as $[C_3]$ or $[C_4]$. Rule $[R_2]$ specifies the semantics of non-primitive method applications, when the expression E in $\mathbf{!}E\ \ F_1 \cdots F_n$ produces a method of the form $\phi\, I_1, \ldots, I_n\, .\, D$ rather than a primitive. Although this appears to be completely standard (evaluate the operator, evaluate the arguments, replace the formals by the actuals and evaluate the body), applications of this rule will usually be preceded by applications of $[R_7]$, which postulates that when an argument F_i is a deduction then its conclusion can

$$\frac{\beta \vdash E \leadsto c \quad \beta \vdash F_i \leadsto V_i \quad \beta \vdash \mathbf{!}c\ V_1 \cdots V_n \leadsto P}{\beta \vdash \mathbf{!}E\ F_1 \cdots F_n \leadsto P}\ [R_1]$$

$$\frac{\beta \vdash E \leadsto \phi\, I_1, \ldots, I_n \,.\, D \quad \beta \vdash F_i \leadsto V_i \quad \beta \vdash D[V_j/I_j] \leadsto P}{\beta \vdash \mathbf{!}E\ F_1 \cdots F_n \leadsto P}\ [R_2]$$

$$\frac{\beta \vdash E \leadsto P \quad \beta \cup \{P\} \vdash D \leadsto Q}{\beta \vdash \textbf{assume}\ E\ \textbf{in}\ D \leadsto P \Rightarrow Q}\ [R_3]$$

$$\frac{}{\beta \vdash V \leadsto V}\ [R_4]$$

$$\frac{\beta \vdash E \leadsto c \quad \beta \vdash F_i \leadsto V_i \quad \beta \vdash c\ V_1 \cdots V_n \leadsto V}{\beta \vdash E\ F_1 \cdots F_n \leadsto V}\ [R_5]$$

$$\frac{\beta \vdash E \leadsto \lambda\, I_1, \ldots, I_n \,.\, E' \quad \beta \vdash F_i \leadsto V_i \quad \beta \vdash E'[V_j/I_j] \leadsto V}{\beta \vdash E\ F_1 \cdots F_n \leadsto V}\ [R_6]$$

$$\frac{\beta \vdash D \leadsto P \quad \beta \cup \{P\} \vdash \mathbf{!}E\ \ \cdots P \cdots \leadsto Q}{\beta \vdash \mathbf{!}E\ \ \cdots D \cdots \leadsto Q}\ [R_7]$$

Figure 1.2: Core semantic rules.

be added to the assumption base before we proceed with the body D of the method. The examples will make this clear shortly. $[R_3]$ specifies the semantics of hypothetical deductions in the usual manner. Rule $[R_4]$ says that values are self-evaluating. Rules $[R_5]$ and $[R_6]$ are the analogues of $[R_1]$ and $[R_2]$ for primitive and non-primitive functions, respectively. These four rules together impose a call-by-value discipline: the arguments of method calls and function calls are fully evaluated before any substitution occurs. The most novel rule is the *cut rule* $[R_7]$, which is the one that allows for inference composition, whereby the conclusion of one deduction is added to the assumption base and serves as a premise for a subsequent deduction. This captures the transitivity of the deducibility relation.
As an example, let D be the deduction

assume $A \wedge B$ **in**
  !both (**!right-and** $A \wedge B$)
      (**!left-and** $A \wedge B$)

The derivation below establishes the judgment $\emptyset \vdash D \leadsto A \wedge B \Rightarrow B \wedge A$:

1. $\{A \wedge B, A\} \vdash$ **!right-and** $A \wedge B \leadsto B$	$[C_6]$
2. $\{A \wedge B, A, B\} \vdash$ **!both** $B\ A \leadsto B \wedge A$	$[C_4]$
3. $\{A \wedge B, A\} \vdash$ **!both** (**!right-and** $A \wedge B$) $A \leadsto B \wedge A$	1, 2, $[R_7]$
4. $\{A \wedge B\} \vdash$ **!left-and** $A \wedge B \leadsto A$	$[C_5]$
5. $\{A \wedge B\} \vdash$ **!both** (**!right-and** $A \wedge B$) (**!left-and** $A \wedge B$) $\leadsto B \wedge A$	4, 3, $[R_7]$

$$\frac{}{\beta \vdash P \equiv P \leadsto \textbf{true}}\ [C_1] \qquad \frac{}{\beta \vdash P \equiv Q \leadsto \textbf{false}}\ [C_2]$$
$$\text{whenever } P \neq Q$$

$$\frac{}{\beta \cup \{P\} \vdash \textbf{!claim}\ \ P \leadsto P}\ [C_3] \qquad \frac{}{\beta \cup \{P, Q\} \vdash \textbf{!both}\ \ P\ \ Q \leadsto P \wedge Q}\ [C_4]$$

$$\frac{}{\beta \cup \{P \wedge Q\} \vdash \textbf{!left-and}\ \ P \wedge Q \leadsto P}\ [C_5] \qquad \frac{}{\beta \cup \{P \wedge Q\} \vdash \textbf{!right-and}\ \ P \wedge Q \leadsto Q}\ [C_6]$$

$$\frac{}{\beta \cup \{\neg\neg P\} \vdash \textbf{!double-negation}\ \ \neg\neg P \leadsto P}\ [C_7]$$

$$\vdots$$

Figure 1.3: The semantics of constants.

6. $\emptyset \vdash A \wedge B \leadsto A \wedge B$ $[R_4]$
7. $\emptyset \vdash D \leadsto A \wedge B \Rightarrow B \wedge A$ 6, 5, $[R_3]$

As another example, consider the deduction

$$D = \mathbf{!}(\phi\, M \,.\, \mathbf{!}M\ \ \neg\neg A)\ \ \textbf{double-negation}.$$

We establish the judgment $\{\neg\neg A\} \vdash D \leadsto A$ as follows:

1. $\{\neg\neg A\} \vdash \phi\, M \,.\, \mathbf{!}M\ \ \neg\neg A \leadsto \phi\, M \,.\, \mathbf{!}M\ \ \neg\neg A$ $[R_4]$
2. $\{\neg\neg A\} \vdash$ **double-negation** $\leadsto$ **double-negation** $[R_4]$
3. $\{\neg\neg A\} \vdash$ **!double-negation** $\neg\neg A \leadsto A$ $[C_7]$
4. $\{\neg\neg A\} \vdash D \leadsto A$ 1, 2, 3, $[R_2]$

The next result can be proved directly on the basis of the formal semantics, or taken as a corollary of the existence of a deterministic interpreter $\mathcal{I}$ such that $\beta \vdash F \leadsto V$ iff $\mathcal{I}(F, \beta) = V$ (we will define such an interpreter in Section 1.8.1).

Theorem 1.1 *A phrase F evaluates to at most one value. That is, for all β, if $\beta \vdash F \leadsto V_1$ and $\beta \vdash F \leadsto V_2$ then $V_1 = V_2$.*

1.4 Proof theory and a first embedding of $\mathcal{NDL}_0$

The following lemma expresses a so-called "dilution" property: if a deduction D yields a conclusion P in some β, then it will continue to do so even if we "dilute" β with additional propositions.

Lemma 1.2 (Dilution Lemma) *If $\beta \vdash D \leadsto P$ then $\beta \cup \beta' \vdash D \leadsto P$.*

Proof: By strong induction on the length of the derivation of the judgment $\beta \vdash D \leadsto P$. ■

Let $\succcurlyeq\ \subseteq \mathcal{P}_\infty(\mathbf{Prop}[\mathcal{NDL}_0^\omega]) \times \mathbf{Prop}[\mathcal{NDL}_0^\omega]$ be an arbitrary binary relation. We will say that $\succcurlyeq$ is *Tarskian* iff it is:

- *reflexive*, i.e., $\beta \succcurlyeq P$ whenever $P \in \beta$;
- *monotonic*, i.e., $\beta_1 \cup \beta_2 \succcurlyeq P$ whenever $\beta_1 \succcurlyeq P$; and
- *transitive*, i.e., $\beta \succcurlyeq P_2$ whenever $\beta \succcurlyeq P_1$ and $\beta \cup \{P_1\} \succcurlyeq P_2$.

Furthermore, we will say that:

- $\succcurlyeq$ *respects* a primitive method M iff $\beta \succcurlyeq P$ whenever $\beta \vdash \mathbf{!}M \;\; V_1, \ldots, V_n \leadsto P$; and that
- $\succcurlyeq$ is *closed under hypothetical reasoning* iff $\beta \succcurlyeq P \Rightarrow Q$ whenever $\beta \cup \{P\} \succcurlyeq Q$.

The judgments $\beta \vdash D \leadsto P$ single out a derivability relation

$$\Vdash_{\mathcal{NDL}_0^\omega} \subseteq \mathcal{P}_\infty(\mathbf{Prop}[\mathcal{NDL}_0^\omega]) \times \mathbf{Prop}[\mathcal{NDL}_0^\omega]$$

as follows:

$$\beta \Vdash_{\mathcal{NDL}_0^\omega} P \quad \text{iff} \quad (\exists\, D)\, [\beta \vdash_{\mathcal{NDL}_0^\omega} D \leadsto P].$$

(When it is clear that we are talking about $\mathcal{NDL}_0^\omega$ we will simply write $\Vdash$ instead of $\Vdash_{\mathcal{NDL}_0^\omega}$.)

Lemma 1.3 *The relation* $\Vdash_{\mathcal{NDL}_0^\omega}$ *is Tarskian.*

Proof: Reflexivity follows by virtue of **claim**: if $P \in \beta$ then $\beta \vdash \mathbf{!claim}\; P \leadsto P$ and therefore $\beta \Vdash P$. Monotonicity will follow if we can show that $\beta \cup \beta' \vdash D \leadsto P$ whenever $\beta \vdash D \leadsto P$. This was established by the dilution lemma. For transitivity, suppose that $\beta \Vdash P_1$ and $\beta \cup \{P_1\} \Vdash P_2$, so that

$$\beta \vdash D_1 \leadsto P_1 \tag{1.4}$$

and

$$\beta \cup \{P_1\} \vdash D_2 \leadsto P_2 \tag{1.5}$$

for some D_1, D_2. Setting

$$D = \mathbf{!}(\phi\, I \,.\, D_2) \;\; D_1$$

for some I that does not occur in D_2, the following derivation establishes the judgment $\beta \vdash D \leadsto P_2$:

1. $\beta \cup \{P_1\} \vdash \phi\, I \,.\, D_2 \leadsto \phi\, I \,.\, D_2$	$[R_4]$
2. $\beta \cup \{P_1\} \vdash P_1 \leadsto P_1$	$[R_4]$
3. $\beta \cup \{P_1\} \vdash D_2[P_1/I] \leadsto P_2$	Assumption 1.5, since $D_2[P_1/I] = D_2$
4. $\beta \cup \{P_1\} \vdash \mathbf{!}(\phi\, I \,.\, D_2) \;\; P_1 \leadsto P_2$	1, 2, 3, $[R_2]$
5. $\beta \vdash \mathbf{!}(\phi\, I \,.\, D_2) \;\; D_1 \leadsto P_2$	Assumption 1.4, 4, $[R_7]$

It follows from $\beta \vdash D \leadsto P_2$ that $\beta \Vdash P_2$. ■

Theorem 1.4 $\Vdash_{\mathcal{NDL}_0^\omega}$ *is the least Tarskian relation that respects the primitive methods of* $\mathcal{NDL}_0^\omega$ *and is closed under hypothetical reasoning.*

Proof: That $\Vdash$ is Tarskian was shown by the preceding lemma. It is also straightforward to prove that $\Vdash$ respects the primitive methods of $\mathcal{NDL}_0^\omega$ and is closed under hypothetical reasoning. Thus here we only need to show that if $\beta \Vdash P$ then $\beta \succcurlyeq P$, for an arbitrary relation $\succcurlyeq$ that respects the primitive methods of $\mathcal{NDL}_0^\omega$ and is closed under hypothetical reasoning. Now the assumption $\beta \Vdash P$ means that there is a D such that

$$\beta \vdash D \rightsquigarrow P \tag{1.6}$$

and hence we may proceed by strong induction on the length n of the derivation of this judgment. When $n = 1$, 1.6 must be an instance of a primitive-method axiom (e.g., $[C_3]$), and in that case the result follows from the assumption that $\succcurlyeq$ respects every primitive method. For the inductive step, let $n > 1$ and assume that the result holds for all derivations of length less than n. We proceed by a case analysis of the rule that is used in the last line of the derivation of 1.6. There are five possibilities for what that rule can be: it is one of the primitive-method axioms; or else it is $[R_1]$; or $[R_2]$; or $[R_3]$; or $[R_7]$. No other rule could possibly be used to derive 1.6 for syntactic reasons: the judgment 1.6 relates a *deduction* D to a proposition P, whereas every rule other than the aforementioned relates an *expression* to a value. We consider each of the five cases in turn. In the first case the result again follows from the supposition that $\succcurlyeq$ respects all primitive methods. In the cases of $[R_1]$ and $[R_2]$ the result follows immediately from the inductive hypothesis. In the case of $[R_3]$ the inductive hypothesis yields $\beta \cup \{P\} \succcurlyeq Q$, and therefore $\beta \succcurlyeq P \Rightarrow Q$ follows from the assumption that $\succcurlyeq$ is closed under hypothetical reasoning. In the case of the cut rule, $[R_7]$, the inductive hypotheses entail $\beta \succcurlyeq P$ and $\beta \cup \{P\} \succcurlyeq Q$, so now the desired judgment $\beta \succcurlyeq Q$ follows from the transitivity of $\succcurlyeq$. This completes the case analysis and the inductive argument. ■

The soundness of $\mathcal{NDL}_0^\omega$ follows as a direct corollary of this theorem. Specifically, define $\models$ as the usual relation of logical implication, so that $\beta \models P$ holds iff every interpretation that satisfies each element of β also satisfies P. It is straightforward to show that $\models$ is Tarskian; that it respects the primitive methods of $\mathcal{NDL}_0^\omega$; and that it is closed under hypothetical reasoning. Hence Theorem 1.4 entails that $\beta \models P$ whenever $\beta \Vdash P$.

Corollary 1.5 (Soundness) *If* $\beta \Vdash_{\mathcal{NDL}_0^\omega} P$ *then* $\beta \models P$.

For completeness, we need only show how to embed $\mathcal{NDL}_0$ into $\mathcal{NDL}_0^\omega$. Since the former has already been shown to be complete, the completeness of the latter will follow by virtue of the embedding. We define a translation mapping $\mathcal{T}$ from $\mathcal{NDL}_0$ deductions to $\mathcal{NDL}_0^\omega$ deductions as follows:

$$\mathcal{T}[\![\mathit{Prim\text{-}Rule}\ P_1, \ldots, P_n]\!] = \mathbf{!}\,\mathit{Prim\text{-}Rule}\ P_1, \ldots, P_n \tag{1.7}$$

$$\mathcal{T}[\![\mathbf{assume}\ P\ \mathbf{in}\ D]\!] = \mathbf{assume}\ P\ \mathbf{in}\ \mathcal{T}[\![D]\!] \tag{1.8}$$

$$\mathcal{T}[\![D_1; D_2]\!] = \mathbf{!}\,(\phi\, I\, .\, \mathcal{T}[\![D_2]\!])\ \mathcal{T}[\![D_1]\!] \tag{1.9}$$

where the identifier I in 1.9 must not occur in D_2. Note that since propositions have the exact same abstract syntax in both languages, we may identify $\mathbf{Prop}[\mathcal{NDL}_0]$ with $\mathbf{Prop}[\mathcal{NDL}_0^\omega]$. Thus we do not bother to distinguish between a $\mathcal{NDL}_0$ proposition and its $\mathcal{NDL}_0^\omega$ counterpart, and this is why the variables $P_1, \ldots, P_n$ appear in the same capacity on both sides of the equation 1.7 (and likewise for the hypothesis P in 1.8). By extension, we will not distinguish between a $\mathcal{NDL}_0$ assumption base and its $\mathcal{NDL}_0^\omega$ counterpart. The following result proves the correctness of this desugaring:

Theorem 1.6 *If* $\beta \vdash_{\mathcal{NDL}_0} D \rightsquigarrow P$ *then* $\beta \vdash_{\mathcal{NDL}_0^\omega} \mathcal{T}[\![D]\!] \rightsquigarrow P$.

Proof: A straightforward induction on the length of the $\mathcal{NDL}_0$ derivation of $\beta \vdash D \rightsquigarrow P$. ■

Completeness is now immediate:

Corollary 1.7 (Completeness) *If* $\beta \models P$ *then* $\beta \Vdash_{\mathcal{NDL}_0^\omega} P$.

Proof: If $\beta \models P$ then there is a $\mathcal{NDL}_0$ deduction D such that $\beta \vdash_{\mathcal{NDL}_0} D \rightsquigarrow P$. By Theorem 1.6, $\beta \vdash_{\mathcal{NDL}_0^\omega} \mathcal{T}[\![D]\!] \rightsquigarrow P$, which is to say $\beta \Vdash_{\mathcal{NDL}_0^\omega} P$. ■

1.5 An interpreter

In this section we informally describe a substitution-based interpreter for evaluating a $\mathcal{NDL}_0^\omega$ phrase F in an assumption base β. This interpreter will be rigorously expressed in ML-pseudocode in Section 1.8.1, where it will also be extended to handle some new syntax forms to be introduced in Section 1.6. We will assume that we have an algorithm at our disposal for applying any primitive method or function to an arbitrary sequence of values in an arbitrary assumption base. The interpreter is syntax-driven; it proceeds in accordance with the syntactic structure of the input phrase F:

- If F is a method application $\mathbf{!}E\ F_1 \cdots F_n$, evaluate E in β to obtain some value V. Then initialize a set of lemmas β' to be the empty set $\emptyset$, and start evaluating each argument F_i in β, for $i = 1, \ldots, n$, to obtain values $V_1, V_2, \ldots, V_n$. If F_i is a deduction, so that the value V_i produced by it is some proposition P, then add P to the lemma set β'. We stress that this is only done for those arguments F_i that are, syntactically speaking, deductions. When we have evaluated every argument F_i, we proceed by a case analysis of the value V that we obtained from E:

  1. If V is a primitive method M, such as **modus-ponens**, then apply M to the values $V_1, \ldots, V_n$ in $\beta \cup \beta'$.
  2. If V is a method of the form $\phi\, I_1, \ldots, I_n \,.\, D$, then evaluate $D[V_1/I_1, \ldots, V_n/I_n]$ in $\beta \cup \beta'$.
  3. Otherwise report an error, since in that case E does not denote a method.

- If F is a deduction of the form **assume** E **in** D, evaluate E in β. If that eventually produces a proposition P, evaluate D in $\beta \cup \{P\}$ to obtain some conclusion Q and return the conditional $P \Rightarrow Q$ as the final result. If the evaluation of E in β produces something other than a proposition, generate an error.

- If F is a constant expression c, a function $\lambda\, I_1, \ldots, I_n \,.\, E$, or a method $\phi\, I_1, \ldots, I_n \,.\, D$, return F.

- If F is a function application $E\ F_1, \ldots, F_n$, evaluate E in β to get a value V from it. Then start evaluating each argument F_i in β, for $i = 1, \ldots, n$. Once every F_i has produced a value V_i, proceed by analyzing V:

  – If V is a primitive function f, then apply f to the arguments $V_1, \ldots, V_n$ in β.

  – If V is a function $\lambda\, I_1, \ldots, I_n \,.\, E$, evaluate $E[V_1/I_1, \ldots, V_n/I_n]$ in β.

  – Otherwise report an error, since in that case E does not represent a function.

The most noteworthy part of the interpreter involves the handling of the cut rule $[R_7]$. The cut rule is non-deterministic in that a method call $\mathbf{!}E\ F_1 \cdots F_n$ may have several deductive arguments, and in general we may have to "guess" which of those deductions are necessary in order for the application of E to succeed. For interpretation purposes, however, we would like to have a fixed evaluation strategy in order to avoid the need for backtracking. The simplest way to remove the nondeterminism associated with $[R_7]$ is to incorporate the conclusions of *all* arguments F_i that are deductions. This eliminates the need to backtrack and is sound owing to the dilution lemma, for if the application goes through with the help of *some* of the deductions from $F_1, \ldots, F_n$, then it will certainly go though with the help of all of them. Moreover, this scheme incurs zero cost because with a call-by-value strategy every argument F_i needs to be evaluated anyway before the application takes place, so the only additional work we have to do is trivial: keep track of the values (conclusions) of those arguments that are deductions, in order to later incorporate them in the assumption base in which the call is taking place.[3]

1.6 Additional syntax forms

As it stands, $\mathcal{NDL}_0^\omega$ is powerful but not quite usable. In that respect it is analogous to the λ-calculus, which is powerful but too austere to be useful for practical purposes. To remedy this, we need to introduce additional syntax forms that enrich the expressiveness of the language. There are two ways to do that. One is to introduce the desired new idioms as syntax sugar. The main advantage of this approach is that it facilitates the study of the language. For instance, if we need to prove that every expression has a certain nice property then we only need to consider core expressions, since every other type of expression is just an abbreviation—albeit a potentially long one—for some core expression. The other approach, which is the one we will take in this paper, is to introduce the new constructs as bona fide new primitives, with their own special semantics. This approach is a better reflection of practice, since languages are rarely implemented by desugaring all possible constructs into a very small core. Nevertheless, we stress that each of the constructs introduced below could be desugared in terms of core $\mathcal{NDL}_0^\omega$. For details we refer the reader to Chapter 8 of "Denotational Proof Languages" [4].

We obtain the extended language $\overline{\mathcal{NDL}_0^\omega}$ by augmenting the grammar of $\mathcal{NDL}_0^\omega$ as follows:

$$\begin{aligned} D \ ::= \ & \mathbf{!}E\ F_1 \cdots F_n \mid \mathbf{assume}\ E\ \mathbf{in}\ D \mid \mathbf{dlet}\ I = F\ \mathbf{in}\ D \mid \mathbf{dmatch}\ E\ (\pi_1?\ D_1) \cdots (\pi_n?\ D_n) \mid \\ & \mathbf{suppose\text{-}absurd}\ E\ \mathbf{in}\ D \mid E\ \mathbf{by}\ D \mid \mathbf{begin}\ D_1; \cdots; D_n\ \mathbf{end} \\ E \ ::= \ & c \mid I \mid \phi\, I_1, \ldots, I_n \,.\, D \mid \lambda\, I_1, \ldots, I_n \,.\, E \mid E\ F_1, \ldots, F_n \mid \mathbf{fix}\, I \,.\, E \mid \mathbf{let}\ I = F\ \mathbf{in}\ E \mid \\ & \mathbf{match}\ E\ (\pi_1?\ E_1) \cdots (\pi_n?\ E_n) \mid \mathbf{begin}\ E_1; \cdots; E_n\ \mathbf{end} \\ F \ ::= \ & E \mid D \end{aligned}$$

Note the bifurcated syntax: most of the new constructs have computational and deductive counterparts. Moreover, note that the body of a **let** is an expression, while the body of a **dlet** is a deduction. Similarly, in an expression of the form

$$\mathbf{match}\ E\ (\pi_1?\ E_1) \cdots (\pi_n?\ E_n)$$

the alternatives $E_1, \ldots, E_n$ are themselves expressions, while in a deduction of the form

$$\mathbf{dmatch}\ E\ (\pi_1?\ D_1) \cdots (\pi_n?\ D_n)$$

the alternatives are *deductions* $D_1, \ldots, D_n$ This maintains the original design invariant of the language, whereby expressions perform computation and can return anything whereas deductions perform inference and must return propositions that are entailed by the assumption base. This principle would clearly be violated if we admitted a deduction form such as **dlet** $I = F$ **in** E. We explain each new construct below.

[3]Observe that in practice we usually have to incorporate the conclusions of all deductive arguments anyway, because if the conclusion of some D in a method call $\mathbf{!}E\ \cdots D \cdots$ is not necessary for the call to go through, then D is extraneous. Such deductions are unlikely to occur in practice.

Sequencing

The **let** $I = F$ **in** E construct is used for sequencing and naming: it prescribes that the phrase F should be evaluated before the body E, and furthermore, that E should be able to refer to the result of F by the name I. Note that F, being a phrase, could be either an expression or a deduction. In the latter case, of course, every free occurrence of I within E will denote a proposition.

In a similar fashion, the construct **dlet** $I = F$ **in** D precedes a deduction D by a phrase F, which can either be a computation E_0 or some other deduction D_0. As with **let**, the identifier I can be used within the body D to refer to the result of F. But, in addition, if F is a deduction D_0, then the conclusion of D_0 becomes available as a lemma within the body D. More precisely, the body D will be evaluated in the assumption base of the entire **dlet** augmented with the conclusion of D_0. These semantics are formally captured by the three rules below. Note that two distinct rules are needed for **dlet** $I = F$ **in** D, one covering the case when F is an expression and one for when F is a deduction, as explained above.

$$\frac{\beta \vdash F \leadsto V_0 \qquad \beta \vdash E[V_0/I] \leadsto V}{\beta \vdash \textbf{let } I = F \textbf{ in } E \leadsto V} \quad [R_8]$$

$$\frac{\beta \vdash E_0 \leadsto V_0 \qquad \beta \vdash D[V_0/I] \leadsto P}{\beta \vdash \textbf{dlet } I = E_0 \textbf{ in } D \leadsto P} \quad [R_9]$$

$$\frac{\beta \vdash D_0 \leadsto P \qquad \beta \cup \{P\} \vdash D[P/I] \leadsto Q}{\beta \vdash \textbf{dlet } I = D_0 \textbf{ in } D \leadsto Q} \quad [R_{10}]$$

The difference between **let** and **dlet** is essentially the same difference between function calls and method calls that we mentioned earlier: when we evaluate the arguments of a method call, we keep track of whether an argument is a deduction; if so, its conclusion will be incorporated in the assumption base once all the arguments have been evaluated and we are ready to apply the given method. In fact, the phrases **let** $I = F$ **in** E and **dlet** $I = F$ **in** D could be taken as syntax sugar for the function call $(\lambda\, I\, .\, E)\ F$ and method call $\mathbf{!}(\phi\, I\, .\, D)\ F$, respectively. It is readily verified that, owing to the cut rule, this desugaring results in the exact same semantics as those given by the rules $[R_8]$—$[R_{10}]$ above.

We will allow for cascading **let** and **dlet** phrases by introducing **let** $I_1 = F_1 \cdots I_n = F_n$ **in** E as an abbreviation for **let** $I_1 = F_1$ **in** $(\cdots($**let** $I_n = F_n$ **in** $E)\cdots)$, and likewise for **dlet** $I_1 = F_1 \cdots I_n = F_n$ **in** D.

Finally, two related constructs are semicolon-separated sequences of deductions and expressions, enclosed with **begin-end** pairs. Their respective semantics are given by the following rules:

$$\frac{\beta \vdash D_1 \leadsto P_1 \qquad \beta \cup \{P_1\} \vdash D_2 \leadsto P_2 \qquad \cdots \qquad \beta \cup \{P_1, P_2, \ldots, P_{n-1}\} \vdash D_n \leadsto P_n}{\beta \vdash \textbf{begin } D_1; \cdots ; D_n \textbf{ end} \leadsto P_n} \quad [R_{11}]$$

$$\frac{\beta \vdash E_1 \leadsto V_1 \qquad \cdots \qquad \beta \vdash E_n \leadsto V_n}{\beta \vdash \textbf{begin } E_1; \cdots ; E_n \textbf{ end} \leadsto V_n} \quad [R_{12}]$$

Deductions of the form **begin** $D_1; \cdots ; D_n$ **end** are useful whenever we wish to compose a number of deductions without bothering to name their respective conclusions. Nameless sequences of expressions are not particularly useful in the absence of side effects, but were included here for symmetry.

$$\frac{}{\emptyset \vdash c \preceq c}\ [M_1] \qquad \frac{}{\emptyset \vdash P \preceq \star}\ [M_2]$$

when c is an atom A or $c \in \{\mathbf{true}, \mathbf{false}\}$

$$\frac{}{\langle I, P\rangle \vdash I \preceq P}\ [M_3] \qquad \frac{\sigma \vdash P \preceq \pi}{\sigma \vdash \neg P \preceq \neg \pi}\ [M_4] \qquad \frac{\sigma_1 \vdash P_1 \preceq \pi_1 \qquad \sigma_2 \vdash P_2 \preceq \pi_2}{\sigma_1, \sigma_2 \vdash P_1 \odot P_2 \preceq \pi_1 \odot \pi_2}\ [M_5]$$

for $\odot \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$

Figure 1.4: Inference rules for pattern matching.

Pattern matching

We define a language of patterns by means of the following abstract grammar:

$$\pi \ ::= \ I \mid A \mid \mathbf{true} \mid \mathbf{false} \mid \star \mid \neg\pi \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \pi_1 \Rightarrow \pi_2 \mid \pi_1 \Leftrightarrow \pi_2.$$

For any pattern π that contains all and only the identifiers in the set $\{I_1, \ldots, I_n\}$, we write $PI(\pi)$ for an arbitrary listing $[I_1, \ldots, I_n]$ of these identifiers (say, in the order in which they appear in the pattern, from left to right). Next we introduce a deduction system for pattern matching, with rules that establish judgments of the form $\sigma \vdash P \preceq \pi$, to be read "With respect to the set of bindings σ, proposition P matches the pattern π". By a "set of bindings" σ we will mean a finite set of ordered pairs $\langle I, P\rangle$, consisting of an identifier I and a proposition P, that is a function, meaning that for all $\langle I_1, P_1\rangle$ and $\langle I_2, P_2\rangle$ in σ, if $I_1 = I_2$ then $P_1 = P_2$. Given two such sets σ_1 and σ_2, we write σ_1, σ_2 for their set-theoretic union, provided that the said union respects the aforementioned functional provision; if not, σ_1, σ_2 is undefined.

The rules for pattern matching are shown in Figure 1.4. As an example of using these rules, the proof below shows that the conjunction $A \wedge (B \vee C)$ matches the pattern $P \wedge Q$ under the bindings $\langle P, A\rangle$ and $\langle Q, B \vee C\rangle$. More formally, the proof establishes the judgment

$$\{\langle P, A\rangle, \langle Q, B \vee C\rangle\} \vdash A \wedge (B \vee C) \preceq P \wedge Q :$$

1. $\{\langle P, A\rangle\} \vdash A \preceq P$ $\quad [M_3]$
2. $\{\langle Q, B \vee C\rangle\} \vdash B \vee C \preceq Q$ $\quad [M_3]$
3. $\{\langle P, A\rangle, \langle Q, B \vee C\rangle\} \vdash A \wedge (B \vee C) \preceq P \wedge Q$ $\quad$ 1, 2, $[M_5]$

It is straightforward to give an algorithm that takes any proposition P and pattern π and produces a set of bindings σ such that $\sigma \vdash P \preceq \pi$, if such a σ exists at all; and fails if no such σ exists. Moreover, this algorithm is efficient, taking time linear in the size of the pattern.
We can now specify the semantics of **match** and **dmatch** with the following rules:

$$\frac{\beta \vdash E \leadsto P \qquad \beta \vdash E_j[P_1/I_1, \ldots, P_k/I_k] \leadsto V}{\beta \vdash \mathbf{match}\ \ E\ \ (\pi_1?\ E_1) \cdots (\pi_n?\ E_n) \leadsto V}\ [R_{13}]$$

provided that $\{\langle I_1, P_1\rangle, \ldots, \langle I_k, P_k\rangle\} \vdash P \preceq \pi_j$
and, for all $i < j$, there is no σ such that $\sigma \vdash P \preceq \pi_i$.

And similarly,

$$\frac{\beta \vdash E \leadsto P \qquad \beta \vdash D_j[P_1/I_1, \ldots, P_k/I_k] \leadsto Q}{\beta \vdash \textbf{dmatch}\ E\ (\pi_1?\ D_1) \cdots (\pi_n?\ D_n) \leadsto Q} \quad [R_{14}]$$

$$\text{provided that } \{\langle I_1, P_1\rangle, \ldots, \langle I_k, P_k\rangle\} \vdash P \preceq \pi_j$$
$$\text{and, for all } i < j, \text{ there is no } \sigma \text{ such that } \sigma \vdash P \preceq \pi_i.$$

The algorithmic interpretation of these rules is as follows: to evaluate an expression of the form **match** E $(\pi_1?\ E_1) \cdots (\pi_n?\ E_n)$ in an assumption base β, start by evaluating the *discriminant* E in β. If and when that produces a proposition P, try to match P against the first pattern π_1. If that succeeds under some set of bindings $\{\langle I_1, P_1\rangle, \ldots, \langle I_k, P_k\rangle\}$, evaluate $E_1[P_1/I_1, \ldots, P_k/I_k]$ in β; otherwise try the next pattern, π_2. If P matches π_2 under some $\sigma = \{(I_1, P_1), \ldots, (I_k, P_k)\}$, evaluate $E_2[P_1/I_1, \ldots, P_k/I_k]$ in β; otherwise continue with the next pattern. This process continues until we either find a pattern π_j that matches P, in which case we evaluate E_j in β after performing on it the substitution determined by the matching set of bindings; or until we exhaust all the given patterns without discovering a match, in which case we report a "No match found" error. It is also an error if the discriminant produces a value other than a proposition. The evaluation of a **dmatch** is entirely analogous, only now the end alternatives are deductions $D_1, \ldots, D_n$ rather than expressions $E_1, \ldots, E_n$.

Proofs by contradiction

Deductions of the form **suppose-absurd** E **in** D perform reasoning by contradiction. Their semantics are given by the following rule:

$$\frac{\beta \vdash E \leadsto P \qquad \beta \cup \{P\} \vdash D \leadsto \textbf{false}}{\beta \vdash \textbf{suppose-absurd}\ E\ \textbf{in}\ D \leadsto \neg P} \quad [R_{15}]$$

The idea here is that we want to establish $\neg P$, where P is the proposition denoted by the expression E, and we are going about it by way of contradiction: we are saying, in effect, "*Suppose* that P holds; then here is a deduction D that derives the contradiction **false** from that supposition." So, intuitively, the operational reading of $[R_{15}]$ is this:

> To evaluate **suppose-absurd** E **in** D in an assumption base β, first evaluate E in β to obtain some proposition P—the hypothesis to be refuted. Then add P to β and evaluate the body D. If the evaluation of D in $\beta \cup \{P\}$ produces the proposition **false**, then return $\neg P$. Report an error if E fails to produce a proposition P or if D fails to produce the constant **false**.

Thus, in natural deduction terminology, this construct can be viewed as a mechanism for "negation introduction".

Conclusion-annotated form

Deductions of the form

$$E\ \textbf{by}\ D \qquad (1.10)$$

are said to be written in "conclusion-annotated" style. Their formal semantics are given by the following rule:

$$\frac{\beta \vdash E \leadsto P \qquad \beta \vdash D \leadsto P}{\beta \vdash E\ \textbf{by}\ D \leadsto P} \quad [R_{16}]$$

Thus the conclusion of 1.10 is simply the conclusion of D. However, 1.10 says something over and above D: it explicitly specifies that the conclusion of D will be the proposition described by E. Accordingly, E can be viewed as an annotation to D.[4] The annotation serves as a promise that the conclusion of D will be as prescribed by E. A typical use is a

$$B \vee C \;\; \textbf{by} \;\; \textbf{!modus-ponens} \;\; A \Rightarrow B \vee C \;\; (\textbf{!dn} \;\; \neg\neg A)$$

It is an error if the conclusion produced by D is *not* identical to the value of E, since this amounts to breaking the promise made by the annotation. Therefore, operationally, to evaluate a deduction of the form 1.10 in an assumption base β, we first evaluate D in β to obtain a conclusion P and then we evaluate E in β to obtain a proposition Q (it is an error if E does not yield a proposition). If $P = Q$, we return P; otherwise we generate an error.

Recursion

The **fix** construct allows for recursive definitions. Intuitively, the expression $\mathbf{fix}\, I \,.\, E$ should be understood to denote the least fixed point of the function $\lambda\, I \,.\, E$. Thus $\mathbf{fix}\, I \,.\, E$ is essentially equivalent to $Y(\lambda\, I \,.\, E)$, where Y is a fixed-point lambda-calculus combinator. As an example, consider a function that takes an arbitrarily long conjunction of atoms $A_1 \wedge A_2 \wedge \cdots \wedge A_n$ and turns in into the disjunction $A_1 \vee A_2 \vee \cdots \vee A_n$. This can be expressed as

$$\begin{array}{ll} \mathbf{fix}\, \mathit{convert} \,.\, \lambda\, P \,.\, \mathbf{match}\;\; P & \\ \qquad\qquad (Q \wedge R\,?\;\; Q \vee (\mathit{convert}\;\; R)) & \\ \qquad\qquad (Q?\;\; Q) & \end{array} \tag{1.11}$$

The formal semantics of **fix** are given by the following rule:

$$\frac{\beta \vdash E[\mathbf{fix}\, I \,.\, E/I] \rightsquigarrow V}{\beta \vdash \mathbf{fix}\, I \,.\, E \rightsquigarrow V} \quad [R_{17}]$$

Reading the rule backwards, we see that one application of $[R_{17}]$ to an expression of the form $\mathbf{fix}\, I \,.\, E$ uncovers the body E, which will typically be a function or a method, and hence a value, but with every free occurrence of I in it replaced by $\mathbf{fix}\, I \,.\, E$, which sets up the recursion for the next unrolling. Also note that the assumption base is simply carried over unchanged during the application of this rule.

The definitions of free and bound identifier occurrences has to be modified in order to account for the new syntax forms. For instance, an expression of the form **let** $I = F$ **in** E or a deduction of the form **dlet** $I = F$ **in** D binds I within E and D respectively; a pattern-expression pair $(Q?\;\; E)$ or pattern-deduction pair $(Q?\;\; D)$ binds all the identifiers that occur in Q within E and D, respectively; an expression of the form $\mathbf{fix}\, I \,.\, E$ binds I within E; and so on. Of course the definition of the substitution operation $F[F_1/I_1, \ldots, F_k/I_k]$ needs to be modified accordingly as well; the details are straightforward and we omit them. Other simple syntactic notions such as that of phantom deductions also need to be properly extended.

[4]If we view E as the "type" of D, then this is similar to explicitly inserting type declarations in a language that does not normally require them, such as ML, for documentation and readability purposes. Note, however, that in most such languages types are statically fixed constants that do not get dynamically evaluated, whereas in our case E will be evaluated at run-time and might even lead to an infinite loop. Thus, if we wish to push the type analogy, it is more appropriate to view E as a dependent type in an undecidable type system—but then it should be noted that type annotations in such systems are usually mandatory rather than optional, in order to ensure that type checking is decidable.

As an example of using $[R_{17}]$ and some of the other new rules, let E_1 be the expression

$$\begin{array}{l} \textbf{fix}\, g\,.\,\lambda\, P.\,\textbf{match}\ P \\ \qquad (Q \wedge R\,?\ Q \vee (g\ \ R)) \\ \qquad (Q?\ \ Q) \end{array}$$

and let E_2 be the body of the above **fix**, namely, the expression

$$\lambda\, P.\,\textbf{match}\ P\ \ (Q \wedge R?\ Q \vee (g\ \ R))\ \ (Q?\ \ Q).$$

To save space, we will use $\lceil \cdots \rceil$ as an "anti-quotation" operator, writing, for instance,

$$\textbf{fix}\, g\,.\,\lceil E_2 \rceil$$

for the expression E_1 given above. The derivation below establishes the judgment

$$\beta \vdash \textbf{let}\ \ \mathit{convert} = \lceil E_1 \rceil\ \ \textbf{in}\ \ \mathit{convert}\ \ A \wedge (B \wedge C) \rightsquigarrow A \vee (B \vee C)$$

for some arbitrary assumption base β:

1. $\beta \vdash \lambda\, P\,.\,\textbf{match}\ P\ \ (Q \wedge R\,?\ Q \vee (\lceil E_1 \rceil\ \ R))\ \ (Q?\ \ Q) \rightsquigarrow$
   $\lambda\, P\,.\,\textbf{match}\ P\ \ (Q \wedge R\,?\ Q \vee (\lceil E_1 \rceil\ \ R))\ \ (Q?\ \ Q)$ $[R_4]$
2. $\beta \vdash \lceil E_1 \rceil \rightsquigarrow \lambda\, P\,.\,\textbf{match}\ P\ \ (Q \wedge R\,?\ Q \vee (\lceil E_1 \rceil\ \ R))\ \ (Q?\ \ Q)$ 1, $[R_{17}]$
3. $\beta \vdash C \rightsquigarrow C$ $[R_4]$
4. $\beta \vdash \textbf{match}\ C\ \ (Q \wedge R\,?\ Q \vee (\lceil E_1 \rceil\ \ R))\ \ (Q?\ \ Q) \rightsquigarrow C$ 3, 2, $[R_{13}]$
5. $\beta \vdash \lceil E_1 \rceil\ \ C \rightsquigarrow C$ 2, 3, 4, $[R_6]$
6. $\beta \vdash B \rightsquigarrow B$ $[R_4]$
7. $\beta \vdash \vee \rightsquigarrow \vee$ $[R_4]$
8. $\beta \vdash B \vee C \rightsquigarrow B \vee C$ $[R_4]$
9. $\beta \vdash B \vee (\lceil E_1 \rceil\ \ C) \rightsquigarrow B \vee C$ 7, 6, 5, 8 $[R_5]$
10. $\beta \vdash B \wedge C \rightsquigarrow B \wedge C$ $[R_4]$
11. $\beta \vdash \textbf{match}\ B \wedge C$
    $(Q \wedge R\,?\ Q \vee (\lceil E_1 \rceil\ \ R))$
    $(Q?\ \ Q) \rightsquigarrow B \vee C$ 10, 9, $[R_{13}]$
12. $\beta \vdash (\lceil E_1 \rceil\ \ B \wedge C) \rightsquigarrow B \vee C$ 2, 10, 11, $[R_6]$
13. $\beta \vdash A \rightsquigarrow A$ $[R_4]$
14. $\beta \vdash A \vee (B \vee C) \rightsquigarrow A \vee (B \vee C)$ $[R_4]$
15. $\beta \vdash A \vee (\lceil E_1 \rceil\ \ B \wedge C) \rightsquigarrow A \vee (B \vee C)$ 7, 13, 12, 14, $[R_5]$
16. $\beta \vdash A \wedge (B \wedge C) \rightsquigarrow A \wedge (B \wedge C)$ $[R_4]$
17. $\beta \vdash \textbf{match}\ A \wedge (B \wedge C)$
    $(Q \wedge R\,?\ Q \vee (\lceil E_1 \rceil\ \ R))$
    $(Q?\ \ Q) \rightsquigarrow A \vee (B \vee C)$ 16, 15, $[R_{13}]$
18. $\beta \vdash (\lambda\, P\,.\,\textbf{match}\ P$
    $(Q \wedge R\,?\ Q \vee (\lceil E_1 \rceil\ \ R))$
    $(Q?\ \ Q))\ \ A \wedge (B \wedge C) \rightsquigarrow A \vee (B \vee C)$ 1, 16, 17, $[R_6]$
19. $\beta \vdash \textbf{let}\ \ \mathit{convert} = \lceil E_1 \rceil\ \ \textbf{in}\ \ \mathit{convert}\ \ A \wedge (B \wedge C) \rightsquigarrow A \vee (B \vee C)$ 2, 18, $[R_8]$

What is more interesting for our purposes is that, because methods are expressions, **fix** can also be used to formulate recursive methods. A recursive method will typically be given in the form

$$\textbf{fix}\, M\,.\,\phi\, I_1, \ldots, I_n\,.\,\cdots\ \textbf{!}M\ \ \cdots \tag{1.12}$$

Because methods are self-evaluating (rule $[R_4]$), one application of $[R_{17}]$ to 1.12 will result in the method

$$\phi\, I_1, \ldots, I_n\,.\,\cdots\ \textbf{!}M\ \ \cdots[(\textbf{fix}\, M\,.\,\phi\, I_1, \ldots, I_n\,.\,\cdots\ \textbf{!}M\ \ \cdots)/M]$$

thereby "uncovering" the method body of the **fix** and also setting up the next unrolling by replacing every free occurrence of M in the body of the method by the entire **fix** expression. We will give several examples of recursive methods in the sequel.

It is straightforward to show that the extended language $\overline{\mathcal{NDL}_0^\omega}$ remains sound. We now have a derivability relation $\Vdash_{\overline{\mathcal{NDL}_0^\omega}}$ that properly extends the relation $\Vdash_{\mathcal{NDL}_0^\omega}$ defined in Section 1.4. The proof that $\Vdash_{\overline{\mathcal{NDL}_0^\omega}}$ is Tarskian is identical to the proof of Lemma 1.3. We can also show that $\Vdash_{\overline{\mathcal{NDL}_0^\omega}}$ is the least Tarskian relation that respects the primitive methods of $\overline{\mathcal{NDL}_0^\omega}$ and is closed under hypothetical reasoning[5] with essentially the same argument that was given in the proof of Theorem 1.4; we only need to consider some additional cases in the inductive step, namely, rules $[R_9]$, $[R_{10}]$, $[R_{11}]$, $[R_{14}]$, $[R_{15}]$, and $[R_{16}]$. All of these are readily handled by the appropriate inductive hypotheses, along with the transitivity of $\Vdash_{\overline{\mathcal{NDL}_0^\omega}}$.

1.7 Examples

Schematic abstraction

Consider the following inference rule:

$$\frac{\vdash P_1 \Rightarrow (P_2 \Rightarrow P_3)}{\vdash (P_1 \wedge P_2) \Rightarrow P_3} \quad [\textit{un-curry}]$$

The name "uncurry" derives from identifying the connectives $\wedge$ and $\Rightarrow$ with the type constructors $\times$ and $\rightarrow$, respectively. In that light the uncurrying rule is viewed as transforming a curried functional type $T_1 \rightarrow T_2 \rightarrow T_3$ to the "uncurried" $(T_1 \times T_2) \rightarrow T_3$. (We will also shortly consider a dual rule for "currying" that proceeds in the reverse direction.) We would like a $\mathcal{NDL}_0^\omega$ method *un-curry* that takes a premise of the form $P_1 \Rightarrow (P_2 \Rightarrow P_3)$ (we say that a proposition P is a "premise" of a method M if P is expected to be in the assumption base when M is invoked) and produces the conclusion $(P_1 \wedge P_2) \Rightarrow P_3$. For instance, we should have

$$\beta \cup \{A \Rightarrow (\neg B \Rightarrow C)\} \vdash \mathbf{!}\,\textit{un-curry}\ A \Rightarrow (\neg B \Rightarrow C) \rightsquigarrow (A \wedge \neg B) \Rightarrow C \qquad (1.13)$$

for any assumption base β. The language itself does not provide any primitive method for that purpose. What we can do instead is define our own method that implements *un-curry* in terms of the existing primitive methods and syntactic forms. The following definition achieves this:

```
un-curry = φ premise . dmatch premise
                     P₁ ⇒ (P₂ ⇒ P₃)?  assume P₁ ∧ P₂ in
                                        begin
                                          P₁ by !left-and P₁ ∧ P₂;
                                          P₂ ⇒ P₃ by !modus-ponens premise P₁;
                                          P₂ by !right-and P₁ ∧ P₂;
                                          P₃ by !modus-ponens P₂ ⇒ P₃ P₂
                                        end
```

[5] Of course since the primitive methods of the extended language are the same as those of the core, this means that

$$\Vdash_{\overline{\mathcal{NDL}_0^\omega}} = \Vdash_{\mathcal{NDL}_0^\omega}$$

which shows that the extension is conservative.

We can now apply *un-curry* just as if it were a primitive method. Indeed, the clients of *un-curry* *should* think of the method as primitive, since the details of its implementation are irrelevant to their purposes. All that matters to a client is how the method behaves as a black box. That behavior is spelled out in the method's specification, which serves as a contract between client and implementor. In this case, the contract only demands that whenever *un-curry* is applied to an argument, that argument should be a proposition of the form $P_1 \Rightarrow (P_2 \Rightarrow P_3)$ and should be in the assumption base. If a client honors these terms, then *un-curry* must behave as advertised: it must derive the conclusion $(P_1 \wedge P_2) \Rightarrow P_3$. Exactly *how* that conclusion is derived is of little concern to the client, as long as the derivation does not require an inordinate amount of time or resources.

In terms of style, those who find the foregoing definition of *un-curry* verbose could opt for a more succinct alternative by removing the conclusion annotations, or even the **dlet** too, resulting for example in:

```
un-curry = φ premise . dmatch premise
                        $P_1 \Rightarrow (P_2 \Rightarrow P_3)$?  assume $P_1 \wedge P_2$ in
                                           !mp (!mp premise (!left-and $P_1 \wedge P_2$))
                                               (!right-and $P_1 \wedge P_2$)
```

where the body of the **assume** is one single nested method call. It is instructive to observe how the assumption base is implicitly manipulated during the evaluation of this method call. Proceeding from inside to outside, the **left-and** method call will produce the conclusion P_1, which will be incorporated in the assumption base when we come to apply the innermost **mp**. That application will yield the conclusion $P_2 \Rightarrow P_3$, and that will in turn be incorporated in the assumption base when we come to apply the outer modus ponens, along with P_2, the conclusion of the **right-and** application—and these two additions will make sure that the outer modus ponens correctly produces the desired conclusion, P_3. In terms of the formal semantics, the evaluation of this nested method call will employ the cut rule three times, once for each nested method application, and will result in three assumption base augmentations. All this work will be done automatically by the $\mathcal{NDL}_0^\omega$ interpreter. The user is thus free to concentrate on the essential steps of the proof rather than the tedious aspects of managing premises and intermediate conclusions. It is largely this streamlining of the assumption base that makes DPL proofs and proof methods readable and writable.[6]

Let us now define a method *curry* that takes a premise of the form $(P_1 \wedge P_2) \Rightarrow P_3$ and derives $P_1 \Rightarrow (P_2 \Rightarrow P_3)$:

```
curry = φ premise . dmatch premise
                    $(P_1 \wedge P_2) \Rightarrow P_3$?  assume $P_1$ in
                                        assume $P_2$ in
```

[6]One might suggest that the utility of a new abstraction can be measured by the amount of extra work done by an interpreter for the new language (the one that has the new abstraction) compared to interpreters for similar languages that lack the abstraction. Consider the dynamic binding afforded by inheritance in some object-oriented language L_O, and let L_{NO} be some conventional, non-object-oriented language. When an interpreter for L_O evaluates a function call, there is something going on behind the curtains that goes over and above what happens in a function call in L_{NO}: dynamic binding. In a non-object-oriented language, function applications are easier to evaluate: we evaluate the function, evaluate the arguments, and then we perform β-reduction: we replace the formal parameters by the actual arguments. But in L_O there is something going on *in addition* to that: method dispatch. The interpreter has to work harder, because it needs to do everything that needs to be done in a conventional language *as well as* determine the most appropriate function to call. Intuitively, it is this extra work that is the payoff of the new abstraction. In non-object-oriented languages this work would have to be done explicitly by the user, via long and convoluted `case` or `switch` statements. The semantics of inheritance and dynamic binding hide all that by shifting the burden of method dispatch from the user to the language. Likewise, DPLs alleviate the tedium of hypothesis management by pushing as much of the burden as possible from the user to the language.

$$\mathbf{!mp}\ \textit{premise}\ (\mathbf{!both}\ P_1\ P_2)$$

The reader will verify that

$$\beta \cup \{(P_1 \wedge P_2) \Rightarrow P_3\} \vdash \mathbf{!}\,\textit{curry}\ \ (P_1 \wedge P_2) \Rightarrow P_3 \rightsquigarrow P_1 \Rightarrow (P_2 \Rightarrow P_3)$$

in any β, and that *curry* and *un-curry* are inverses in the sense that

$$\beta \cup \{P\} \vdash \mathbf{!}\,\textit{curry}\ \ (\mathbf{!}\,\textit{un-curry}\ \ P) \rightsquigarrow P$$

and

$$\beta \cup \{P\} \vdash \mathbf{!}\,\textit{un-curry}\ \ (\mathbf{!}\,\textit{curry}\ \ P) \rightsquigarrow P$$

for all β and every P of the right form.

The two foregoing methods can be viewed as "derived inference rules". Every logic comes with a fixed collection of primitive inference rules, where an inference rule R is usually depicted graphically as

$$\frac{\vdash P_1 \quad \cdots \quad \vdash P_n}{\vdash P}\ [R]$$

and can be understood as an algorithm that is given theorems of the form $P_1, \ldots, P_n$ and produces P. Since it is usually impractical to write down proofs using nothing but primitive rules, it is useful to be able to introduce additional inference rules as needed. To ensure that such extensions are conservative, we must be able to guarantee that a new rule does not allow us to prove anything that we could not already prove with the primitive inference rules alone. This is usually achieved by presenting an algorithm that can "eliminate" or "expand" any particular use of the derived rule. That is, suppose we have a proof which, at a certain step, applies a derived rule DR to certain premises $P_1, \ldots, P_n$ and produces some conclusion P. Then our algorithm should be able to eliminate that step and replace it with a "primitive proof", i.e., a proof that does *not* use DR but has the same effect, namely, the deduction of P from $P_1, \ldots, P_n$. Hence the term "derived inference rule": the new rule DR is derived from—or expressed in terms of—the primitive rules. As Manna [21] puts is:

> In order to be able to write shorter deductions for wwfs in practice, it is most convenient to have a library of *derived inference rules.* Each such rule can be given an effective proof in the sense that we can show effectively how to replace any derived rule of inference whenever it is used in a deduction by an appropriate sequence of wffs using only the "primitive" rules of inference and axioms.

In the case of $\mathcal{NDL}_0^\omega$ methods such as *curry*, the "elimination algorithm" is given by the very definition of the method. The body of the method, interpreted operationally in accordance with the evaluation semantics of the language, specifies precisely how to "expand out" any application of the method into a proof that uses only primitive methods (or previously defined methods, which will in turn be expanded to primitives). The expansion will occur dynamically at evaluation time and will be performed by the interpreter. In fact it is straightforward to instrument the interpeter so that it produces not only the conclusion (denotation) of a given deduction D, but also, as a side effect, a primitive proof D', that is, one that uses nothing but the primitive inference rules of $\mathcal{NDL}_0^\omega$ and its primitive deductive forms (**assume**, etc.). The two deductions D and D' will be observationally equivalent, but one of them, D', will be much simpler than the other because most of the computation embedded in D will have been discarded. In fact, D' will be a type-α proof—it will contain virtually no expressions E in it and will be guaranteed to terminate in linear time. As we will argue later, both D and D' should be viewed as bona fide proofs; the only difference is that D' is a *simpler* proof because it has much less computational content. To emphasize this distinction, we refer to D' as a *certificate*. The process of constructing a certificate as a side effect of evaluating a type-ω proof is known as *proof expansion*, or *certificate generation*. We will discuss this subject further in Section 1.9.

Let us formulate some more interesting methods. Consider De Morgan's laws:

$$\frac{\neg(P \vee Q)}{\neg P \wedge \neg Q}\ [dm\text{-}1] \qquad \frac{\neg P \wedge \neg Q}{\neg(P \vee Q)}\ [dm\text{-}2] \qquad \frac{\neg(P \wedge Q)}{\neg P \vee \neg Q}\ [dm\text{-}3] \qquad \frac{\neg P \vee \neg Q}{\neg(P \wedge Q)}\ [dm\text{-}4]$$

We would like to have four such methods, each of which takes a premise of the appropriate form and produces the corresponding conclusion. For instance, we would like to have a method *dm-1* such that

$$\beta \cup \{\neg(A \vee (B \Rightarrow C))\} \vdash \mathbf{!}\, dm\text{-}1\ \ \neg(A \vee (B \Rightarrow C)) \rightsquigarrow \neg A \wedge \neg(B \Rightarrow C)$$

in any β. Since the language does not provide any primitive methods for De Morgan's laws, we have to define such methods ourselves. We begin with *dm*-1:

```
$dm$-1 = $\phi$ premise . dmatch premise
                    $\neg(P \vee Q)$?  dlet not-$P$ = suppose-absurd $P$ in
                                              !absurd (!left-either $P$ $Q$) $\neg(P \vee Q)$
                                 not-$Q$ = suppose-absurd $Q$ in
                                              !absurd (!right-either $P$ $Q$) $\neg(P \vee Q)$
                              in
                                !both not-$P$ not-$Q$
```

The reader will verify that $\beta \cup \{\neg(P \vee Q)\} \vdash \mathbf{!}\, dm\text{-}1\ \ \neg(P \vee Q) \rightsquigarrow \neg P \wedge \neg Q$ in any assumption base β. We continue with the remaining three laws:

```
$dm$-2 = $\phi$ premise . dmatch premise
                    $\neg P \wedge \neg Q$?  dlet imp-1 = ($P \Rightarrow$ false) by assume $P$ in
                                                            !absurd $P$ (!left-and premise)
                                 imp-2 = ($Q \Rightarrow$ false) by assume $Q$ in
                                                            !absurd $P$ (!left-and premise)
                              in
                                $\neg(P \vee Q)$ by suppose-absurd $P \vee Q$ in
                                                   !cd $P \vee Q$ imp-1 imp-2

$dm$-3 = $\phi$ premise . dmatch premise
                    $\neg(P \wedge Q)$?  dlet $L$ = $\neg\neg(\neg P \vee \neg Q)$ by
                                       suppose-absurd $\neg(\neg P \vee \neg Q)$ in
                                         dlet $L_1$ = $\neg\neg P \wedge \neg\neg Q$ by !dm-1 $\neg(\neg P \vee \neg Q)$
                                              $L_2 = P$ by !dn (!left-and $L_1$)
                                              $L_3 = Q$ by !dn (!right-and $L_1$)
                                              $L_4 = P \wedge Q$ by !both $L_2$ $L_3$
                                         in
                                           !absurd $L_4$ premise
                              in
                                $\neg P \vee \neg Q$ by !dn $L$
```

Here we have already begun to witness the benefits of abstraction: lemma L_1 above is derived by applying the previously defined method *dm*-1 to the hypothesis $\neg(\neg P \vee \neg Q)$. We conclude with *dm*-4:

```
dm-4 = ϕ premise . dmatch premise
                     ¬P ∨ ¬Q?  ¬(P ∧ Q) by
                                 suppose-absurd P ∧ Q in
                                   dlet imp-1 = (¬P ⇒ false) by
                                                  assume ¬P in
                                                    !absurd (!left-and P ∧ Q) ¬P
                                        imp-2 = (¬Q ⇒ false) by
                                                  assume ¬Q in
                                                    !absurd (!right-and P ∧ Q) ¬Q
                                   in
                                     !cd premise imp-1 imp-2
```

We can now write a generic De Morgan method *dm* that dispatches the appropriate method in accordance with the form of the input proposition:

```
dm = ϕ premise . dmatch premise
                  ¬(P ∨ Q)?  !dm-1 premise
                  ¬P ∧ ¬Q?   !dm-2 premise
                  ¬(P ∧ Q)?  !dm-3 premise
                  ¬P ∨ ¬Q?   !dm-4 premise
```

The next method derives $P \vee \neg P$ for any given P:

```
excl-middle = ϕ P . !dn suppose-absurd ¬(P ∨ ¬P) in
                          dlet contradiction = ¬P ∧ ¬¬P by !dm ¬(P ∨ ¬P)
                          in
                            !absurd (!left-and contradiction) (!right-and contradiction)
```

It is now straightforward to define a method that does reasoning by cases. Graphically, we want a binary method *cases* with the following behavior:

$$\frac{\vdash P \Rightarrow Q \qquad \vdash \neg P \Rightarrow Q}{\vdash Q} \quad [cases]$$

This can be achieved by using the primitive method **cd** ("constructive dilemma") in tandem with *excl-middle*:

```
cases = ϕ P1, P2 . dmatch P1 ∧ P2
                     (P ⇒ Q) ∧ (¬P ⇒ Q)?  !cd (!excl-middle P) P1 P2
```

Recursive methods

All of the methods we have written so far are obtainable from concrete deductions through simple schematic abstraction. For instance, we observe that the deduction

```
assume ¬A in
  !claim ¬A
```

will work properly even if we substitute an arbitrary proposition P in place of $\neg A$, and thus we arrive at the method

```
foo = φ P . assume P in
              !claim P
```

The body of this method is the deduction "schema" **assume** P **in !claim** P, from which we can obtain infinitely many concrete deductions by replacing the parameter P by some particular proposition.

Most derived inference rules are usually obtained from concrete proofs through schematic abstraction. However, owing to the power of recursion and conditional branching, $\mathcal{NDL}_0^\omega$ methods go well beyond schematic abstraction. As a simple example, consider a method dn^* that takes a premise of the form $\neg \cdots \neg P$ and removes as many pairs of negation signs from the front as possible. For instance, we should have

$$\beta \cup \{\neg\neg\neg\neg A\} \vdash \mathbf{!}\, dn^* \;\; \neg\neg\neg\neg A \leadsto A$$

while $\beta \cup \{\neg\neg\neg B\} \vdash \mathbf{!}\, dn^* \;\; \neg\neg\neg B \leadsto \neg B$, and $\beta \cup \{A \vee C\} \vdash \mathbf{!}\, dn^* \;\; A \vee C \leadsto A \vee C$. If we were to specify dn^* in traditional graphical notation it would look as follows, where we write $(\neg\neg)^n$ for $n \geq 0$ consequtive occurences of $\neg\neg$:

$$\frac{\vdash (\neg\neg)^n P}{\vdash P} \quad [dn^*]$$

And to justify this as a derived inference rule, we would have to give an "expansion algorithm" capable of replacing any application of dn^* to a premise of the form $(\neg\neg)^n P$ by a proof D that does not use dn^*. This algorithm would most likely proceed by induction on n:

> When $n = 0$ the desired proof D consists simply of claiming P, which is sound because presumably $(\neg\neg)^0 P = P$ is a theorem. For the inductive step of $n + 1$, the premise can be written as $\neg\neg(\neg\neg)^n P$, for $n \geq 0$. In that case let D_1 be the one-line proof we get by applying the primitive rule of double negation to the premise $\neg\neg(\neg\neg)^n P$. Accordingly, D_1 deduces $(\neg\neg)^n P$ from the premise $\neg\neg(\neg\neg)^n P$. Next, apply the transformation algorithm recursively to $(\neg\neg)^n P$ to obtain a deduction D_2 of P from $(\neg\neg)^n P$. The concatenation of D_1 with D_2 then constitutes the desired deduction D of P from $\neg\neg(\neg\neg)^n P$.

In $\mathcal{NDL}_0^\omega$ we can readily express dn^* as follows:

```
dn* = fix M . φ premise . dmatch premise
                              ¬¬P?  begin
                                       P by !dn premise;
                                       !M P
                                    end
                              ⋆?  !claim premise
```

This definition is an almost verbatim transcription of the inductive "expansion algorithm" described above. It essentially tells us how to put together a primitive deduction of P from $(\neg\neg)^n P$ for *any* given $n \geq 0$. For instance, if we apply dn^* to $\neg\neg\neg\neg A$, in an assumption base that contains $\neg\neg\neg\neg A$, then the expanded proof (the "certificate", as we will call such expanded proofs in Section 1.9) will be:

```
!dn $\neg\neg\neg\neg A$;
!dn $\neg\neg A$;
!claim $A$
```

The interesting point here is that *dn** is not obtained through simple schematic abstraction from any particular proof. The time complexity of this method is asymptotically proportional to the size of the input premise as measured by the number of negation signs prepended to it; whereas the complexity of a method that is obtained by schematic abstraction does not vary with the size of its inputs. Thus we see that recursion and conditional branching—achieved here via pattern matching—allow us to express powerful derived inference rules with arbitrarily sophisticated "expansion algorithms". The semantics of the language, owing to the abstraction of assumption bases, guarantee that no matter how complicated the method is, the final conclusion will always be sound.

As a more useful example, consider an inference rule *equiv-cong* that takes a premise of the form $P_1 \Leftrightarrow P_2$ and derives the equivalence $P \Leftrightarrow P[P_2/P_1]$, for some arbitrary proposition P, where we write $P[P_2/P_1]$ to denote the proposition obtained from P by replacing every occurrence of P_1 by P_2. Therefore, graphically, the rule may be depicted as

$$\frac{\vdash P_1 \Leftrightarrow P_2}{\vdash P \Leftrightarrow P[P_2/P_1]} \quad [\textit{equiv-cong}]$$

We can formulate this as a binary method *equiv-cong* that takes the arbitrary proposition P as its first argument and the premise $P_1 \Leftrightarrow P_2$ as its second argument and attempts to deduce the conclusion $P \Leftrightarrow P[P_2/P_1]$. The "expansion algorithm" proceeds inductively as follows. The first thing we do is check whether P is equal to P_1. If so, the desired conclusion $P \Leftrightarrow P[P_2/P_1]$ is identical to the given premise $P_1 \Leftrightarrow P_2$, so we simply claim that premise. Otherwise we analyze the structure of P. If P is a negation $\neg Q_1$, we call the algorithm recursively on Q_1 and $P_1 \Leftrightarrow P_2$. This will presumably result in a theorem of the form $Q_1 \Leftrightarrow Q_1[P_2/P_1]$. But from this it is sound to conclude $\neg Q_1 \Leftrightarrow \neg Q_1[P_2/P_1]$, which is to say $P \Leftrightarrow P[P_2/P_1]$, since, in general, the following rule is sound:

$$\frac{\vdash P_1 \Leftrightarrow P_1'}{\vdash \neg P_1 \Leftrightarrow \neg P_1'} \quad [\textit{not-cong}]$$

The remaining propositional cases are handled similarly, using analogous "congruence" rules such as

$$\frac{\vdash P_1 \Leftrightarrow P_1' \qquad \vdash P_2 \Leftrightarrow P_2'}{\vdash P_1 \wedge P_2 \Leftrightarrow P_1' \wedge P_2'} \quad [\textit{and-cong}] \qquad \frac{\vdash P_1 \Leftrightarrow P_1' \qquad \vdash P_2 \Leftrightarrow P_2'}{\vdash P_1 \vee P_2 \Leftrightarrow P_1' \vee P_2'} \quad [\textit{or-cong}]$$

and likewise for *if-cong* and *iff-cong*. Finally, if P is neither identical to P_1 nor a compound proposition, then it must be an atom distinct from P_1, in which case $P[P_2/P_1] = P$ and the desired conclusion is the trivial equivalence $P \Leftrightarrow P$, which is easily deduced for an arbitrary P by the following method:

```
reflex-equiv = $\phi$ P . dlet imp = assume P in !claim P
                     in
                       !equiv imp imp
```

The definitions of *not-cong*, *and-cong*, etc., are also straightforward. We illustrate with *and-cong* and leave the remaining cases as straightforward exercises:

```
and-cong = $\phi$ $eq_1$, $eq_2$ .
  dmatch $eq_1 \wedge eq_2$
```

```
$(P_1 \Leftrightarrow P_1') \wedge (P_2 \Leftrightarrow P_2')$?  **dlet** $imp_1 =$ **assume** $P_1 \wedge P_2$ **in**
                                          **begin**
                                            $P_1'$ **by !mp** (**!left-iff** $eq_1$) (**!left-and** $P_1 \wedge P_2$);
                                            $P_2'$ **by !mp** (**!left-iff** $eq_2$) (**!right-and** $P_1 \wedge P_2$);
                                            $P_1' \wedge P_2'$ **by !both** $P_1' P_2'$
                                          **end**
                                 $imp_2 =$ **assume** $P_1' \wedge P_2'$ **in**
                                          **begin**
                                            $P_1$ **by !mp** (**!right-iff** $eq_1$) (**!left-and** $P_1' \wedge P_2'$);
                                            $P_2$ **by !mp** (**!right-iff** $eq_2$) (**!right-and** $P_1' \wedge P_2'$);
                                            $P_1 \wedge P_2$ **by !both** $P_1' P_2'$
                                          **end**
                             **in**
                               $(P_1 \wedge P_2) \Leftrightarrow P_1' \wedge P_2'$ **by !equiv** $imp_1$ $imp_2$
```

We can now define *equiv-cong* as shown below. Note that we avoid having to pass the equivalence $P_1 \Leftrightarrow P_2$ as a second argument to each recursive call, since it remains constant throughout, by formulating an inner recursive method of one argument that lexically references P_1 and P_2:

```
$equiv\text{-}cong = \phi\, P, eq\,.$
  **dmatch** $eq$
    $P_1 \Leftrightarrow P_2$?  **dlet** $recurse =$ **fix** $M\,.\,\phi\, Q\,.$
                            **dmatch** $Q \equiv P_1$
                              **true**?  **!claim** $P_1 \Leftrightarrow P_2$
                              **false**?  **dmatch** $Q$
                                          $\neg Q_1$?  **!**$not\text{-}cong$ (**!**$M$ $Q_1$)
                                          $Q_1 \wedge Q_2$?  **!**$and\text{-}cong$ (**!**$M$ $Q_1$) (**!**$M$ $Q_2$)
                                          $Q_1 \vee Q_2$?  **!**$or\text{-}cong$ (**!**$M$ $Q_1$) (**!**$M$ $Q_2$)
                                          $Q_1 \Rightarrow Q_2$?  **!**$if\text{-}cong$ (**!**$M$ $Q_1$) (**!**$M$ $Q_2$)
                                          $Q_1 \Leftrightarrow Q_2$?  **!**$iff\text{-}cong$ (**!**$M$ $Q_1$) (**!**$M$ $Q_2$)
                                          $\star$?  **!**$reflex\text{-}equiv$ $Q$
                **in**
                  **!**$recurse$ $P$
```

We can now readily define a powerful binary method *replace* that takes an arbitrary premise P and a premise of the form $P_1 \Leftrightarrow P_2$ and derives the conclusion $P[P_2/P_1]$:

$replace = \phi\, P, eq\,.$**!mp** (**!left-iff** (**!**$equiv\text{-}cong$ P eq)) P

For instance, suppose that P is the proposition $A \wedge [C \Rightarrow (\neg\neg B \vee D)]$ and that we want to replace the occurence of $\neg\neg B$ by B, where both P and the equivalence $\neg\neg B \Leftrightarrow B$ are in the assumption base. Then applying *replace* to P and the said equivalence would derive the conclusion $A \wedge [C \Rightarrow (B \vee D)]$.

1.8 Evaluation and cost measures

1.8.1 An interpreter

In this section we present an interpreter $\mathcal{I}$ for $\overline{\mathcal{NDL}_0^\omega}$ that takes any closed phrase F and assumption base β and either produces a value V that represents the result of F in β; or else it diverges or generates an error. (We assume the availability of a nullary function *error* that aborts computation.) The definition of $\mathcal{I}$ appears in Figure 1.5, in ML pseudocode; a complete implementation in SML-NJ using environments rather than substitutions can be found in Section 1.16.

The interpreter uses two auxiliary functions *eval-fun-args* and *eval-meth-args* that evaluate the arguments $F_1, \ldots, F_n$ of a method and function call, respectively, in a given assumption base. The *eval-fun-args* function simply evaluates each argument F_i in the given assumption base, producing its value V_i, and when all arguments are evaluated it returns the results in a list $[V_1, \ldots, V_n]$. In addition, *eval-meth-args* keeps track of those arguments F_i that are deductions and stores their conclusions in a "lemma set" β' that is finally passed out along with the values of the arguments in a pair $\langle [V_1, \ldots, V_n], \beta' \rangle$. (As we discussed in Section 1.5, this implements the cut rule.) For generality, both functions take an arbitrary interpreter as an additional, third argument. We thus have:

```
$eval\text{-}fun\text{-}args(phrases, \beta, \mathcal{I}) = let\ f([], vals) = \overleftarrow{vals}$
                                              $f(F{::}L, vals) = f(L, \mathcal{I}(F, \beta){::}vals)$
                                           $in$
                                             $f(phrases, [])$
```

and

```
$eval\text{-}meth\text{-}args(phrases, \beta, \mathcal{I}) = let\ f([], vals, \beta') = \langle \overleftarrow{vals}, \beta' \rangle$
                                                $f(E{::}L, vals, \beta') = f(L, \mathcal{I}(E, \beta){::}vals, \beta')$
                                                $f(D{::}L, vals, \beta') = let\ P = \mathcal{I}(D, \beta)$
                                                                          $in$
                                                                            $f(L, P{::}vals, \beta' \cup \{P\})$
                                             $in$
                                               $f(phrases, [], \emptyset)$
```

For pattern matching, we need a function *match* that takes a proposition P, a pattern π, and a set of bindings σ, and either produces a set of bindings $\sigma' \supseteq \sigma$ with respect to which P matches π or else it returns *false*, indicating a match failure. In the seventh clause below, we use the symbol $\odot$ to range over the binary propositional connectives $\wedge, \vee, \Rightarrow, \Leftrightarrow$.

```
$match(A, A, \sigma) = \sigma$
$match(\mathbf{true}, \mathbf{true}, \sigma) = \sigma$
$match(\mathbf{false}, \mathbf{false}, \sigma) = \sigma$
$match(P, \star, \sigma) = \sigma$
$match(P, I, \sigma) = if\ \sigma\ contains\ a\ binding\ (I, Q)\ then$
                        $(if\ P = Q\ then\ \sigma\ else\ false)\ else\ \sigma \cup \{\langle I, P \rangle\}$
$match(\neg P, \neg \pi, \sigma) = match(P, \pi, \sigma)$
$match(P_1 \odot P_2, \pi_1 \odot \pi_2, \sigma) = let\ \sigma' = match(P_1, \pi_1, \sigma)$
                                         $in$
                                           $if\ \sigma' = false\ then\ false\ else\ match(P_2, \pi_2, \sigma')$
$match(\_, \_, \_) = false$
```

It is straightforward to prove that $match(P, \pi, \emptyset) = \sigma$ iff $\sigma \vdash P \preceq \pi$. The interpreter also uses a function *check* that takes a proposition P and a list of "cases" $[\langle \pi_1, F_1 \rangle, \ldots, \langle \pi_1, F_1 \rangle]$, finds the first pattern π_i that matches P with respect to some set of bindings σ, and returns the pair $\langle \sigma, F_i \rangle$. If no pattern matches P, an error is reported:

```
$check(P, cases) = let\ \ f([]) = error()$
                         $f(\langle \pi, F \rangle{::}L) = let\ \ \sigma = match(P, \pi, \emptyset)$
                                              $in$
                                                $if\ \ \sigma = false\ \ then\ \ f(L)\ \ else\ \langle \sigma, F \rangle$
                    $in$
                      $f(cases)$
```

Finally, there are two ternary functions *apply-prim-meth* and *apply-prim-fun* which take a primitive method M (or primitive function f) along with a list of values $[V_1, \ldots, V_n]$ and an assumption base

```
$\mathcal{I}(\mathbf{!}E\ \ F_1 \cdots F_n, \beta) = let\ \ V = \mathcal{I}(E, \beta)$
                                $\langle [V_1, \ldots, V_n], \beta' \rangle = eval\text{-}meth\text{-}args([F_1, \ldots, F_n], \beta, \mathcal{I})$
                            $in$
                              $Is\ V\ a\ primitive\ method\ M\,?$
                                $Yes\!: apply\text{-}prim\text{-}meth(M, [V_1, \ldots, V_n], \beta \cup \beta')$
                                $No\!:\ \ Is\ V\ a\ method\ of\ the\ form\ \phi\, I_1, \ldots, I_n\, .\, D\,?$
                                     $Yes\!: \mathcal{I}(D[V_1/I_1, \ldots, V_n/I_n], \beta \cup \beta')$
                                     $No\!:\ \ error()$
$\mathcal{I}(\mathbf{assume}\ E\ \mathbf{in}\ D) = let\ \ P = \mathcal{I}(E, \beta)$
                                $Q = \mathcal{I}(D, \beta \cup \{P\})$
                            $in$
                              $P \Rightarrow Q$
$\mathcal{I}(\mathbf{suppose\text{-}absurd}\ E\ \mathbf{in}\ D) = let\ \ P = \mathcal{I}(E, \beta)$
                                        $in$
                                          $if\ \mathcal{I}(D, \beta \cup \{P\}) = \mathbf{false}\ then\ \neg P\ \ else\ \ error()$
$\mathcal{I}(\mathbf{dlet}\ I = E\ \mathbf{in}\ D) = let\ V_E = \mathcal{I}(E, \beta)\ \ in\ \ \mathcal{I}(D[V_E/I], \beta)$
$\mathcal{I}(\mathbf{dlet}\ I = D'\ \mathbf{in}\ D) = let\ P = \mathcal{I}(D', \beta)\ \ in\ \ \mathcal{I}(D[P/I], \beta \cup \{P\})$
$\mathcal{I}(E\ \mathbf{by}\ D) = let\ \langle P, Q \rangle = \langle \mathcal{I}(D, \beta), \mathcal{I}(E, \beta) \rangle\ \ in\ \ (if\ P = Q\ \ then\ \ P\ \ else\ \ error())$
$\mathcal{I}(\mathbf{begin}\ D_1; \ldots; D_n\ \mathbf{end}, \beta) = let\ f([D], \beta') = \mathcal{I}(D, \beta')$
                                          $f(D_1{::}D_2{::}L, \beta') = (let\ P = \mathcal{I}(D_1, \beta')\ \ in\ \ f(D_2{::}L, \beta' \cup \{P\}))$
                                        $in$
                                          $f([D_1, \ldots, D_n], \beta)$
$\mathcal{I}(\mathbf{dmatch}\ \ E\ \ (\pi_1?\ \ D_1) \cdots (\pi_n?\ \ D_n), \beta) = let\ P = \mathcal{I}(E, \beta)$
                                                    $\langle \{\langle I_1, P_1 \rangle, \ldots, \langle I_k, P_k \rangle\}, D \rangle = check(P, [\langle \pi_1, D_1 \rangle, \ldots, \langle \pi_n, D_n \rangle])$
                                                  $in$
                                                    $\mathcal{I}(D[P_1/I_1, \ldots, P_k/I_k], \beta)$
$\mathcal{I}(c, \beta) = c$
$\mathcal{I}(I, \beta) = error()$
$\mathcal{I}(\phi\, I_1, \ldots, I_n\, .\, D, \beta) = \phi\, I_1, \ldots, I_n\, .\, D$
$\mathcal{I}(\lambda\, I_1, \ldots, I_n\, .\, E, \beta) = \lambda\, I_1, \ldots, I_n\, .\, E$
$\mathcal{I}(E\ \ F_1, \ldots, F_n, \beta) = let\ \ V = \mathcal{I}(E, \beta)$
                                $[V_1, \ldots, V_n] = eval\text{-}fun\text{-}args([F_1, \ldots, F_n], \beta, \mathcal{I})$
                            $in$
                              $Is\ V\ a\ primitive\ function\ f\,?$
                                $Yes\!: apply\text{-}prim\text{-}fun(f, [V_1, \ldots, V_n], \beta)$
                                $No\!:\ \ Is\ V\ a\ function\ of\ the\ form\ \lambda\, I_1, \ldots, I_n\, .\, E'\,?$
                                     $Yes\!: \mathcal{I}(E'[V_1/I_1, \ldots, V_n/I_n], \beta)$
                                     $No\!:\ \ error()$
$\mathcal{I}(\mathbf{let}\ I = F\ \mathbf{in}\ E, \beta) = let\ V_F = \mathcal{I}(F, \beta)\ \ in\ \ \mathcal{I}(E[V_F/I], \beta)$
$\mathcal{I}(\mathbf{fix}\, I\, .\, E, \beta) = \mathcal{I}(E[\mathbf{fix}\, I\, .\, E/I], \beta)$
$\mathcal{I}(\mathbf{begin}\ E_1; \ldots; E_n\ \mathbf{end}, \beta) = let\ f([E]) = \mathcal{I}(E, \beta)$
                                          $f(E_1{::}E_2{::}L) = (let\ \_ = \mathcal{I}(E_1, \beta)\ \ in\ \ f(E_2{::}L))$
                                        $in$
                                          $f([E_1, \ldots, E_n])$
$\mathcal{I}(\mathbf{match}\ \ E\ \ (\pi_1\ \ E_1) \cdots (\pi_n\ \ E_n), \beta) = let\ P = \mathcal{I}(E, \beta)$
                                                $\langle \{\langle I_1, P_1 \rangle, \ldots, \langle I_k, P_k \rangle\}, E \rangle = check(P, [\langle \pi_1, E_1 \rangle, \ldots, \langle \pi_n, E_n \rangle])$
                                              $in$
                                                $\mathcal{I}(E[P_1/I_1, \ldots, P_k/I_k], \beta)$
```

Figure 1.5: An interpreter for $\mathcal{NDL}_0^{\omega}$.

β and apply M (or f) to $[V_1, \ldots, V_n]$ in β. These functions are defined by a case analysis of M and f. We illustrate *apply-prim-meth* with a couple of cases; the rest are straightforward exercises:

$apply\text{-}prim\text{-}meth(\mathbf{claim}, [P], \beta) = if\ P \in \beta\ then\ P\ \ else\ error()$
$apply\text{-}prim\text{-}meth(\mathbf{both}, [P_1, P_2], \beta) = if\ \{P_1, P_2\} \subseteq \beta\ then\ P_1 \wedge P_2\ \ else\ error()$
$apply\text{-}prim\text{-}meth(\mathbf{left\text{-}and}, [P_1 \wedge P_2], \beta) = if\ P_1 \wedge P_2 \in \beta\ then\ P_1\ \ else\ error()$
$apply\text{-}prim\text{-}meth(\mathbf{modus\text{-}ponens}, [P_1 \Rightarrow P_2, P1], \beta) = if\ \{P_1 \Rightarrow P_2, P_1\} \subseteq \beta\ then\ P_2\ \ else\ error()$

$\vdots$

$apply\text{-}prim\text{-}meth(\_, \_, \_) = error()$

The definition of *apply-prim-fun* is even simpler, since there are only five propositional connectives and the equality function:

$\mathit{apply\text{-}prim\text{-}fun}(\equiv, [P_1, P_2], \_) = \mathit{if}\ P_1 = P_2\ \mathit{then}\ \mathbf{true}\ \mathit{else}\ \mathbf{false}$
$\mathit{apply\text{-}prim\text{-}fun}(\neg, [P], \_) = \neg P$
$\mathit{apply\text{-}prim\text{-}fun}(\odot, [P_1, P_2], \_) = P_1 \odot P_2$

for $\odot \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$. Observe that the assumption base is not used at all in the application of the primitive functions. Such primitive functions are called "context-independent" [4].

The following theorem expresses the correctness of $\mathcal{I}$ with respect to the formal semantics of the language. In particular, if the computation of $\mathcal{I}(F, \beta)$ terminates successfully with some output value V, then the theorem assures us that the judgment $\beta \vdash F \rightsquigarrow V$ is derivable. On the other hand, if the computation of $\mathcal{I}(F, \beta)$ halts in error or gets into an infinite loop, the theorem then entails that there is no value V such that $\beta \vdash F \rightsquigarrow V$. For if such a value exists, the interpreter will eventually find it. (Note that Theorem 1.1 now follows immediately: since $\mathcal{I}$ is deterministic there can be at most one value V such that $\mathcal{I}(F, \beta) = V$; hence, from Theorem 1.8, there can be at most one V such that $\beta \vdash D \rightsquigarrow V$.)

Theorem 1.8 *$\beta \vdash F \rightsquigarrow V$ iff $\mathcal{I}(F, \beta) = V$.*

Proof: By induction on the structure of F. ■

We will say that a phrase F *converges* in an assumption base β iff the computation of $\mathcal{I}(F, \beta)$ terminates; and we will say that F *fails* or *diverges* in β iff the computation of $\mathcal{I}(F, \beta)$ generates an error or continues ad infinitum, respectively. For a phrase F that converges in β, we define $\mathcal{V}(F, \beta)$, *the value of F in β*, as the result V produced by $\mathcal{I}(F, \beta)$. By the foregoing theorem, this result will be none other than the unique value V such that $\beta \vdash F \rightsquigarrow V$. If F is a deduction then $\mathcal{V}(F, \beta)$ will be a proposition, and we will refer to it as the *conclusion* of F in β. If F fails or diverges in β, then $\mathcal{V}(F, \beta)$ is undefined.

Finally, we will say that two phrases F_1 and F_2 are *observationally equivalent* with respect to some β, written $F_1 \approx_\beta F_2$, iff $\mathcal{I}(F_1, \beta) = \mathcal{I}(F_2, \beta)$, where this equation is understood to hold iff $\mathcal{I}(F_1, \beta)$ and $\mathcal{I}(F_2, \beta)$ produce the same value V; or both of them fail; or both of them diverge. And we will say that F_1 and F_2 are observationally equivalent iff $F_1 \approx_\beta F_2$ for *all* β.

1.8.2 Evaluation complexity

In this section we will define $\mathcal{IC}$, an instrumented version of the interpreter $\mathcal{I}$ which returns not just the value of a given phrase F in some β, but also an integer $c \geq 0$ representing the "cost" of evaluating F in β. More precisely, whenever $\mathcal{IC}(F, \beta)$ terminates successfully it produces a pair $\langle V, c\rangle$, where V is the value of F in β and c is its "cost". We will denote this latter quantity by $\mathcal{C}(F, \beta)$. Of course when $\mathcal{IC}$ fails or diverges, $\mathcal{C}(F, \beta)$ is undefined. The definition of $\mathcal{IC}$ appears in Figures 1.6 and 1.7.

The auxiliary functions *cost-eval-fun-args*, *cost-eval-meth-args*, *cost-match*, and *cost-check* are obtained by simple modifications of their previous corresponding definitions. In particular, the first two functions need to return an integer reflecting the total cost of evaluating the given arguments; *cost-match* needs to return an integer indicating the cost of matching a proposition to a pattern (even if the matching was unsuccessful); and *cost-check* needs to sum up the cost of all the matching that took place in examining the various cases. The new definitions are shown in Figure 1.8.

For every primitive function f and primitive method M we introduce quantities *prim-fun-cost*(f) and *prim-meth-cost*(M) that represent the cost of one single application of f and M, respectively. For a constructor $\odot \in \{\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow\}$, we set *prim-fun-cost*$(\odot) = 0$. In other words, we consider constructor applications to be "free". In tandem with the fact that all constants have zero evaluation

```
$\mathcal{IC}(\mathbf{!}E\ \ F_1 \cdots F_n, \beta) =$ let $\langle V, c\rangle = \mathcal{IC}(E, \beta)$
                                    $\langle [V_1, \ldots, V_n], \beta', c'\rangle =$ cost-eval-meth-args$([F_1, \ldots, F_n], \beta, \mathcal{IC})$
                                in
                                  Is $V$ a primitive method $M$?
                                    Yes: $\langle$apply-prim-meth$(M, [V_1, \ldots, V_n], \beta \cup \beta')$, prim-meth-cost$(M) + c + c'\rangle$
                                    No: Is $V$ a method of the form $\phi\, I_1, \ldots, I_n\, .\, D$?
                                          Yes: let $\langle V_D, c_D\rangle = \mathcal{IC}(D[V_1/I_1, \ldots, V_n/I_n], \beta \cup \beta')$
                                                 in
                                                   $\langle V_D, c + c' + c_D + 1\rangle$
                                          No: error()
$\mathcal{IC}(\mathbf{assume}\ E\ \mathbf{in}\ D) =$ let $\langle P, c_E\rangle = \mathcal{IC}(E, \beta)$
                                  $\langle Q, c_D\rangle = \mathcal{IC}(D, \beta \cup \{P\})$
                              in
                                $\langle P \Rightarrow Q, c_E + c_D + 1\rangle$
$\mathcal{IC}(\mathbf{suppose\text{-}absurd}\ E\ \mathbf{in}\ D) =$ let $\langle P, c_E\rangle = \mathcal{IC}(E, \beta)$
                                          $\langle Q, c_D\rangle = \mathcal{IC}(D, \beta \cup \{P\})$
                                      in
                                        if $Q = \mathbf{false}$ then $\langle \neg P, c_E + c_D + 1\rangle$ else error()
$\mathcal{IC}(\mathbf{dlet}\ I = E\ \mathbf{in}\ D) =$ let $\langle V_E, c_E\rangle = \mathcal{IC}(E, \beta)$
                                  $\langle P, c_D\rangle = \mathcal{IC}(D[V_E/I], \beta)$
                              in
                                $\langle P, c_E + c_D + 1\rangle$
$\mathcal{IC}(\mathbf{dlet}\ I = D'\ \mathbf{in}\ D) =$ let $\langle P, c\rangle = \mathcal{IC}(D', \beta)$
                                  $\langle Q, c_D\rangle = \mathcal{IC}(D[P/I], \beta \cup \{P\})$
                              in
                                $\langle Q, c + c_D + 1\rangle$
$\mathcal{IC}(E\ \mathbf{by}\ D) =$ let $\langle P, c_D\rangle = \mathcal{IC}(D, \beta)$
                        $\langle Q, c_E\rangle = \mathcal{IC}(E, \beta)$
                    in
                      If $P = Q$ then $\langle P, c_E + c_D + 1\rangle$ else error()
$\mathcal{IC}(\mathbf{begin}\ D_1; \ldots; D_n\ \mathbf{end}, \beta) =$ let $f([D], \beta', c') = ($let $\langle P, c\rangle = \mathcal{IC}(D, \beta')$ in $\langle P, c + c'\rangle)$
                                          $f(D_1{::}D_2{::}L, \beta', c') = ($let $\langle P, c\rangle = \mathcal{IC}(D_1, \beta')$ in $f(D_2{::}L, \beta' \cup \{P\}, c + c'))$
                                      in
                                        $f([D_1, \ldots, D_n], \beta, 0)$
$\mathcal{IC}(\mathbf{dmatch}\ \ E\ \ (\pi_1?\ D_1) \cdots (\pi_n?\ D_n), \beta) =$
    let $\langle P, c_E\rangle = \mathcal{IC}(E, \beta)$
        $\langle \{\langle I_1, P_1\rangle, \ldots, \langle I_k, P_k\rangle\}, D, c_m\rangle =$ cost-check$(P, [\langle \pi_1, D_1\rangle, \ldots, \langle \pi_n, D_n\rangle])$
        $\langle Q, c\rangle = \mathcal{IC}(D[P_1/I_1, \ldots, P_k/I_k], \beta)$
    in
      $\langle Q, c_E + c_m + c + 1\rangle$
```

Figure 1.6: Cost-instrumented interpreter, deductive part.

cost, this entails that the cost of building a proposition is zero. For the equality function we set *prim-fun-cost*$(\equiv) = 1$, so that checking whether two propositions are identical takes one "unit" of work. Finally, we define *prim-meth-cost*$(M) = 1$ for every primitive method M. Thus one application of **modus-ponens**, for example, takes one unit of work. More refined cost measures for the primitives could be given without affecting the sequel.

The cost of a method application

$$\mathbf{!}E\ \ F_1 \cdots F_n \tag{1.14}$$

in which E produces a primitive method M equals the sum of the cost of evaluating E, plus the total cost of producing the values $V_1, \ldots, V_n$ of the arguments $F_1, \ldots, F_n$, plus 1 (the cost of applying M to the resulting values); while if E produces a method $\phi\, I_1, \ldots, I_n\, .\, D$, the cost of 1.14 is the sum of the cost of E, plus the cost of the arguments, plus the cost of evaluating $D[V_1/I_1, \ldots, V_n/I_n]$, plus 1 for the substitution. Likewise, the cost of **dlet** $I = F$ **in** D is the sum of the cost of F plus the cost of D plus 1 for the substitution cost of replacing every free occurrence of I within D by the value of F. The rest

```
$\mathcal{IC}(c, \beta) = \langle c, 0 \rangle$
$\mathcal{IC}(I, \beta) = \mathit{error}()$
$\mathcal{IC}(\phi\, I_1, \ldots, I_n \,.\, D, \beta) = \langle \phi\, I_1, \ldots, I_n \,.\, D, 0 \rangle$
$\mathcal{IC}(\lambda\, I_1, \ldots, I_n \,.\, E, \beta) = \langle \lambda\, I_1, \ldots, I_n \,.\, E, 0 \rangle$
$\mathcal{IC}(E \;\; F_1, \ldots, F_n, \beta) = \mathit{let}\ \langle V, c \rangle = \mathcal{IC}(E, \beta)$
                                    $\langle [V_1, \ldots, V_n], c' \rangle = \mathit{cost\text{-}eval\text{-}fun\text{-}args}([F_1, \ldots, F_n], \beta, \mathcal{IC})$
                                  $\mathit{in}$
                                    $\mathit{Is\ V\ a\ primitive\ function\ f?}$
                                      $\mathit{Yes{:}}\ \langle \mathit{apply\text{-}prim\text{-}fun}(f, [V_1, \ldots, V_n], \beta), \mathit{prim\text{-}fun\text{-}cost}(f) + c + c' \rangle$
                                      $\mathit{No{:}\ \ Is\ V\ a\ function\ of\ the\ form}\ \lambda\, I_1, \ldots, I_n \,.\, E'\,?$
                                            $\mathit{Yes{:}\ let}\ \langle V'_E, c'_E \rangle = \mathcal{IC}(E'[V_1/I_1, \ldots, V_n/I_n], \beta)$
                                                $\mathit{in}$
                                                  $\langle V'_E, c + c' + c'_E + 1 \rangle$
                                            $\mathit{No{:}\ \ error}()$
$\mathcal{IC}(\mathbf{let}\ I = F\ \mathbf{in}\ E, \beta) = \mathit{let}\ \langle V_F, c_F \rangle = \mathcal{IC}(F, \beta)$
                                      $\langle V_E, c_E \rangle = \mathcal{IC}(E[V_F/I], \beta)$
                                   $\mathit{in}$
                                     $\langle V_E, c_E + c_F + 1 \rangle$
$\mathcal{IC}(\mathbf{fix}\, I \,.\, E, \beta) = \mathit{let}\ \langle V, c \rangle = \mathcal{IC}(E[\mathbf{fix}\, I \,.\, E/I], \beta)\ \ \mathit{in}\ \ \langle V, c + 1 \rangle$
$\mathcal{IC}(\mathbf{begin}\ E_1; \ldots; E_n\ \mathbf{end}, \beta) = \mathit{let}\ f([E], c') = (\mathit{let}\ \langle V, c \rangle = \mathcal{IC}(E, \beta)\ \mathit{in}\ \langle V, c + c' \rangle)$
                                            $f(E_1{::}E_2{::}L, c') = (\mathit{let}\ \langle V, c \rangle = \mathcal{IC}(E_1, \beta)\ \mathit{in}\ f(E_2{::}L, c + c'))$
                                         $\mathit{in}$
                                           $f([E_1, \ldots, E_n], 0)$
$\mathcal{IC}(\mathbf{match}\ \ E\ \ (\pi_1?\ E_1) \cdots (\pi_n?\ E_n), \beta) = \mathit{let}\ \langle P, c_E \rangle = \mathcal{IC}(E, \beta)$
                                                      $\langle \{\langle I_1, P_1 \rangle, \ldots, \langle I_k, P_k \rangle\}, E', c_m \rangle = \mathit{cost\text{-}check}(P, [\langle \pi_1, E_1 \rangle, \ldots, \langle \pi_n, E_n \rangle])$
                                                      $\langle V, c' \rangle = \mathcal{IC}(E'[P_1/I_1, \ldots, P_k/I_k], \beta)$
                                                    $\mathit{in}$
                                                      $\langle V, c_E + c_m + c' + 1 \rangle$
```

Figure 1.7: Cost-instrumented interpreter, computational part.

of the clauses can be similarly understood. As an example, let P be $(\neg\neg A \Rightarrow B \vee \neg\neg A) \wedge (B \Leftrightarrow C)$, let β contain the equivalence $\neg\neg A \Leftrightarrow A$, and let D be the deduction

$$\mathbf{!}\,\mathit{replace}\ \ \lceil P \rceil\ \ \neg\neg A \Leftrightarrow A$$

(where *replace* is the method defined in page 27). In our implementation of $\mathcal{IC}$, we find the cost of evaluating D in an appropriate β to be 281.

We also define two instrumented interpreters $\mathcal{IC}_d$ and $\mathcal{IC}_e$ for computing the "deductive" and "computational" cost of a phrase F in some β, respectively, shown in Figures 1.9 and 1.10. Intuitively, the deductive cost of a proof D is the cost of every primitive method call that is made during the evaluation of D, as well as the cost of every hypothetical deduction and proof by contradiction that is performed during that evaluation. We do not "count" anything else, such as the cost of pattern matching or the cost of substitutions. Thus, loosely speaking, the deductive cost of D accounts for the strictly logical work done by D that is deductively necessary for the derivation of the conclusion, and for nothing else. By fiat, the deductive cost of an expression E is zero, since an expression does not deduce anything. This entails that phantom deductions are not considered to have any deductive cost, which is appropriate since such deductions do not make any logical contributions to enclosing deductions. When the computation of $\mathcal{IC}_d(F, \beta)$ produces a pair $\langle V, c \rangle$, we write $\mathcal{C}_d(F, \beta)$ to denote the integer c; otherwise $\mathcal{C}_d(F, \beta)$ is undefined.

The computational cost of a phrase F is in some sense the inverse of its deductive cost. Intuitively, it accounts for whatever work F does for computational rather than for deductive purposes. We thus have $\mathcal{IC}_e(E, \beta) = \mathcal{IC}(E, \beta)$ for all expressions E. That is, the computational cost of an expression E is equal to the entire cost of E. The interesting case is the definition of the computational cost of a

```
$cost\text{-}eval\text{-}fun\text{-}args(phrases, \beta, \mathcal{IC}) = let\, f([], vals, c) = \langle \overleftarrow{vals}, c\rangle$
                                              $f(F{::}L, vals, c) = let\ \langle V_F, c_F\rangle = \mathcal{IC}(F, \beta)$
                                                                    $in$
                                                                      $f(L, V_F{::}vals, c + c_F)$
                                          $in$
                                            $f(phrases, [], 0)$
$cost\text{-}eval\text{-}meth\text{-}args(phrases, \beta, \mathcal{IC}) = let\, f([], vals, \beta', c) = \langle \overleftarrow{vals}, \beta', c\rangle$
                                               $f(E{::}L, vals, \beta', c) = let\ \langle V_E, c_E\rangle = \mathcal{IC}(E, \beta)$
                                                                        $in$
                                                                          $f(L, V_E{::}vals, \beta', c + c_E)$
                                               $f(D{::}L, vals, \beta', c) = let\ \langle P, c_D\rangle = \mathcal{IC}(D, \beta)$
                                                                        $in$
                                                                          $f(L, P{::}vals, \beta' \cup \{P\}, c + c_D)$
                                           $in$
                                             $f(phrases, [], \emptyset, 0)$
$cost\text{-}match(P, \pi, \sigma) = let\, f(A, A, \sigma, c) = (\sigma, c + 1)$
                             $f(\mathbf{true}, \mathbf{true}, \sigma, c) = (\sigma, c + 1)$
                             $f(\mathbf{false}, \mathbf{false}, \sigma, c) = (\sigma, c + 1)$
                             $f(P, \star, \sigma, c) = (\sigma, c + 1)$
                             $f(P, I, \sigma, c) = if\ \sigma\ contains\ a\ binding\ (I, Q)\ then$
                                     $(if\ P = Q\ then\ \langle \sigma, c + 2\rangle\ else\ \langle false, c + 2\rangle)\ else\ \langle false, c + 1\rangle$
                             $f(P_1 \odot P_2, \pi_1 \odot \pi_2, \sigma, c) = let\ \langle \sigma_1, c_1\rangle = f(P_1, \pi_1, \sigma, c)$
                                                             $in$
                                                               $if\ \sigma_1 = false\ then\ \langle \sigma_1, c_1\rangle\ else\ f(P_2, \pi_2, \sigma_1, c_1)$
                             $f(\_, \_, \_, c) = \langle false, c\rangle$
                           $in$
                             $f(P, \pi, \sigma, 0)$
$cost\text{-}check(P, cases) = let\, f([], c) = error()$
                               $f(\langle \pi, F\rangle{::}L, c) = let\ \langle \sigma, c'\rangle = cost\text{-}match(P, \pi)$
                                                       $in$
                                                         $if\ \ \sigma = fail\ \ then\ \ f(L, c + c')\ else\ \langle \sigma, F, c + c'\rangle$
                            $in$
                              $f(cases, 0)$
```

Figure 1.8: Auxiliary definitions.

deduction D, which measures essentially the "search" performed by D. That is, the computational cost of D represents the interpreter work done during the evaluation of D that could be eliminated by a more focused proof that nevertheless uses the same basic reasoning. Indeed, we will see that this is precisely the work that is eliminated when we generate a certificate for the original proof. The computational cost of F in β will be denoted by $\mathcal{C}_e(F, \beta)$. That is, $\mathcal{C}_e(F, \beta)$ denotes the integer part of the result produced by $\mathcal{IC}_d(F, \beta)$, when the latter terminates successfully. When it does not, $\mathcal{C}_e(F, \beta)$ is undefined.

We now have one "standard" interpreter $\mathcal{I}$ and three instrumented versions thereof: $\mathcal{IC}$, $\mathcal{IC}_d$, and $\mathcal{IC}_e$. It is somewhat tedious but not difficult to prove that every one of these interpreters converges (respectively, fails or diverges) on a given F and β iff all three of the other interpreters also converge (respectively, fail or diverge) on F and β; and that if one of them does terminate on some F and β, then the value V it produces is identical to the value produced by the other three interpreters for F and β. Accordingly, when convergence is taken for granted we will speak of *the* value V of a phrase F in an assumption base β without bothering to specify whether V is obtained by $\mathcal{I}$, $\mathcal{IC}$, $\mathcal{IC}_e$, or $\mathcal{IC}_d$.

Theorem 1.9 $\mathcal{C}(F, \beta) = \mathcal{C}_e(F, \beta) + \mathcal{C}_d(F, \beta)$ *whenever F converges in β.*

Proof: When F is an expression E the equality is immediate since $\mathcal{C}_d(E, \beta) = 0$ and $\mathcal{C}_e(E, \beta) = \mathcal{C}(E, \beta)$. For a deduction D we proceed by strong induction on the quantity $\mathcal{C}(D, \beta)$. We illustrate

```
$\mathcal{IC}_d(\mathbf{!}E\ \ F_1 \cdots F_n, \beta) = let\ V = \mathcal{I}(E,\beta)$
                                    $\langle [V_1,\ldots,V_n], \beta', c\rangle = cost\text{-}eval\text{-}meth\text{-}args([F_1,\ldots,F_n],\beta,\mathcal{IC}_d)$
                                  $in$
                                    $Is\ V\ a\ primitive\ method\ M?$
                                      $Yes{:}\ \langle apply\text{-}prim\text{-}meth(M,[V_1,\ldots,V_n],\beta\cup\beta'), prim\text{-}meth\text{-}cost(M)+c\rangle$
                                      $No{:}\ \ Is\ V\ a\ method\ of\ the\ form\ \phi\, I_1,\ldots,I_n\ .\ D?$
                                            $Yes{:}\ let\ \langle V_D, c_D\rangle = \mathcal{IC}_d(D[V_1/I_1,\ldots,V_n/I_n],\beta\cup\beta')$
                                                 $in$
                                                   $\langle V_D, c + c_D\rangle$
                                            $No{:}\ \ error()$
$\mathcal{IC}_d(\mathbf{assume}\ E\ \mathbf{in}\ D) = let\ P = \mathcal{I}(E,\beta)$
                                    $\langle Q, c\rangle = \mathcal{IC}_d(D,\beta\cup\{P\})$
                                  $in$
                                    $\langle P \Rightarrow Q, c+1\rangle$
$\mathcal{IC}_d(\mathbf{suppose\text{-}absurd}\ E\ \mathbf{in}\ D) = let\ P = \mathcal{I}(E,\beta)$
                                            $\langle Q, c\rangle = \mathcal{IC}_d(D,\beta\cup\{P\})$
                                          $in$
                                            $if\ Q = \mathbf{false}\ then\ \langle\neg P, c+1\rangle\ else\ error()$
$\mathcal{IC}_d(\mathbf{dlet}\ I = E\ \mathbf{in}\ D) = \mathcal{IC}_d(D[\mathcal{I}(E,\beta)/I],\beta)$
$\mathcal{IC}_d(\mathbf{dlet}\ I = D'\ \mathbf{in}\ D) = let\ \langle P, c'\rangle = \mathcal{IC}_d(D',\beta)$
                                      $\langle Q, c\rangle = \mathcal{IC}_d(D[P/I],\beta\cup\{P\})$
                                    $in$
                                      $\langle Q, c'+c\rangle$
$\mathcal{IC}_d(E\ \mathbf{by}\ D) = let\ \langle P, c\rangle = \mathcal{IC}_d(D,\beta)$
                            $Q = \mathcal{I}(E,\beta)$
                          $in$
                            $If\ P = Q\ then\ \langle P, c\rangle\ else\ error()$
$\mathcal{IC}_d(\mathbf{begin}\ D_1;\ldots;D_n\ \mathbf{end},\beta) = let\ f([D],\beta',c') = (let\ \langle P,c\rangle = \mathcal{IC}_d(D,\beta')\ in\ \langle P, c+c'\rangle)$
                                          $f(D_1{::}D_2{::}L,\beta',c') = (let\ \langle P,c\rangle = \mathcal{IC}_d(D_1,\beta')\ in\ f(D_2{::}L,\beta'\cup\{P\},c+c'))$
                                        $in$
                                          $f([D_1,\ldots,D_n],\beta,0)$
$\mathcal{IC}_d(\mathbf{dmatch}\ \ E\ \ (\pi_1?\ D_1)\cdots(\pi_n?\ D_n),\beta) = let\ P = \mathcal{I}(E,\beta)$
                                                          $\langle\{\langle I_1,P_1\rangle,\ldots,\langle I_k,P_k\rangle\}, D\rangle = check(P,[\langle\pi_1,D_1\rangle,\ldots,\langle\pi_n,D_n\rangle])$
                                                        $in$
                                                          $\mathcal{IC}_d(D[P_1/I_1,\ldots,P_k/I_k],\beta)$
$\mathcal{IC}_d(E,\beta) = \langle\mathcal{I}(E,\beta),0\rangle$
```

Figure 1.9: Instrumented interpreter for computing the deductive cost of a phrase.

with method calls $\mathbf{!}E\ \ F_1 \cdots F_n$. When $\mathcal{IC}(E,\beta)$ results in a primitive method M, we have

$$\mathcal{C}(F,\beta) = \mathcal{C}(E,\beta) + \sum_{i=1}^{n} \mathcal{C}(F_i,\beta) + 1. \tag{1.15}$$

Moreover,

$$\mathcal{C}_e(F,\beta) = \mathcal{C}(E,\beta) + \sum_{i=1}^{n} \mathcal{C}_e(F_i,\beta) \tag{1.16}$$

and

$$\mathcal{C}_d(F,\beta) = \sum_{i=1}^{n} \mathcal{C}_d(F_i,\beta) + 1. \tag{1.17}$$

Inductively, we have

$$\sum_{i=1}^{n} \mathcal{C}(F_i,\beta) = \sum_{i=1}^{n} [\mathcal{C}_e(F_i,\beta) + \mathcal{C}_d(F_i,\beta)]$$

```
$\mathcal{IC}_e(\textbf{!}E\ F_1 \cdots F_n, \beta) =$ let $\langle V, c\rangle = \mathcal{IC}(E, \beta)$
                                   $\langle [V_1, \ldots, V_n], \beta', c'\rangle =$ cost-eval-meth-args$([F_1, \ldots, F_n], \beta, \mathcal{IC}_e)$
                                 in
                                   Is $V$ a primitive method $M$?
                                     Yes: $\langle$apply-prim-meth$(M, [V_1, \ldots, V_n], \beta \cup \beta'), c + c'\rangle$
                                     No: Is $V$ a method of the form $\phi\, I_1, \ldots, I_n \,.\, D$?
                                           Yes: let $\langle P, c_D\rangle = \mathcal{IC}_e(D[V_1/I_1, \ldots, V_n/I_n], \beta \cup \beta')$
                                                  in
                                                    $\langle P, c + c' + c_D + 1\rangle$
                                           No: error()
$\mathcal{IC}_e(\textbf{assume}\ E\ \textbf{in}\ D) =$ let $\langle P, c\rangle = \mathcal{IC}_e(E, \beta)$
                                     $\langle Q, c'\rangle = \mathcal{IC}_e(D, \beta \cup \{P\})$
                                   in
                                     $\langle P \Rightarrow Q, c + c'\rangle$
$\mathcal{IC}_e(\textbf{suppose-absurd}\ E\ \textbf{in}\ D) =$ let $\langle P, c\rangle = \mathcal{IC}_e(E, \beta)$
                                           $\langle Q, c'\rangle = \mathcal{IC}_e(D, \beta \cup \{P\})$
                                         in
                                           if $Q = \textbf{false}$ then $\langle \neg P, c + c'\rangle$ else error()
$\mathcal{IC}_e(\textbf{dlet}\ I = E\ \textbf{in}\ D) =$ let $\langle V, c\rangle = \mathcal{IC}_e(E, \beta)$
                                     $\langle P, c'\rangle = \mathcal{IC}_e(D[V/I], \beta)$
                                   in
                                     $\langle P, c + c' + 1\rangle$
$\mathcal{IC}_e(\textbf{dlet}\ I = D\ \textbf{in}\ D') =$ let $\langle P, c\rangle = \mathcal{IC}_e(D, \beta)$
                                      $\langle Q, c'\rangle = \mathcal{IC}_e(D'[P/I], \beta \cup \{P\})$
                                    in
                                      $\langle Q, c + c' + 1\rangle$
$\mathcal{IC}_e(E\ \textbf{by}\ D) =$ let $\langle P, c\rangle = \mathcal{IC}_e(D, \beta)$
                             $\langle Q, c'\rangle = \mathcal{IC}_e(E, \beta)$
                           in
                             If $P = Q$ then $\langle P, c + c' + 1\rangle$ else error()
$\mathcal{IC}_e(\textbf{begin}\ D_1; \ldots; D_n\ \textbf{end}, \beta) =$ let $f([D], \beta', c') =$ (let $\langle P, c\rangle = \mathcal{IC}_e(D, \beta')$ in $\langle P, c + c'\rangle$)
                                          $f(D_1{::}D_2{::}L, \beta', c') =$ (let $\langle P, c\rangle = \mathcal{IC}_e(D_1, \beta')$ in $f(D_2{::}L, \beta' \cup \{P\}, c + c')$)
                                        in
                                          $f([D_1, \ldots, D_n], \beta, 0)$
$\mathcal{IC}_e(\textbf{dmatch}\ E\ (\pi_1?\ D_1) \cdots (\pi_n?\ D_n), \beta) =$
    let $\langle P, c\rangle = \mathcal{IC}_e(E, \beta)$
        $\langle \{\langle I_1, P_1\rangle, \ldots, \langle I_k, P_k\rangle\}, D, c'\rangle =$ cost-check$(P, [\langle \pi_1, D_1\rangle, \ldots, \langle \pi_n, D_n\rangle])$
        $\langle Q, c''\rangle = \mathcal{IC}_e(D[P_1/I_1, \ldots, P_k/I_k], \beta)$
    in
      $\langle Q, c + c' + c'' + 1\rangle$
$\mathcal{IC}_e(E, \beta) = \mathcal{IC}(E, \beta)$
```

Figure 1.10: Instrumented interpreter for obtaining the computational cost of a phrase.

and therefore

$$\mathcal{C}(E,\beta) + \sum_{i=1}^{n} \mathcal{C}(F_i,\beta) + 1 = \mathcal{C}(E,\beta) + \sum_{i=1}^{n} [\mathcal{C}_e(F_i,\beta) + \mathcal{C}_d(F_i,\beta)] + 1$$

which by virtue of 1.15—1.17 means that

$$\mathcal{C}(F,\beta) = \mathcal{C}_e(F,\beta) + \mathcal{C}_d(F,\beta).$$

When $\mathcal{IC}(E,\beta)$ results in a method $\phi\, I_1, \ldots, I_n \,.\, D$ we have

$$\mathcal{C}(F,\beta) = \mathcal{C}(E,\beta) + \sum_{i=1}^{n} \mathcal{C}(F_i,\beta) + \mathcal{C}(D[V_1/I_1, \ldots, V_n/I_n], \beta \cup \beta') + 1 \tag{1.18}$$

where $V_1, \ldots, V_n$ are the values of $F_1, \ldots, F_n$ in β and β' is the lemma set obtained by evaluating those phrases amongst $F_1, \ldots, F_n$ that are deductions. Furthermore,

$$\mathcal{C}_e(F,\beta) = \mathcal{C}(E,\beta) + \sum_{i=1}^{n} \mathcal{C}_e(F_i,\beta) + \mathcal{C}_e(D[V_1/I_1,\ldots,V_n/I_n],\beta \cup \beta') + 1 \tag{1.19}$$

and

$$\mathcal{C}_d(F,\beta) = \sum_{i=1}^{n} \mathcal{C}_d(F_i,\beta) + \mathcal{C}_d(D[V_1/I_1,\ldots,V_n/I_n],\beta \cup \beta'). \tag{1.20}$$

Again the inductive hypothesis entails

$$\sum_{i=1}^{n} \mathcal{C}(F_i,\beta) = \sum_{i=1}^{n} [\mathcal{C}_e(F_i,\beta) + \mathcal{C}_d(F_i,\beta)]$$

and

$$\mathcal{C}(D[V_1/I_1,\ldots,V_n/I_n],\beta \cup \beta') = \mathcal{C}_e(D[V_1/I_1,\ldots,V_n/I_n],\beta \cup \beta') + \mathcal{C}_d(D[V_1/I_1,\ldots,V_n/I_n],\beta \cup \beta')$$

and hence the desired equality $\mathcal{C}(F,\beta) = \mathcal{C}_e(F,\beta) + \mathcal{C}_d(F,\beta)$ follows from 1.18—1.20. ■

1.9 Relationship to type-α DPLs and certificates

Every type-ω DPL properly contains a type-α DPL in a sense that will be illustrated in this section with $\mathcal{NDL}_0^\omega$. It is for this reason that type-ω DPLs are superior; they can achieve everything that type-α DPLs achieve, namely, perspicuous proof presentation and efficient checking, plus a good deal more. Specifically, one could use a type-ω DPL $\mathcal{L}$ exclusively for proof presentation and checking, if so desired, simply by restricting attention to the type-α subset of $\mathcal{L}$. But, in addition, as we have already seen, one could also use methods and the type-ω features of $\mathcal{L}$ to formulate powerful derived inference rules and theorem provers with a strong soundness guarantee. In practice this ability is widely exercised and greatly facilitates the derivation of non-trivial theorems.

In short, the only advantages of type-α DPLs are: (a) they are very easy to implement; and (b) they are easy to study and understand because their behavior is so straightforward. These advantages could be critical, e.g., if we desire an exceptionally small and simple proof checker in order to minimize our trusted base, or if a DPL is being introduced to students in an elementary logic course. When these considerations are not particularly important, type-ω DPLs are to be preferred for the aforementioned reasons.

In fact when it comes to minimizing our trusted base, we will shortly show that type-ω DPLs allow us to have our cake and eat it too: it is straightforward to make a terminating type-ω proof D automatically generate an equivalent type-α proof $\mathbf{D}$—a so-called *certificate*. While we need a type-ω interpreter $\mathcal{I}$ in order to generate the certificate $\mathbf{D}$, we need not *trust* $\mathcal{I}$, for once we have $\mathbf{D}$ we can check it with a type-α interpreter $\mathcal{I}'$, obtaining the same conclusion that is produced by D. Thus in the end the only component that we have to trust is $\mathcal{I}'$, which, being a type-α interpreter, is eminently simple. We will discuss this in more detail shortly.

We begin by observing that the following subset of $\mathcal{NDL}_0^\omega$ is essentially identical to $\mathcal{NDL}_0$:

$$D ::= \mathbf{!}c\ P_1 \cdots P_n \mid \textbf{assume}\ P\ \textbf{in}\ D \mid \textbf{suppose-absurd}\ P\ \textbf{in}\ D \mid \textbf{begin}\ D_1; \ldots; D_n\ \textbf{end} \tag{1.21}$$

where the constant c ranges over the primitive methods of Figure 1.1.[7] $\mathcal{NDL}_0^\omega$ deductions generated by the above grammar will be called *pure*. We write $\mathfrak{PD}$ for the set of all pure $\mathcal{NDL}_0^\omega$ deductions and use the letters $\mathbf{D}, \mathbf{D}_1, \mathbf{D}', \ldots$ to range over $\mathfrak{PD}$.

The claim that $\mathcal{NDL}_0$ and $\mathfrak{PD}$ are "essentially identical" can be formalized by giving a meaning-preserving isomorphism $\mathcal{T}$ between the two sets of deductions. We define $\mathcal{T}$ as a simple desugaring that maps any $\mathcal{NDL}_0$ deduction into a pure $\mathcal{NDL}_0^\omega$ deduction as follows:

$$\mathcal{T}[\![\textit{Prim-Rule } P_1, \ldots, P_n]\!] = \mathbf{!}\,\textit{Prim-Rule } P_1, \ldots, P_n \tag{1.22}$$

$$\mathcal{T}[\![\textbf{assume } P \textbf{ in } D]\!] = \textbf{assume } P \textbf{ in } \mathcal{T}[\![D]\!] \tag{1.23}$$

$$\mathcal{T}[\![\textbf{suppose-absurd } P \textbf{ in } D]\!] = \textbf{suppose-absurd } P \textbf{ in } \mathcal{T}[\![D]\!] \tag{1.24}$$

$$\mathcal{T}[\![D_1; D_2]\!] = \textbf{begin } \mathcal{T}[\![D_1]\!]; \mathcal{T}[\![D_2]\!] \textbf{ end} \tag{1.25}$$

Under this mapping, for instance, the $\mathcal{NDL}_0$ proof

assume $A \wedge B$ **in**
  begin
    right-and $A \wedge B$;
    left-and $A \wedge B$;
    both B, A
  end

corresponds to the pure $\mathcal{NDL}_0^\omega$ proof

assume $A \wedge B$ **in**
  begin
    !right-and $A \wedge B$;
    !left-and $A \wedge B$;
    !both $B\ A$
  end

The two proofs look virtually identical and behave in the exact same way. In general, if we restrict ourselves to pure $\mathcal{NDL}_0^\omega$ deductions we are essentially using $\mathcal{NDL}_0$. The proofs will be perspicuous—albeit potentially long and tedious—and proof checking will be guaranteed to terminate in linear time on average. The following lemma asserts the correctness of $\mathcal{T}$; it can be proved by a straightforward induction on D:

Lemma 1.10 (Pure deduction isomorphism) *$\beta \vdash_{\mathcal{NDL}_0} D \leadsto P$ iff $\beta \vdash_{\mathcal{NDL}_0^\omega} \mathcal{T}[\![D]\!] \leadsto P$.*

The notion of size for $\mathcal{NDL}_0$ deductions carries over directly to pure $\mathcal{NDL}_0^\omega$ deductions:

$$\begin{aligned} SZ(\mathbf{!}c\ P_1 \cdots P_n) &= n \\ SZ(\textbf{assume } P \textbf{ in } \mathbf{D}) &= 1 + SZ(\mathbf{D}) \\ SZ(\textbf{suppose-absurd } P \textbf{ in } \mathbf{D}) &= 1 + SZ(\mathbf{D}) \\ SZ(\textbf{begin } \mathbf{D}_1; \ldots; \mathbf{D}_n \textbf{ end}) &= SZ(\mathbf{D}_1) + \cdots + SZ(\mathbf{D}_n) \end{aligned}$$

The next lemma shows that a pure deduction $\mathbf{D}$ has zero computational cost; and that its deductive cost is $\Theta(SZ(\mathbf{D}))$. Accordingly, checking a pure deduction requires exactly as much effort as the size of the deduction.

[7] In what follows it will be convenient to treat **suppose-absurd** as a primitive construct of $\mathcal{NDL}_0$; this does not affect our discussion in any substantial way.

Lemma 1.11 *We have $\mathcal{C}_e(\mathbf{D}, \beta) = 0$ for every pure $\mathbf{D}$. Accordingly, $\mathcal{C}(\mathbf{D}, \beta) = \mathcal{C}_d(\mathbf{D}, \beta)$. In addition, $\mathcal{C}_d(\mathbf{D}, \beta) = \Theta(SZ(\mathbf{D}))$, and therefore $\mathcal{C}(\mathbf{D}, \beta) = \Theta(SZ(\mathbf{D}))$ for all β.*

We now turn our attention to the subject of certificates:

Lemma 1.12 (Certificate Lemma) *For every convergent deduction D there is an equivalent pure deduction $\mathbf{D}$. More precisely, for all D, if $\beta \vdash D \leadsto P$ then there is a pure $\mathbf{D}$ such that $\beta \vdash \mathbf{D} \leadsto P$. We say that $\mathbf{D}$ is a* **certificate** *for D.*

The lemma itself is not surprising at all. It is a direct corollary of the completeness of $\mathcal{NDL}_0$ in tandem with the soundness of $\mathcal{NDL}_0^\omega$: if $\beta \vdash_{\mathcal{NDL}_0^\omega} D \leadsto P$ then, by the soundness of $\mathcal{NDL}_0^\omega$, we have $\beta \models P$; but then, by the completeness of $\mathcal{NDL}_0$, there is a $\mathcal{NDL}_0$ proof D' such that $\beta \vdash_{\mathcal{NDL}_0} D' \leadsto P$, and therefore $\mathcal{T}[\![D']\!]$ is the desired pure $\mathcal{NDL}_0^\omega$ deduction. What is more interesting for our purposes is that certificates can be mechanically constructed for arbitrary deductions. Specifically, in Figure 1.11 we instrument the interpreter $\mathcal{I}$ of Section 1.8.1 to arrive at a new interpreter $\mathcal{IX}$ that produces not only the conclusion of a deduction D in a given β, but also a pure deduction $\mathbf{D}$ that is observationally equivalent to D in β. The definition of $\mathcal{IX}$ requires the following modified version of *eval-meth-args*:

```
$\textit{cert-eval-meth-args}(\textit{phrases}, \beta, \mathcal{IX})$ = $\textit{let}$ $f([], \textit{vals}, \beta', DL) = \langle \overleftarrow{\textit{vals}}, \beta', \overleftarrow{DL} \rangle$
                                          $f(E{::}L, \textit{vals}, \beta', DL) = f(L, \mathcal{I}(E, \beta){::}\textit{vals}, \beta', DL)$
                                          $f(D{::}L, \textit{vals}, \beta', DL) =$ $\textit{let}$ $\langle P, \mathbf{D} \rangle = \mathcal{IX}(D, \beta)$
                                                                        $\textit{in}$
                                                                          $f(L, P{::}\textit{vals}, \beta' \cup \{P\}, \mathbf{D}{::}DL)$
                                       $\textit{in}$
                                         $f(\textit{phrases}, [], \emptyset, [])$
```

It is a straightforward induction on D to prove that $\mathcal{I}$ converges (respectively, fails or diverges) on given D and β iff $\mathcal{IX}$ converges (respectively, fails or diverges) on D and β; and furthermore, that $\mathcal{I}(D, \beta) = P$ iff $(\exists\, \mathbf{D})\, \mathcal{IX}(D, \beta) = \langle P, \mathbf{D} \rangle$. We will thus continue to speak of a deduction D converging on some β without specifying whether we are referring to the convergence of $\mathcal{IX}$ or of $\mathcal{I}$ (or, for that matter, of $\mathcal{IC}$, $\mathcal{IC}_d$, or $\mathcal{IC}_e$). And likewise, we will speak of "the value" of a D in some β without bothering to specify whether the value is obtained from $\mathcal{IX}$ or from $\mathcal{I}$.

For any D that converges on β we define $\mathcal{X}(D, \beta)$ as the pure deduction produced by running $\mathcal{IX}$ on D and β. More precisely,

$$\mathcal{X}(D, \beta) = \mathbf{D} \text{ iff } (\exists\, P)\, \mathcal{IX}(D, \beta) = \langle P, \mathbf{D} \rangle.$$

We call $\mathcal{X}(D, \beta)$ the "certificate" or the *expansion* of D in β. Accordingly, the observational equivalence of D and the pure deduction produced by $\mathcal{IX}(D, \beta)$ is expressed by the following equation:

$$\mathcal{I}(\mathcal{X}(D, \beta), \beta) = \mathcal{I}(D, \beta).$$

The certificate $\mathcal{X}(D, \beta)$ can be seen as the "trace" or "yield" produced by evaluating D in β. In a sense, it is the justification that the instrumented interpreter $\mathcal{IX}$ produces in order to back up its conclusion. It consists of every primitive-method application as well as every hypothetical deduction and proof by contradiction in non-phantom positions performed during the evaluation of D in β, stringed together in temporal order.

It is of interest to consider the complexity relationship between a deduction D and its certificate $\mathcal{X}(D, \beta)$ in some β. It is clear that evaluating D in β must be at least as expensive as evaluating the certificate $\mathcal{X}(D, \beta)$ in β, since every step made by the certificate must also be made by the original D. But how much cheaper $\mathcal{X}(D, \beta)$ will be depends largely on how focused D is. In general, when we produce a certificate for some D we "throw away" all the computation performed by D and keep only

```
$\mathcal{IX}(\mathbf{!}E\ \ F_1 \cdots F_n, \beta) = let\ \ V = \mathcal{I}(E,\beta)$
                              $\langle [V_1,\ldots,V_n], \beta', [\mathbf{D}_1,\ldots,\mathbf{D}_k]\rangle = cert\text{-}eval\text{-}meth\text{-}args([F_1,\ldots,F_n],\beta,\mathcal{IX})$
                          $in$
                            $Is\ V\ a\ primitive\ method\ M?$
                              $Yes: \langle apply\text{-}prim\text{-}meth(M,[V_1,\ldots,V_n],\beta\cup\beta'), \mathbf{begin}\ \mathbf{D}_1;\ldots,\mathbf{D}_k; \mathbf{!}M\ \ V_1,\ldots,V_k\ \mathbf{end}\rangle$
                              $No:\ \ Is\ V\ a\ method\ of\ the\ form\ \phi\, I_1,\ldots,I_n\,.\,D?$
                                  $Yes: let\ \langle V_D, \mathbf{D}\rangle = \mathcal{IX}(D[V_1/I_1,\ldots,V_n/I_n],\beta\cup\beta')$
                                        $in$
                                          $\langle V_D, \mathbf{begin}\ \mathbf{D}_1;\ldots,\mathbf{D}_k;\mathbf{D};\ \mathbf{end}\rangle$
                                  $No:\ \ error()$
$\mathcal{IX}(\mathbf{assume}\ E\ \mathbf{in}\ D) = let\ P = \mathcal{I}(E,\beta)$
                               $\langle Q,\mathbf{D}\rangle = \mathcal{IX}(D,\beta\cup\{P\})$
                           $in$
                             $\langle P \Rightarrow Q, \mathbf{assume}\ P\ \mathbf{in}\ \mathbf{D}\rangle$
$\mathcal{IX}(\mathbf{suppose\text{-}absurd}\ E\ \mathbf{in}\ D) = let\ P = \mathcal{I}(E,\beta)$
                                     $\langle \mathbf{false},\mathbf{D}\rangle = \mathcal{IX}(D,\beta\cup\{P\})$
                                 $in$
                                   $\langle \neg P, \mathbf{suppose\text{-}absurd}\ P\ \mathbf{in}\ \mathbf{D}\rangle$
$\mathcal{IX}(\mathbf{dlet}\ I = E\ \mathbf{in}\ D) = let\ V_E = \mathcal{I}(E,\beta)$
                               $\langle P,\mathbf{D}\rangle = \mathcal{IX}(D[V_E/I],\beta)$
                           $in$
                             $\langle P,\mathbf{D}\rangle$
$\mathcal{IX}(\mathbf{dlet}\ I = D'\ \mathbf{in}\ D) = let\ \langle P,\mathbf{D}'\rangle = \mathcal{IX}(D',\beta)$
                                $\langle Q,\mathbf{D}\rangle = \mathcal{IX}(D[P/I],\beta\cup\{P\})$
                            $in$
                              $\langle Q, \mathbf{begin}\ \mathbf{D}';\mathbf{D}\ \mathbf{end}\rangle$
$\mathcal{IX}(E\ \mathbf{by}\ D) = let\ \langle P,\mathbf{D}\rangle = \mathcal{IX}(D,\beta)$
                      $Q = \mathcal{I}(E,\beta)$
                  $in$
                    $If\ P = Q\ then\ \langle P,\mathbf{D}\rangle\ else\ error()$
$\mathcal{IX}(\mathbf{begin}\ D_1;\ldots;D_n\ \mathbf{end},\beta) = let\ \ f([D],\beta',[\mathbf{D}_1,\ldots,\mathbf{D}_k]) = let\ \langle P,\mathbf{D}\rangle = \mathcal{IX}(D,\beta')$
                                                                    $in$
                                                                      $\langle P, \mathbf{begin}\ \mathbf{D}_k;\ldots;\mathbf{D}_1;\mathbf{D}\ \mathbf{end}\rangle$
                                         $f(D_1{::}D_2{::}L,\beta',DL) = let\ \langle P,\mathbf{D}_1\rangle = \mathcal{IX}(D_1,\beta')$
                                                                  $in$
                                                                    $f(D_2{::}L,\beta'\cup\{P\},\mathbf{D}_1{::}DL))$
                                     $in$
                                       $f([D_1,\ldots,D_n],\beta,[])$
$\mathcal{IX}(\mathbf{dmatch}\ \ E\ \ (\pi_1?\ D_1)\cdots(\pi_n?\ D_n),\beta) = let\ P = \mathcal{I}(E,\beta)$
                                                    $\langle\{\langle I_1,P_1\rangle,\ldots,\langle I_k,P_k\rangle\},D\rangle = check(P,[\langle\pi_1,D_1\rangle,\ldots,\langle\pi_n,D_n\rangle])$
                                                $in$
                                                  $\mathcal{IX}(D[P_1/I_1,\ldots,P_k/I_k],\beta)$
```

Figure 1.11: Certificate-generating interpreter for $\mathcal{NDL}_0^\omega$.

those primitive inferences that are essential for the derivation of the final conclusion.[8] If D performs a lot of computation, e.g., a lot of pattern matching, then the savings will be substantial because all that computation will be discarded. But if D is fairly focused and does not perform many gratuitous calculations then the savings will be minimal. Indeed, in the extreme case D might be pure to begin with, in which case it performs zero computation (Lemma 1.11) and can serve as its own certificate; in that case we will have $\mathcal{X}(D,\beta) = D$ and the savings will be zero.

Lemma 1.13 *$\mathcal{C}_d(D,\beta) = \mathcal{C}(\mathcal{X}(D,\beta),\beta)$.*

[8]This is not to say that the resulting certificate will be an optimal deduction in any sense. Far from it, it will likely have several sources of redundancy naively created by the expansion algorithm, although most of these can be mechanically removed by a small number of simple optimizing transformations; see Chapter 5 of "Denotational Proof Languages" [4].

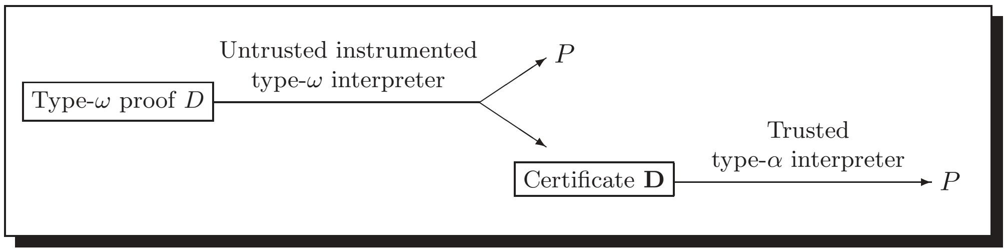

Figure 1.12: The generation and validation of certificates.

The above intuitions can be made precise as follows. For any D that is convergent in a given β, let us define the *speed-up* of D in β, denoted $SU(D,\beta)$, as

$$SU(D,\beta) = \mathcal{C}(D,\beta) - \mathcal{C}(\mathcal{X}(D,\beta),\beta).$$

This simply says that the speed-up achieved by the certificate is equal to its cost difference from the original D (this quantity will always be non-negative). Now the statement that "certificates throw away computation" is formally captured by the following result:

Theorem 1.14 *$SU(D,\beta) = \mathcal{C}_e(D,\beta)$. In words, the speed-up achieved by "purifying" D is precisely equal to the computational cost of D.*

Proof: By definition,

$$SU(D,\beta) = \mathcal{C}(D,\beta) - \mathcal{C}(\mathcal{X}(D,\beta),\beta). \tag{1.26}$$

By Lemma 1.13, $\mathcal{C}(\mathcal{X}(D,\beta),\beta) = \mathcal{C}_d(D,\beta)$, hence 1.26 gives

$$SU(D,\beta) = \mathcal{C}(D,\beta) - \mathcal{C}_d(D,\beta). \tag{1.27}$$

But by Theorem 1.9, $\mathcal{C}(D,\beta) = \mathcal{C}_e(D,\beta) + \mathcal{C}_d(D,\beta)$, and thus 1.27 yields $SU(D,\beta) = \mathcal{C}_e(D,\beta)$. ■

Besides increased efficiency, certificates have the advantage of being pure deductions, and hence very simple and easy to trust. This means that a certificate-generating type-ω interpreter such as $\mathcal{IX}$ allows us to have the best of both worlds—the expressive power of type-ω DPLs and the simplicity of type-α DPLs. That is, we can write a deduction D in a type-ω DPL, making full use of the convenience of methods, recursion, etc., and once D is functional and correctly produces the desired conclusion P, we can run it through the certificate-generating interpreter and obtain an equivalent pure deduction $\mathbf{D}$. Then if we have to convince a skeptical agent S about P, we simply submit the certificate $\mathbf{D}$ as our evidence. S can check $\mathbf{D}$ with a type-α interpreter for pure deductions, which can be implemented in about one page of code. That one page of code is ultimately the only component that S needs to trust. This process, depicted in Figure 1.12, is the main idea behind the DPL approach to *certified computation* [3].

1.10 A hierarchy of DPLs

$\mathcal{NDL}_0^\omega$ deductions can be ordered along a continuum on the basis of how much computation they perform. On one end of the spectrum we have pure deductions, which have zero computational cost.

Pure deductions are focused, proceed in very small steps, and can be checked in linear time. On the other end we have full-blown type-ω proofs, which use unrestricted iteration and conditional branching and can thus incur arbitrarily large computational cost. Such proofs can expend a lot of effort on searching, can take large steps via derived inference rules (defined methods), and cannot in general be checked efficiently. In between these two extremes lies a hierarchy of classes of proofs, readily identifiable by syntactic criteria and characterized by the amount of computation that each allows. Three classes of proofs appear particularly interesting and useful:

1.10.1 Type-β proofs

We define a *propositional form* H as follows:

$$H ::= I \mid A \mid \textbf{true} \mid \textbf{false} \mid \neg H \mid H_1 \wedge H_2 \mid H_1 \vee H_2 \mid H_1 \Rightarrow H_2 \mid H_1 \Leftrightarrow H_2.$$

Hence, as abstract syntax trees, propositional forms have the same structure as propositions, except that identifiers are also allowed at the leaves along with atoms and the constants **true** and **false**. Now a type-β *proof* D_β is defined as follows:

$$\begin{array}{rcl} D_\beta & ::= & \mathbf{!}c \;\; H_1 \cdots H_k \mid \textbf{assume } H \textbf{ in } D_\beta \mid \textbf{suppose-absurd } H \textbf{ in } D_\beta \\ & \mid & \textbf{dlet } I_1 = K_1 \cdots I_n = K_n \textbf{ in } D_\beta \\ K_i & ::= & H \mid D_\beta \end{array}$$

As their syntax manifests, type-β proofs are very similar to pure deductions. The only difference is that **dlet** deductions are used instead of **begin** $D_1; \cdots ; D_n$ **end** blocks, which, in combination with propositional forms, allows for the ability to refer to propositions by name. This can reduce the size of pure deductions by replacing multiple occurrences of the same proposition by a name. As a simple example consider the following pure type-α deduction of $A \wedge B \Rightarrow B \wedge A$:

```
assume A ∧ B  in
  !both  (!right-and A ∧ B)
         (!left-and A ∧ B)
```

This can be expressed as a type-β deduction as follows:

```
dlet P = A ∧ B in
  assume P in
    !both (!right-and P) (!left-and P)
```

The space savings become more substantial of course as the size and number of occurences of the same proposition increase. The following can be proved by a straightforward induction on the structure of D_β:

Theorem 1.15 *A type-β deduction D_β can be checked in linear time on average.*

In situtations where one wants the simplest proofs possible in order to minimize the trust placed upon the proof checker but also wants to reduce the size of proofs as much as possible (perhaps because the proofs are to be shipped over a network), type-β proofs will be a better alternative than type-α proofs, because they are essentially just as simple as type-α proofs; they can be checked just as efficiently, according to the preceding result; and they achieve better compactness thanks to naming.

1.10.2 Type-γ proofs

Let $\mathcal{I} = \{I_1, \ldots, I_m\}$ be a set of identifiers. A type-γ proof *parameterized over* $\mathcal{I}$ is defined by the following abstract grammar:

$$\begin{array}{rcl}
D_\gamma[\mathcal{I}] & ::= & \mathbf{!}M \;\; H_1 \cdots H_k \mid \textbf{assume } H \textbf{ in } D_\gamma[\mathcal{I}] \mid \textbf{suppose-absurd } H \textbf{ in } D_\gamma[\mathcal{I}] \\
 & \mid & \textbf{dlet } I_1 = K_1 \cdots I_n = K_n \textbf{ in } D_\gamma[\mathcal{I}] \mid \textbf{dmatch } \; H \;\; (\pi_1 \;\; D^1_\gamma[\mathcal{I}]) \cdots (\pi_n \;\; D^n_\gamma[\mathcal{I}]) \\
K_i & ::= & H \mid D_\gamma[\mathcal{I}] \\
M & ::= & c \mid I_1 \mid \;\cdots\; \mid I_m
\end{array}$$

where H is a propositional form as defined previously. There are two notable differences from type-β proofs. First, method applications are of the form $\mathbf{!}M \;\; H_1 \cdots H_k$ where the expression M no longer has to be a constant; it can also be an identifier drawn from $\mathcal{I}$. Second, matching is allowed. A type-γ proof is now defined as a proof of the form

$$\begin{array}{ll}
\textbf{dlet} & M_1 = \phi\, I^1_1, \ldots, I^1_{k_1} \,.\, D^1_\gamma[\emptyset] \\
 & M_2 = \phi\, I^2_1, \ldots, I^1_{k_2} \,.\, D^2_\gamma[\{M_1\}] \\
 & \vdots \\
 & M_n = \phi\, I^n_1, \ldots, I^1_{k_n} \,.\, D^n_\gamma[\{M_1, \ldots, M_{n-1}\}] \\
\textbf{in} & \\
 & D_\gamma[\{M_1, \ldots, M_{n-1}, M_n\}]
\end{array}$$

Thus a type-γ proof introduces a series of methods $M_1, \ldots, M_n$, each of which can freely use any of the methods previously introduced. The body of M_1, in particular, may only use primitive methods. Because no form of iteration is allowed, evaluation still only needs to scan the proof tree once from top to bottom, which means that the proof can be checked in linear time (assuming $O(1)$ assumption-base queries). However, proofs now are not quite as simple as their type-α or type-β counterparts, since substitution and pattern matching are allowed.

Most methods that are obtainable via schematic abstraction can be expressed within type-γ proofs. For instance, here is a type-γ proof that formulates and applies modus tollens:

```
dlet $mt = \phi\, prem\text{-}1, prem\text{-}2 \,.$ dmatch $prem\text{-}1 \wedge prem\text{-}2$
                                   $(P \Rightarrow Q) \wedge \neg Q$ ?  suppose-absurd $P$ in
                                                     !absurd (!mp $prem\text{-}1$  $P$) $prem\text{-}2$
in
  assume $A \Rightarrow B$ in
    assume $\neg B$ in
      !$mt$  $A \Rightarrow B$  $\neg B$
```

In fact all of the methods we defined by schematic abstraction in Section 1.7 can be assembled and used in a type-γ proof:

```
dlet $un\text{-}curry = \cdots$
     $curry = \cdots$
     $dm\text{-}1 = \cdots$
     $dm\text{-}2 = \cdots$
     $dm\text{-}3 = \cdots$
     $dm\text{-}4 = \cdots$
     $dm = \cdots$
```

```
    excl-middle = ···
    cases = ···
    reflex-equiv = ···
    and-cong = ···
    ⋮
in
  ···
```

Thus type-γ proofs give us non-trivial abstraction capability while preserving efficiency—proofs can be written in a fluid style but can still be checked expeditiously. However, we are still unable to formulate recursive methods.

1.10.3 Type-δ or primitive recursive proofs

Type-δ proofs are similar to type-γ proofs, except that each method M_j is of the form

$$\mathbf{fix}\, M_j \,.\, \phi\, I^j \,.\, D_\gamma[M_1, \ldots, M_{j-1}, M_j]$$

where M_j may or may not have free occurences within $D_\gamma[M_1, \ldots, M_{j-1}, M_j]$ (if it does not, then it is a regular type-γ method, by the semantics of **fix**). If M_j is recursive then we require that every recursive call to it appears within a **dmatch** deduction in one of the following three forms:

1. **dmatch** $I^j \ \cdots (\neg Q\,? \ \cdots \mathbf{!} M_j \ Q \cdots)$; or
2. **dmatch** $I^j \ \cdots (Q_1 \odot Q_2)\,? \ \cdots \mathbf{!} M_j \ Q_1 \cdots)$; or
3. **dmatch** $I^j \ \cdots (Q_1 \odot Q_2)\,? \ \cdots \mathbf{!} M_j \ Q_2 \cdots)$,

for $\odot \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$. Accordingly, the method's formal parameter, I_j, must range over propositions, and the argument to every recursive call must be an immediate subtree of I_j, lexically obtained by decomposing I_j with patterns. Hence this kind of method recursion is the same kind of "primitive recursion" encountered in functions over Herbrand universes [22], whereby the argument to each recursive call is an immediate subterm of the parameter, obtained either via pattern matching or via the application of some appropriate selector function such as "`tail`" or "`left-tree`". (Primitive recursion on Herbrand terms is a natural generalization of primitive recursion on the non-negative integers; instead of recursing on the predecessor of a number, we recurse on the tail of a list, the left or right branch of a binary tree, and so on. The connection to our case is apparent once we view propositions as a term algebra.)

Because the size of the argument strictly decreases with each recursive call, termination is guaranteed for type-δ proofs. A typical type-δ method is the *recurse* method written given within *equiv-cong*.

Continuing in this fashion we can extend the hierarchy by defining type-ϵ proofs, type-ζ proofs, and so on, with each level allowing for more computation than its predecessors and eventually culminating in unrestricted type-ω proofs that may diverge. We stress that all of these classes of proofs are equally powerful from a deductive viewpoint, meaning that they are all logically complete; if a conclusion follows logically from an assumption base then it can be derived by a type-α proof.[9] The difference lies in expressiveness, abstraction, and usability. Type-α proofs can be checked quickly but can also

[9]However, the classes are certainly not equally powerful from the extensional viewpoint of what proposition-yielding functions one can formulate in them as methods. For instance, it is not difficult to see that there are methods which one can define in unrestricted type-ω $\mathcal{NDL}_0^\omega$ that cannot be expressed as type-δ methods.

be tedious to write for non-trivial theorems. As we start moving to the type-β and type-γ levels, the proofs become smaller and easier to write but somewhat more expensive to check; in a sense, we are trading space for time. The tradeoff continues to intensify as we move through the various levels and reaches its peak with unrestricted type-ω proofs, which may exchange infinitely much space for infinitely much time. This hierarchical classification also serves as a rigorous characterization of the intuitive distinction between proof checking, which is generally considered to be computationally tractable, and automated theorem proving, which is computationally expensive. The hierarchy that extends from type-α proofs to type-ω proofs essentially captures the spectrum that lies between proof checking and automated deduction.

1.11 A general framework for type-ω DPLs

A cursory inspection of the syntax and semantics of $\mathcal{NDL}_0^\omega$ will reveal little that is specific to classical propositional logic. The fragments that are specific to that logic are small and easy to isolate:

1. Most of the constants—the propositional atoms, the constants **true** and **false**, the five propositional constructors, and the primitive methods.

2. The definition of what constitutes a proposition, as given by 1.3.

3. The syntax forms **assume** and **suppose-absurd**.

The essential ideas behind $\mathcal{NDL}_0^\omega$—most notably, the two-tier syntax and the semantic abstraction of assumption bases—are independent of any particular logic. Indeed, every syntactic form of $\mathcal{NDL}_0^\omega$ outside of **assume** and **suppose-absurd** is thoroughly generic. For instance, method applications $\mathbf{!}E\ F_1 \cdots F_n$ and their semantics, as given by rules $[R_1]$, $[R_2]$, and the cut rule $[R_7]$, have nothing to do with classical zero-order logic; they are applicable intact to any logic that induces a transitive provability relation. Likewise for deductions of the form **dlet** $I = F$ **in** D, E **by** D, **dmatch** $F\ (\pi_1\ D_1) \cdots (\pi_n\ D_n)$, and so on.

This means that the same core syntax and semantics may be used for a wide variety of logics. We need only change the constants, in order to introduce different propositional constructors and different primitive methods. Occasionally we may also need to introduce some special syntax forms for deductions, such as the **assume** construct, in order to capture certain modes of inference that are difficult or impossible to model with primitive methods. But this is a type-α concern. Once a type-α DPL has been formulated for a given logic, lifting it to the type-ω level is standard and immediate.

This observation led to the development of the *$\lambda\phi$-calculus* [4], an abstract framework for type-ω DPLs.[10] The $\lambda\phi$-calculus has the same advantages for the study of type-ω DPLs that the λ-calculus has for the study of functional programming languages: it offers a terse formal system that strips away inessential features and crystallizes the key ideas of type-ω DPLs in a single uniform framework in which the theory of such DPLs can be studied in isolation from any particular language. What we gain is generality and a deeper understanding of the essential issues. For instance, the soundness result of Theorem 1.4 can be obtained in a more general form in the setting of the $\lambda\phi$-calculus—we can show that the provability relation induced by *any* type-ω DPL is the least Tarskian relation that respects the primitive methods and special syntax forms of the language. The soundness of a particular type-ω DPL then follows as a corollary simply by demonstrating the soundness of its primitive methods and special forms, which is usually quite straightforward (e.g., no inductive argument is required).

[10]This system originally appeared under the name "$\lambda\mu$-calculus" in "Denotational Proof Languages" [4]; the μ operator was subsequently changed to ϕ to avoid confusion with Parigot's $\lambda\mu$-calculus [26].

Indeed, most of the concepts that we have introduced in this paper, such as the notions of deductive and computational cost, the hierarchy of type-β DPLs, type-γ DPLs, and so forth, can be defined and studied in the abstract setting of the $\lambda\phi$-calculus.

1.12 Static type systems for type-ω DPLs

$\mathcal{NDL}_0^\omega$ is an untyped language. It is a distinguishing feature of DPLs that they achieve soundness dynamically, by virtue of their assumption-base evaluation semantics, rather than by virtue of a static type system. This is in marked contrast to LCF-based systems such as HOL, whose soundness depends to a large extent on a specific static type discipline.

Nevertheless, strongly typed DPLs are also possible. Many different type disciplines could be imposed on any given DPL, and in this section we will illustrate this by formulating a polymorphic Hindley-Milner type system for $\mathcal{NDL}_0^\omega$. We also present a type inference procedure for this system, along the lines of Milner's original algorithm, that infers the most general type of a phrase. Thus this type system is *à la* Curry [6]: types need not be explicitly declared by the $\mathcal{NDL}_0^\omega$ user; they can be automatically inferred by the system.

We stress that a type system is *not* necessary for the logical soundness of the language, which is guaranteed by its evaluation semantics. In particular, there is no need for any type "`theorem`". The benefit that a static type system would confer on a DPL is similar to the benefit conferred by such systems on programming languages: it would statically weed out a certain class of type errors (in the case of a DPL, such as applying **modus-ponens** to an integer) which could otherwise only be detected at run-time. To what extent static type systems are advantageous is a controversial subject, and it is not our intention here to take sides. We only wish to show that it is straightforward to define sophisticated static type disciplines for type-ω DPLs. (In fact, because soundness is guaranteed by the dynamic semantics, DPLs are left with greater freedom to choose a static type system. Whereas HOL or Isabelle must by necessity use a specific type system, a DPL is not locked into any particular choice.)

We define the types of $\mathcal{NDL}_0^\omega$ by the following grammar:

$$\tau ::= \alpha \mid \mathbf{prop} \mid \tau_1 \times \cdots \times \tau_n \rightarrow \tau \mid \tau_1 \times \cdots \times \tau_n \twoheadrightarrow \mathbf{prop} \mid (\forall\, \alpha)\, \tau$$

where α ranges over an unspecified countably infinite set of *type variables*, disjoint from the set of identifiers. Free and bound occurrences of a type variable in a given type are defined in the usual manner; e.g., in the type $(\alpha_1 \times \alpha_2) \rightarrow (\forall\, \alpha_2)\, (\alpha_2 \rightarrow \alpha_2)$, α_1 occurs free while α_2 occurs both free and bound. We will write $FV(\tau)$ for the set of type variables that occur free in τ. Types of the form $\tau_1 \times \cdots \times \tau_n \twoheadrightarrow \mathbf{prop}$ will describe methods. Since a method must always return a proposition, the type **prop** appears by default to the right of the arrow $\twoheadrightarrow$. Types of the form $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$ describe functions as usual.

By a *type substitution* (or just "substitution") θ we will mean a function from the set of type variables to the set of types that is the identity almost everywhere. For distinct $\alpha_1, \ldots, \alpha_n$, we write

$$\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$$

for the substitution that maps each α_i to τ_i, and every other type variable to itself. Thus $\{\}$ denotes the identity function on the set of type variables. Further, we write $\theta[\alpha \mapsto \tau]$ for the substitution that maps α to τ and every other type variable α' to $\theta(\alpha')$. Given a substitution θ, we define a function $\overline{\theta}$

from the set of all types to itself as follows:

$$\begin{aligned}
\overline{\theta}(\alpha) &= \theta(\alpha)\\
\overline{\theta}(\mathbf{prop}) &= \mathbf{prop}\\
\overline{\theta}(\tau_1 \times \cdots \times \tau_n \rightarrow \tau) &= \overline{\theta}(\tau_1) \times \cdots \times \overline{\theta}(\tau_n) \rightarrow \overline{\theta}(\tau)\\
\overline{\theta}(\tau_1 \times \cdots \times \tau_n \twoheadrightarrow \mathbf{prop}) &= \overline{\theta}(\tau_1) \times \cdots \times \overline{\theta}(\tau_n) \twoheadrightarrow \mathbf{prop}\\
\overline{\theta}((\forall\, \alpha)\, \tau) &= (\forall\, \alpha)\, \overline{\theta[\alpha \mapsto \alpha]}(\tau)
\end{aligned}$$

The function $\overline{\theta}$ called the *lift* of θ.[11] Note that applying $\overline{\theta}$ to a type τ could result in variable capture; but in what follows we will make sure that this never happens.[12] For a list of types $L = [t_1, \ldots, t_n]$, we define $\overline{\theta}(L)$ as $[\overline{\theta}(t_1), \ldots, \overline{\theta}(t_n)]$. Finally, given any two substitutions θ_1, θ_2 we can define a new substitution $\theta_1 \circ \theta_2$ as:

$$\theta_1 \circ \theta_2 = \lambda\, \alpha\, .\, \overline{\theta_1}(\theta_2(\alpha)). \tag{1.28}$$

We refer to $\theta_1 \circ \theta_2$ as the *composition* of θ_1 and θ_2. (The letter λ in 1.28 is used as part of our metalanguage; it is not related to the λ of the object language $\mathcal{NDL}_0^{\omega}$.)

A type τ_2 is an *instance* of a type τ_1 iff there is a θ such that $\overline{\theta}(\tau_1) = \tau_2$. We say that τ_1 is *more general* that τ_2. For example, the type $\mathbf{prop} \rightarrow \mathbf{prop}$ is an instance of $\alpha \rightarrow \alpha$ under the substitution $\{\alpha \mapsto \mathbf{prop}\}$. Two types τ_1, τ_2 are *unifiable* iff there is a θ such that $\overline{\theta}(\tau_1) = \overline{\theta}(\tau_2)$.

The following algorithm U takes two quantifier-free types τ_1, τ_2 and produces a unifying substitution for them, if τ_1 and τ_2 are unifiable. If not, the algorithm raises an exception. The algorithm is written using ML-style pattern matching. Also, for a boolean-valued expression B, we write $B \Rightarrow E_1, E_2$ to mean "if B then E_1 else E_2".

```
$U(\mathbf{prop}, \mathbf{prop}) = \{\}$
$U(\alpha, \tau) = \textit{if}\ \ \alpha \notin FV(\tau)\ \textit{then}\ \{\alpha \mapsto \tau\}\ \textit{else}\ (\textit{if}\ \ \alpha = \tau\ \textit{then}\ \{\}\ \textit{else error}())$
$U(\tau, \alpha) = U(\alpha, \tau)$
$U(\tau_1 \times \cdots \tau_n \twoheadrightarrow \mathbf{prop}, \tau_1' \times \cdots \tau_n' \twoheadrightarrow \mathbf{prop}) = U^*([\tau_1, \ldots, \tau_n], [\tau_1', \ldots, \tau_n'])$
$U(\tau_1 \times \cdots \tau_n \rightarrow \tau, \tau_1' \times \cdots \tau_n' \rightarrow \tau') = U^*([\tau_1, \ldots, \tau_n, \tau], [\tau_1', \ldots, \tau_n', \tau'])$
$U(\_, \_) = \textit{error}()$
$U^*([], []) = \{\}$
$U^*(s{::}L_1, t{::}L_2) = \theta' \circ \theta$, where $\theta = U(s, t), \theta' = U^*(\overline{\theta}(L_1), \overline{\theta}(L_2))$
$U^*(\_, \_) = \textit{error}()$
```

By an "atomic type assignment" we will mean an ordered pair $\langle I, \tau \rangle$ consisting of an identifier I and a type τ. We will write such a pair more suggestively as $I : \tau$, and we will call I and τ the subject and value of that assignment, respectively. By a *type context* we will mean a finite list of atomic assignments:

$$[I_1 : \tau_1, \ldots, I_n : \tau_n]. \tag{1.29}$$

We will use the letter Γ to range over type contexts. For any given substitution θ and type context Γ of the form 1.29, we write $\overline{\theta}(\Gamma)$ for the context $[I_1 : \overline{\theta}(\tau_1), \ldots, I_n : \overline{\theta}(\tau_n)]$. The expression $FV(\Gamma)$ will denote the set of type variables that occur free in the value of some assignment in Γ. We write $\Gamma(I) = \tau$ to signify that $I : \tau$ is the first (leftmost) assignment in Γ with subject I. Finally, for a pattern π with $PI(\pi) = [I_1, \ldots, I_n]$, we define $\Delta(\pi)$ as the type context $[\langle I_1, \mathbf{prop}\rangle, \ldots, \langle I_n, \mathbf{prop}\rangle]$.

[11] If we exclude quantifications, the remaining types form a free term algebra over the set of type variables, and $\overline{\theta}$ then coincides with the unique homomorphic extension of θ. Milner's inference algorithm essentially ignores quantifications and treats the set of all types as a free algebra.

[12] Of course in practice this can always be ensured by alphabetically renaming τ before applying $\overline{\theta}$.

$$\frac{\Gamma \vdash E : \tau_1 \times \cdots \times \tau_n \twoheadrightarrow \mathbf{prop} \quad \Gamma \vdash F_1 : \tau_1 \;\; \cdots \;\; \Gamma \vdash F_n : \tau_n}{\Gamma \vdash \mathbf{!}E \;\; F_1 \cdots F_n : \mathbf{prop}} \; [T_1]$$

$$\frac{\Gamma \vdash E : \mathbf{prop} \quad \Gamma \vdash D : \mathbf{prop}}{\Gamma \vdash \mathbf{assume} \; E \; \mathbf{in} \; D : \mathbf{prop}} \; [T_2] \qquad \frac{\Gamma \vdash E : \mathbf{prop} \quad \Gamma \vdash D : \mathbf{prop}}{\Gamma \vdash \mathbf{suppose\text{-}absurd} \; E \; \mathbf{in} \; D : \mathbf{prop}} \; [T_3]$$

$$\frac{\Gamma \vdash F : \tau \quad \langle I, \tau \rangle{::}A \vdash D : \mathbf{prop}}{\Gamma \vdash \mathbf{dlet} \; I = F \; \mathbf{in} \; D : \mathbf{prop}} \; [T_4] \qquad \frac{\Gamma \vdash E : \mathbf{prop} \quad \Gamma \vdash D : \mathbf{prop}}{\Gamma \vdash E \; \mathbf{by} \; D : \mathbf{prop}} \; [T_5]$$

$$\frac{\Gamma \vdash F : \mathbf{prop} \quad \Delta(\pi_1) \oplus \Gamma \vdash D_1 : \mathbf{prop} \quad \Delta(\pi_n) \oplus \Gamma \vdash E_n : \mathbf{prop}}{\Gamma \vdash \mathbf{dmatch} \;\; F \;\; (\pi_1? \; D_1) \cdots (\pi_n? \; D_n) : \mathbf{prop}} \; [T_6]$$

$$\frac{}{\begin{array}{c}\Gamma \vdash I : \tau \\ \text{provided } \Gamma(I) = \tau\end{array}} \; [T_7] \qquad \frac{\langle I_1, \tau_1 \rangle{::} \cdots \langle I_n, \tau_n \rangle{::}\Gamma \vdash D : \mathbf{prop}}{\Gamma \vdash \phi \, I_1, \ldots, I_n \, . \, D : \tau_1 \times \cdots \times \tau_n \twoheadrightarrow \mathbf{prop}} \; [T_8]$$

$$\frac{\Gamma \vdash E : \tau_1 \times \cdots \times \tau_n \rightarrow \tau \quad \Gamma \vdash F_1 : \tau_1 \;\; \cdots \;\; \Gamma \vdash F_n : \tau_n}{\Gamma \vdash E \;\; F_1 \cdots F_n : \tau} \; [T_9]$$

$$\frac{\langle I, \tau \rangle{::}\Gamma \vdash E : \tau}{\Gamma \vdash \mathbf{fix} \, I \, . \, E : \tau} \; [T_{10}] \qquad \frac{\Gamma \vdash F : \tau \quad \langle I, \tau \rangle{::}A \vdash D : \mathbf{prop}}{\Gamma \vdash \mathbf{dlet} \; I = F \; \mathbf{in} \; D : \mathbf{prop}} \; [T_{11}]$$

$$\frac{\Gamma \vdash F : \mathbf{prop} \quad \Delta(\pi_1) \oplus \Gamma \vdash E_1 : \tau \quad \Delta(\pi_n) \oplus \Gamma \vdash D_n : \tau}{\Gamma \vdash \mathbf{match} \;\; F \;\; (\pi_1? \; E_1) \cdots (\pi_n? \; E_n) : \tau} \; [T_{12}]$$

$$\frac{\Gamma \vdash E : \tau}{\begin{array}{c}\Gamma \vdash E : (\forall \, \alpha) \, \tau \\ \text{provided } \alpha \notin FV(\Gamma)\end{array}} \; [T_{13}] \qquad \frac{\Gamma \vdash E : (\forall \, \alpha) \, \tau}{\Gamma \vdash E : \overline{\{\alpha \mapsto \tau'\}}(\tau)} \; [T_{14}]$$

Figure 1.13: Core rules of the static type system of $\mathcal{NDL}_0^{\omega}$.

The judgments of the type system are of the form $\Gamma \vdash F : \tau$, asserting that the phrase F has type τ with respect to the context Γ. The core rules of the system are shown in Figure 1.13. Type axioms for the constants are trivial: we have $\Gamma \vdash A : \mathbf{prop}$ for all atoms A, $\Gamma \vdash \neg : \mathbf{prop} \rightarrow \mathbf{prop}$, $\Gamma \vdash \odot : \mathbf{prop} \times \mathbf{prop} \rightarrow \mathbf{prop}$ for $\odot \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow, \equiv\}$, and $\Gamma \vdash c : \mathbf{prop}$ for $c \in \{\mathbf{true}, \mathbf{false}\}$. The types for the primitive methods are obvious, e.g., we have $\Gamma \vdash \mathbf{both} : \mathbf{prop} \times \mathbf{prop} \twoheadrightarrow \mathbf{prop}$. Note that deductions are always of type **prop**, since for all rules that establish conclusions of the form $\Gamma \vdash D : \tau$ (namely, rules $[T_1]$—$[T_6]$), we have $\tau = \mathbf{prop}$.

A type is called *shallow* iff it is of the form $(\forall \, \alpha_1) \cdots (\forall \, \alpha_n) \, \tau$ for some $n \geq 0$ and quantifier-free τ. That is, a type is shallow iff it has no quantifiers or else every quantifier is up front and its scope includes everything to its right. This type system allows us to prove that certain expressions have non-shallow types. For example, the "self-application" $\lambda \, I \, . \, I \; I$ can be shown to have the non-shallow

$\mathcal{W}(e, A) = \tau$, where $\langle \tau, \theta \rangle = \mathcal{V}(e, A)$ and

$\mathcal{V}(\mathbf{!}E\ \ F_1 \cdots F_n, \Gamma) = \langle \mathbf{prop}, \theta_2 \circ \theta_1 \rangle$, where $\langle \tau, \theta \rangle = \mathcal{V}(E, \Gamma)$, $\langle [\tau_1, \ldots, \tau_n], \theta_1 \rangle = \widehat{\mathcal{V}}([F_1, \ldots, F_n], [\Gamma, \ldots, \Gamma], \theta)$,
$\theta_2 = U(\overline{\theta_1}(\tau), \tau_1 \times \cdots \times \tau_n \twoheadrightarrow \mathbf{prop})$

$\mathcal{V}(\mathbf{assume}\ E\ \mathbf{in}\ D, \Gamma) = \langle \mathbf{prop}, \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1 \rangle$, where $\langle \tau, \theta_1 \rangle = \mathcal{V}(E, \Gamma)$, $\theta_2 = U(\tau_1, \mathbf{prop})$,
$\langle \tau', \theta_3 \rangle = \mathcal{V}(D, \overline{\theta_2 \circ \theta_1}(\Gamma))$, $\theta_4 = U(\tau', \mathbf{prop})$

$\mathcal{V}(\mathbf{suppose\text{-}absurd}\ E\ \mathbf{in}\ D, \Gamma) = \langle \mathbf{prop}, \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1 \rangle$, where $\langle \tau, \theta_1 \rangle = \mathcal{V}(E, \Gamma)$, $\theta_2 = U(\tau_1, \mathbf{prop})$,
$\langle \tau', \theta_3 \rangle = \mathcal{V}(D, \overline{\theta_2 \circ \theta_1}(\Gamma))$, $\theta_4 = U(\tau', \mathbf{prop})$

$\mathcal{V}(\mathbf{dlet}\ I = F\ \mathbf{in}\ D, \Gamma) = (\mathbf{prop}, \theta_3 \circ \theta_2 \circ \theta_1)$,
where $(\tau_1, \theta_1) = \mathcal{V}(F, \Gamma)$, $\Gamma' = \langle I, (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau_1 \rangle{::}\overline{\theta_1}(\Gamma)$, $(\tau_2, \theta_2) = \mathcal{V}(D, \Gamma')$, $\theta_3 = U(\tau_2, \mathbf{prop})$
and $\{\alpha_1, \ldots, \alpha_n\} = FV(\tau_1) - FV(\overline{\theta_1}(\Gamma))$

$\mathcal{V}(\mathbf{dmatch}\ E\ \ (\pi_1?\ D_1) \cdots (\pi_n?\ D_n), \Gamma) = \langle \mathbf{prop}, \theta' \circ \theta \rangle$, where $\langle \tau, \theta_1 \rangle = \mathcal{V}(E, \Gamma), \theta_2 = U(\tau, \mathbf{prop})$,
$[\Gamma_1, \ldots, \Gamma_n] = map\ \ \Delta\ \ [\pi_1, \ldots, \pi_n]$, $\langle [\tau_1, \ldots, \tau_n], \theta \rangle = \widehat{\mathcal{V}}([D_1, \ldots, D_n], [\Gamma_1 \oplus \Gamma, \ldots, \Gamma_n \oplus \Gamma], \theta_2 \circ \theta_1)$,
$\theta' = U^*([\overline{\theta}(\tau_1), \ldots, \overline{\theta}(\tau_n)], [\mathbf{prop}, \ldots, \mathbf{prop}])$

$\mathcal{V}(E\ \mathbf{by}\ D, \Gamma) = \langle \mathbf{prop}, \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1 \rangle$, where $\langle \tau, \theta_1 \rangle = \mathcal{V}(D, \Gamma)$, $\theta_2 = U(\tau, \mathbf{prop})$
$\langle \tau', \theta_3 \rangle = \mathcal{V}(E, \overline{\theta_2 \circ \theta_1}(\Gamma))$, $\theta_4 = U(\tau', \mathbf{prop})$

$\mathcal{V}(I, \Gamma) = \mathit{if}\ \ \Gamma(I) = (\forall\, \alpha_1) \cdots (\forall\, \alpha_k)\, \tau\ \ \mathit{then}\ \ \langle \overline{\{\alpha_1 \mapsto \alpha'_1, \ldots, \alpha_k \mapsto \alpha'_k\}}(\tau), \{\} \rangle\ \ \mathit{else}\ \ \mathit{error}()$,
where $\alpha'_1, \ldots, \alpha'_k$ are fresh, $k \geq 0$

$\mathcal{V}(\lambda\, I_1, \ldots, I_n\, .\, E, \Gamma) = \langle \overline{\theta}(\alpha_1 \times \cdots \times \alpha_n \to \tau), \theta \rangle$,
where $\langle \tau, \theta \rangle = \mathcal{V}(E, [\langle I_1, \alpha_1 \rangle, \ldots, \langle I_n, \alpha_n \rangle] \oplus \Gamma)$ and $\alpha_1, \ldots, \alpha_n$ are fresh

$\mathcal{V}(\phi\, I_1, \ldots, I_n\, .\, D, \Gamma) = \langle \overline{\theta_2}(\alpha_1) \times \cdots \times \overline{\theta_2}(\alpha_n) \twoheadrightarrow \mathbf{prop}, \theta_2 \rangle$,
where $\langle \tau, \theta \rangle = \mathcal{V}(D, [\langle I_1, \alpha_1 \rangle, \ldots, \langle I_n, \alpha_n \rangle] \oplus \Gamma), \theta_1 = U(\tau, \mathbf{prop}), \theta_2 = \theta_1 \circ \theta$, and $\alpha_1, \ldots, \alpha_n$ are fresh

$\mathcal{V}(E\ \ F_1 \cdots F_n, \Gamma) = \langle \overline{\theta_2}(\alpha), \theta_2 \circ \theta_1 \rangle$, where $\langle \tau, \theta \rangle = \mathcal{V}(E, \Gamma)$, $\langle [\tau_1, \ldots, \tau_n], \theta_1 \rangle = \widehat{\mathcal{V}}([F_1, \ldots, F_n], [\Gamma, \ldots, \Gamma], \theta)$,
$\theta_2 = U(\overline{\theta_1}(\tau), \tau_1 \times \cdots \times \tau_n \to \alpha)$, and α is fresh

$\mathcal{V}(\mathbf{fix}\, I\, .\, E, \Gamma) = \langle \theta_2(\alpha), \theta_2 \rangle$, where $\langle \tau, \theta \rangle = \mathcal{V}(E, \langle I, \alpha \rangle{::}\Gamma), \theta_1 = U(\tau, \theta(\alpha)), \theta_2 = \theta_1 \circ \theta$, and α is fresh

$\mathcal{V}(\mathbf{let}\ I = F\ \mathbf{in}\ E, \Gamma) = (\tau_2, \theta_2 \circ \theta_1)$,
where $(\tau_1, \theta_1) = \mathcal{V}(F, \Gamma)$, $\Gamma' = \langle I, (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau_1 \rangle{::}\overline{\theta_1}(\Gamma)$, $(\tau_2, \theta_2) = \mathcal{V}(E, \Gamma')$
and $\{\alpha_1, \ldots, \alpha_n\} = FV(\tau_1) - FV(\overline{\theta_1}(\Gamma))$

$\mathcal{V}(\mathbf{match}\ E\ \ (\pi_1?\ E_1) \cdots (\pi_n?\ E_n), \Gamma) = \langle \overline{\theta'}(\tau_1), \theta' \circ \theta \rangle$, where $\langle \tau_1, \theta_1 \rangle = \mathcal{V}(E, \Gamma), \theta_2 = U(\tau_1, \mathbf{prop})$,
$[\Gamma_1, \ldots, \Gamma_n] = map\ \ \Delta\ \ [\pi_1, \ldots, \pi_n]$, $\langle [\tau_1, \ldots, \tau_n], \theta \rangle = \widehat{\mathcal{V}}([E_1, \ldots, E_n], [\Gamma_1 \oplus \Gamma, \ldots, \Gamma_n \oplus \Gamma], \theta_2 \circ \theta_1)$,
$\theta' = U^*([\overline{\theta}(\tau_1), \ldots, \overline{\theta}(\tau_{n-1})], [\overline{\theta}(\tau_2), \ldots, \overline{\theta}(\tau_n)])$

and

```
$\widehat{\mathcal{V}}(L, \overrightarrow{\Gamma}, \theta) = let\ \ h([], [], \theta, T) = \langle \theta, \overleftarrow{T} \rangle$
                      $h(F{::}L, \Gamma{::}L', \theta, T) = let\ \ \langle \tau, \theta' \rangle = V(F, \overline{\theta}(\Gamma))$
                                                 $in$
                                                   $h(L, L', \theta' \circ \theta, \tau{::}\,T)$
                  $in$
                    $h(L, \overrightarrow{\Gamma}, \theta, [])$
```

Figure 1.14: A Milner-style type-inference algorithm for $\mathcal{NDL}_0^\omega$.

type

$$[(\forall\, \alpha)\, \alpha \to \alpha] \to [(\forall\, \alpha)\, \alpha \to \alpha]$$

(with respect to the empty context) as follows:

1. $[I : (\forall\, \alpha)\, \alpha \to \alpha] \vdash I : (\forall\, \alpha)\, \alpha \to \alpha$ $\quad$ $[T_7]$
2. $[I : (\forall\, \alpha)\, \alpha \to \alpha] \vdash I : [(\forall\, \alpha)\, \alpha \to \alpha] \to [(\forall\, \alpha)\, \alpha \to \alpha]$ $\quad$ 1, $[T_{14}]$
3. $[I : (\forall\, \alpha)\, \alpha \to \alpha] \vdash I\ I : (\forall\, \alpha)\, \alpha \to \alpha$ $\quad$ 2, 1, $[T_9]$
4. $[] \vdash \lambda\, I\, .\, I\ I : [(\forall\, \alpha)\, \alpha \to \alpha] \to [(\forall\, \alpha)\, \alpha \to \alpha]$ $\quad$ 3, $[T_9]$

Our type type inference *algorithm*, however, like Milner's original algorithm [24], will only able to infer shallow types. For instance, the algorithm will fail on the above self-application. The problem is that there is no effective (algorithmic) way of deducing non-shallow types. Accordingly, just like Milner's procedure, our algorithm will be incomplete with respect to the type system of Figure 1.13: even though certain judgments are provable in the system, the algorithm will not derive them. Alternatively, we could syntactically restrict our set of types by weeding out non-shallow types and then slightly reformulating the rules so as to make the algorithm complete with respect to the inference system, which is the approach of Milner and Damas [14]. Our approach here is similar to Cardelli's [9].

The algorithm $\mathcal{W}$ appears in Figure 1.14 in conventional ML-like notation. It uses an auxiliary function $\mathcal{V}$ that takes a phrase F and a context Γ and returns a pair (τ, θ) consisting of a type τ (the most general type of F w.r.t. Γ), and a substitution θ that is used to update our current "guesses" about the types of the bound variables of F. The definition of $\mathcal{V}(F, \Gamma)$ is given by pattern matching on F. We omit the cases when F is a constant, as those are trivial. Although we will not prove this here, we claim that if $\mathcal{W}(F, \Gamma) = \tau$ then $\Gamma \vdash F : \tau$. Moreover, $\mathcal{W}(e, A)$ is a *principal* type, i.e., more general that any other type that can be derived for F with respect to Γ. Formally: for all τ, if $\Gamma \vdash F : \tau$ then τ is an *instance* of $\mathcal{W}(F, \Gamma)$. Principality follows from the fact that unification is used to "guess" the types of bound $\mathcal{NDL}_0^\omega$ identifiers, and unification always returns the most general possible unifying substitution.

Note the difference in the handling of **let**-bound and **dlet**-bound identifiers vs. λ-bound and ϕ-bound identifiers. When we encounter a function $\lambda\, I_1, \ldots, I_n \,.\, E$ or method $\phi\, I_1, \ldots, I_n \,.\, D$, we augment the current type context with n assignments $I_1 : \alpha_1, \ldots, I_n : \alpha_n$, for fresh α_j, and move on with the body E or D. Once a type variable α_j becomes instantiated to some type τ_j within the body, it remains an instance of τ_j throughout. This means that the type of I_j cannot be freshly instantiated as needed at different places within the body E (or D): in Hindley-Milner parlance, I_j is a *non-generic* identifier. Every occurence of I_j within the body must have the same (principal) type. That is why the algorithm is unable to infer non-shallow types. By contrast, for an expression such as **let** $I = F$ **in** E or a deduction **dlet** $I = F$ **in** D, we can process the phrase F first, obtain a type judgment $I : \tau$, then *generalize* τ over its free variables $\alpha_1, \ldots, \alpha_k$ to get a judgment $I : (\forall\, \alpha_1) \cdots (\forall\, \alpha_k)\, \tau$, and then freely specialize this generic type as needed within the body E (or D). That is why we say that the free occurrences of I within the bodies E and D are *generic*. Note that $\mathcal{W}$ performs specialization in the clause for identifers, which is

$$\mathcal{V}(I, \Gamma) = \textit{if}\ \ \Gamma(I) = (\forall\, \alpha_1) \cdots (\forall\, \alpha_k)\, \tau \ \ \textit{then}\ \ \langle \overline{\{\alpha_1 \mapsto \alpha_1', \ldots, \alpha_k \mapsto \alpha_k'\}}(\tau), \{\} \rangle \ \ \textit{else}\ \ \textit{error}(),$$
$$\text{where } \alpha_1', \ldots, \alpha_k' \text{ are fresh, } k \geq 0.$$

Hence, if I is generic, i.e., if it was previously bound by a **let** or **dlet**, then its type in the current context is of the form $(\forall\, \alpha_1) \cdots (\forall\, \alpha_k)\, \tau$, and the above clause will return a copy of τ with $\alpha_1, \ldots, \alpha_k$ replaced by fresh type variables $\alpha_1', \ldots, \alpha_k'$.

1.13 Type-ω DPLs as programming languages

In Section 1.9 we saw that there is an isomorphism between $\mathcal{NDL}_0$ and the set of pure $\mathcal{NDL}_0^\omega$ deductions $\mathfrak{PD}$, which meant that by restricting ourselves within $\mathfrak{PD}$ we can use $\mathcal{NDL}_0^\omega$ as a type-α DPL. In this section we will single out a set of "pure expressions" $\mathfrak{PE} \subseteq \mathfrak{E}$ and show that there is an isomorphism between $\mathfrak{PE}$ and the regular λ-calculus. This will entail that by syntactically restricting ourselves within $\mathfrak{PE}$ we can use $\mathcal{NDL}_0^\omega$ as a conventional programming language.

Our discussion here will be concerned only with the kernel of $\mathcal{NDL}_0^\omega$, but the ideas readily extend to the full language. We will say that a $\mathcal{NDL}_0^\omega$ expression E is *pure* iff it contains no deductions D. That is, viewing E as a parse tree, no subtree of it is a deduction. We write $\mathfrak{PE}$ to denote the set of all pure expressions of core $\mathcal{NDL}_0$. The intuition that pure expressions E look and behave just like regular λ-calculus expressions can be made precise by defining a regular λ-calculus $\mathcal{L}$ and then establishing an isomorphism between $\mathcal{L}$ and $\mathfrak{PE}$. Accordingly, let us define a language $\mathcal{L}$ of expressions E as follows:

$$E \ ::= \ c \mid I \mid \lambda\, I_1, \ldots, I_n \,.\, E \mid E\ \ E_1 \cdots E_n \qquad (1.30)$$

As constants c we take the propositional atoms, **true**, **false**, the five propositional constructors, and the propositional equality function $\equiv$. So we have here the same constants that we had in $\mathcal{NDL}_0^\omega$ except for the primitive methods. We define propositions P as before and the set of values V as

$$V \ ::= \ c \mid P \mid \lambda\, I^* \,.\, E.$$

Free and bound identifier occurrences are defined as usual, expressions are viewed as identical modulo alphabetic conversion, and the substitution operation $E[E_1, \ldots, E_n/I_1, \ldots, I_n]$ is introduced in the customary manner. A formal substitution-based call-by-value semantics can be given by three rules that establish judgments of the form $\vdash E \hookrightarrow V$:

$$\frac{}{\vdash V \hookrightarrow V}\ [\mathcal{L}_1] \qquad \frac{\vdash E \hookrightarrow c \quad \vdash E_i \hookrightarrow V_i \quad \vdash c\ V_1 \cdots V_n \hookrightarrow V}{\vdash E\ \ E_1 \cdots E_n \hookrightarrow V}\ [\mathcal{L}_2]$$

$$\frac{\vdash E \hookrightarrow \lambda\, I_1, \ldots, I_n \,.\, E' \quad \vdash E_i \hookrightarrow V_i \quad \vdash E'[V_i/I_i] \hookrightarrow V}{\vdash E\ \ E_1 \cdots E_n \hookrightarrow V}\ [\mathcal{L}_3]$$

along with two "δ-axioms" for the equality function:

$$\frac{}{\vdash P \equiv P \hookrightarrow \textbf{true}}\ [\mathcal{L}_4] \qquad \frac{}{\begin{array}{c}\vdash P \equiv Q \hookrightarrow \textbf{false}\\ \text{when } P \neq Q\end{array}}\ [\mathcal{L}_5]$$

Thus $\mathcal{L}$ is a conventional applied λ-calculus [23]: it consists of the usual λ-calculus core (variable references, abstractions, and applications), augmented with constants that represent certain objects and computable operations on a particular domain of interest—in this case, the set of propositions of classical zero-order logic.

Note that $\mathcal{L}$ is embedded intact within $\mathcal{NDL}_0^\omega$. Accordingly, the isomorphism mapping between $\mathcal{L}$ and $\mathfrak{PE}$ is the identity function, since every expression of $\mathcal{L}$ is a pure expression of $\mathcal{NDL}_0^\omega$ and every pure expression of $\mathcal{NDL}_0^\omega$ is an expression of $\mathcal{L}$. This gives a trivial syntactic bijection between $\mathcal{L}$ and $\mathfrak{PE}$. The next result makes this bijection into an isomorphism by showing that the similarity extends to the semantics: every expression in the intersection of $\mathcal{L}$ and $\mathcal{NDL}_0^\omega$ has the exact same meaning in both languages.

Theorem 1.16 (Pure expression isomorphism) $\vdash E \hookrightarrow V$ *iff* $(\forall\, \beta)\, [\beta \vdash E \rightsquigarrow V]$.

Proof: A straightforward induction on E. ■

The fact that this equivalence holds for *all* β captures the intuition that the behavior of pure $\mathcal{NDL}_0^\omega$ expressions is completely independent of the assumption base. In particular, we can choose $\beta = \emptyset$, which means that pure expressions may well be evaluated in the empty assumption base.

These ideas extend to the full $\mathcal{NDL}_0^\omega$ language. Every pure expression of $\overline{\mathcal{NDL}_0^\omega}$ looks and behaves like a regular "program" in a higher-order call-by-value lexically-scoped language. This could readily be made precise by extending the λ-calculus $\mathcal{L}$ given by 1.30 to a "full" λ-calculus $\overline{\mathcal{L}}$ as follows:

$$E \ ::= \ \cdots \mid \textbf{fix}\, I \,.\, E \mid \textbf{let}\ I = E_1\ \textbf{in}\ E_2 \mid \textbf{begin}\ E_1 \cdots E_n\ \textbf{end} \mid \textbf{match}\ \ E\ \ (\pi_1?\ E_1) \cdots (\pi_n?\ E_n)$$

Rules $[\mathcal{L}_1]$—$[\mathcal{L}_3]$ would then be augmented with rules that give the semantics of these new syntax forms in the usual manner, and Theorem 1.16 would be extended to assert an isomorphism between $\overline{\mathcal{L}}$ and the pure expressions of $\overline{\mathcal{NDL}_0^\omega}$, the relevant bijection again being simply the identity function.

The practical import of this containment is that a type-ω DPL could be used as a regular programming language. In the case of $\mathcal{NDL}_0^\omega$, we can write arbitrary functions to operate on propositions, e.g., functions to compute normal forms for propositions (such as CNF or DNF), functions to implement procedures such as resolution, or the Davis-Putnam algorithm, or semantic tableaux, and so on. But in fact there is no *a priori* reason why the computational part of a type-ω DPL should be limited to propositions. One could easily introduce additional primitive values (represented as constants) for integers, reals, strings, lists, and other scalar and compound data types. The set of pure expressions will then constitute a powerful higher-order functional language in the tradition of Scheme or ML (depending mainly on whether or not a static type discipline is enforced; see Section 1.12). In fact data types such as integers and lists are very useful for writing powerful and flexible methods and so any sophisticated type-ω DPL is likely to offer them. Accordingly, such a DPL would constitute a general-purpose programming language. As an example, Athena [1] is a type-ω DPL whose computational part comprises a rich programming language featuring strings, numbers, lists, input/output facilities, side effects, and so on.

In closing, we draw attention to how *orthogonally* type-ω DPLs integrate computation and deduction. One could be using a type-ω DPL as a regular programming language for months on end without even being aware of the deductive aspects of the language, because those aspects are non-intrusive in every respect:

- *Syntactically*, no special keywords or syntax forms are required if one simply wants to write programs. Programmers can write code in type-ω DPLs just as they would write it in a conventional modern language such as Scheme or ML.

- *Semantically*, type-ω programs (pure expressions) have the exact same meaning that one would expect them to have in a conventional modern programming language based on the call-by-value λ-calculus, as made precise by Theorem 1.16. The presence of assumption bases has no bearing on the meaning of such expressions.

- *Pragmatically*, type-ω programs incur zero static and zero run-time penalty, because one only "pays" for the deductive machinery if one uses it. Pure expressions contain no deductions and hence they never touch the assumption base—an intuition formally captured in Theorem 1.16 by the universal quantification over all assumption bases.

Conversely, of course, one could be using a type-ω DPL as a type-α DPL by remaining within the set of pure deductions, without even being aware that a full-blown higher-order programming language is available alongside the deductive framework.

1.14 Related work

We have seen that a type-ω DPL $\mathcal{L}$ can be used both for proof presentation, when we simply wish to express a proof in a form that can be checked efficiently; and for proof search, when we want to write a theorem prover with a strong soundness guarantee. In the first case, the type-α subset of $\mathcal{L}$ is used; in the second case, the full range of type-ω features of $\mathcal{L}$ can be used. We will first discuss related work on proof presentation and checking, and then related work on proof search.

There are many systems for proof presentation and checking. Prominent among them are systems deriving from Automath [8], such as the Calculus of Constructions [12, 11], Nuprl [10], and LF [18]. These systems are largely based on and inspired by the Curry-Howard isomorphism [19] and make heavy use of fairly sophisticated type theories. A proof is represented by a term of the typed λ-calculus, using a sufficiently rich type theory to guarantee that the proof is sound iff the term that represents it is well-typed. If type checking is decidable in the theory at hand, then this reduction means that proofs can be checked mechanically.

Proofs in such systems are typically annotated with a large amount of type information, which must be explicitly given in order to make sure that type checking is decidable. This increases the size of the proofs and makes them harder to read and write. In addition, assumption scope in Curry-Howard systems is captured by the lexical scope of λ-bound variables. By contrast, in a type-ω DPL such as $\mathcal{NDL}_0^\omega$ there are two distinct notions of scope, the lexical scope of identifiers and the assumption scope of propositions. We believe that teasing apart these two notions of scope results in much cleaner deductions. For additional comments on the difference between DPLs and Curry-Howard systems, refer to Section 1.4 of the paper "Denotational Proof Languages" [5]. A deeper comparison of DPLs and LF, in particular, can be found in Chapter 3 of Arkoudas's dissertation [4].

On the proof discovery front, the systems that are closest to type-ω DPLs are those derived from LCF [16], such as HOL [17]. They are the only systems we are aware of that make strong soundness guarantees on the basis of rigorously formulated semantics. They are substantially different than DPLs, however. HOL is a programming language (ML) augmented with some abstract data types that model sequents and various built-in functions for constructing valid sequents. There is nothing in the *core semantics* of HOL that pertains to proofs. This essentially means that everything has to be done via regular functional abstraction and application, with a lot of top-level helper functions and a strong type system mounted on top to guarantee soundness. While this is possible, it has several drawbacks. Most notably, the burden of hypothesis management falls on the users, who must explicitly keep track of their assumptions and intermediate conclusions by manipulating sequents.

The difference between assumption bases and sequents is fundamental. It is akin to the difference between recursion versus explicit manipulation of the control stack. In a language that allows recursion, the user manipulates the control stack *implicitly*. The tedium of pushing and popping stack frames is relegated to the language implementation. The benefits are well known: programs are shorter, easier to read and write, and expressed at a higher level of abstraction. Likewise, in DPLs the user manipulates the assumption base implicitly. The tedium of assumption manipulation (e.g., discharging hypotheses) is relegated to the DPL semantics.

Furthermore, it is not the case that the abstraction capabilities and rich type system of a language such as ML can magically "hide" the onus of sequent manipulation. To see this, it is useful to draw another parallel, this time between assumption bases and stores. While in principle everything can be done with regular functional abstraction and application, perhaps in tandem with a type system, some applications—e.g., those dealing with bank accounts—have such a strong state component that they can be expressed much more naturally in an imperative programming language that provides stores and mutation. Of course we could, in principle, develop the application in a purely functional language; we could represent the store using state lists and simulate its manipulation by explicitly passing those lists around at run time. However, this would be inordinately cumbersome. No matter how wonderfully abstract the functional language might be in other respects, in some cases it is simply much easier to use an imperative programming language that posits the store as a fundamental abstraction and shifts the burden of its management to the formal semantics. In the case of proofs, the role of the store is played by the assumption base. HOL represents assumption bases by lists (sequents) and explicitly passes those around at run time (e.g., every primitive HOL inference rule operates on sequents), much as a purely functional language would simulate stores by passing around state lists dynamically. While this works, it is inordinately cumbersome. It is much easier to use a language that posits the assumption base as a fundamental abstraction and shifts the burden of its management from the user to the formal semantics. That is exactly what DPLs do, and we believe this to be the chief reason why proofs and proof methods are easier to express in such languages. A more detailed technical comparison of HOL and a specific type-ω DPL, Athena, can be found in another paper [2].

1.15 Concluding remarks

Beginning users of type-ω DPLs occasionally protest that a proof in such a language is "not really a proof" but rather a proof recipe—an algorithm for constructing a proof, which, like any algorithm, might or might not succeed. This reflects a misunderstanding of the fundamental distinction between expressions (computation) and deductions (inference). What such users mean by "a proof recipe", of course, is a method. And they are quite correct in that a method is not by itself a proof of anything. This is made explicit in type-ω DPLs owing to the fact that methods are classified as expressions, not as deductions. It is the *application* of a method that constitutes a deduction. To put it in simple syntactic terms: a proof in a type-ω DPL is "a D": a parse tree generated by the abstract grammar for deductions. The evaluation of any given D in any assumption base β will always result in some specific proposition P, if it results in anything at all, which is logically *derived* from β. And we argue that this is sufficient justification for regarding D as a deduction—a deduction of the conclusion P from the premises in β.

A more interesting issue is raised by an appeal to the traditional view of a proof as an argument whose validity can be checked *efficiently*. One might argue that whatever a proof is, it ought to be something that we must be able to check promptly. Determining whether an argument successfully derives its professed conclusion should be an expeditious process—at any rate, a process that is at least guaranteed to terminate.

There are two main motivations for this viewpoint, one theoretical and the other practical. The theoretical motivation is that if proof checking terminates then, assuming a countable set of proofs, we can mechanically enumerate all theorems that can be derived from a given set of axioms β: we go through each proof D, check it, and if we find that D derives a conclusion P from β then we append P to our list of theorems, otherwise we continue with the next proof. Using standard Gödel numbering schemes, this means that we can identify first-order theories—sets of theorems—with recursively enumerable sets of integers, and this has pleasant consequences. For instance, many concepts and results from the theory of recursively enumerable sets, such as productive sets, creative sets, the positive solution to Post's problem, etc., carry over immediately to the subject of first-order theories in mathematical logic.

But it is easy to see that the recursively enumerability of theorems is preserved even when proofs can diverge. Specifically, let β be a recursive set of axioms, let D be a proof, and let $Yields_\beta$ be a binary relation from proofs to propositions such that $Yields_\beta(D, P)$ holds iff D deduces P from β. Then the set of theorems of β can be defined as

$$\mathfrak{T}_\beta = \{P \mid (\exists\, D)\ Yields_\beta(D, P)\}.$$

When $Yields_\beta$ is decidable, the above set is in Σ_1, i.e., it is recursively enumerable. In the case of DPLs, $Yields_\beta(D, P)$ holds iff $\beta \vdash D \rightsquigarrow P$, which is not decidable in the type-ω case. Nevertheless, $\mathfrak{T}_\beta$ remains recursively enumerable in the type-ω case because the relation $\beta \vdash D \rightsquigarrow P$ is still *semi*-decidable. We need only note that $\mathfrak{T}_\beta$ can be expressed via an additional existential projection as

$$\mathfrak{T}_\beta = \{P \mid (\exists\, D)\,(\exists\, n)\ Yields\text{-}Step_\beta(D, P, n)\}$$

or, in the DPL case,

$$\mathfrak{T}_\beta = \{P \mid (\exists\, D)\,(\exists\, n)\,\beta \vdash_n D \rightsquigarrow P\} \tag{1.31}$$

where the predicate $Yields\text{-}Step_\beta(D, P, n)$ (or, in the DPL case, $\beta \vdash_n D \rightsquigarrow P$), holds iff D derives P from β in *at most n steps.* Since $\beta \vdash_n D \rightsquigarrow P$ *is* decidable for type-ω proofs, we conclude from 1.31 that the set of theorems $\mathfrak{T}_\beta$ is recursively enumerable. (To see this algorithmically, note that we can enumerate all conclusions of an infinite list of type-ω proofs $D_1, D_2, D_3, \ldots$ by *dovetailing*: we perform one step from D_1, then two steps from D_1 and one from D_2, then three steps from D_1, two from D_2 and one from D_3, and so forth.)

The practical motivation for insisting on terminating proofs appears to stem from the social view of proofs as arguments adduced in a debate: to settle such matters conclusively, we must be able to tell in a finite time period whether or not a given argument is valid. Here we simply point out that no rational agent S would ever adduce a proof D in order to convince others that a conclusion P follows from some premises β unless S is already certain that the judgment $\beta \vdash D \leadsto P$ holds. Presumably, S has already privately checked D and has ascertained that it derives P from β. In fact S might have well counted the number of steps it took to check D, call it n, and can confidently allow others to halt their efforts after n steps.

We can gain some additional insight into this issue by drawing an analogy with the formalization of algorithms as Turing machines. Prior to the 20^{th} century an algorithm was customarily viewed as a procedure that always terminates with a well-defined result—what is the use, after all, of a recipe that never produces anything? But the diagonalization arguments which showed that no class of terminating machines could ever constitute a thorough formalization of the concept of algorithm led researchers to adopt diverging computations. So why are Turing machines viewed as a good formalization of the concept of algorithm even though they can diverge? The answer does not lie merely in the pragmatic desire to avert the diagonalization obstacle. Rather, it is mainly because a Turing machine embodies all the essential characteristics that we informally attribute to algorithms: it admits a finite description, it operates in a discrete stepwise fashion, and so on (see properties 1—5 in the list given by Rogers in Section 1.1 of his "Theory of recursive functions and effective computability" [29].) Termination is not an *essential* property of algorithms. If it were, then no formalization that allowed infinite computations could ever claim to capture the concept successfully.

Likewise, we submit that termination is not an essential characteristic of proofs. We see no *qualitative* difference between type-α deductions, which adhere to the conventional view of proofs and can be checked efficiently, and $\mathcal{NDL}_0^\omega$ deductions, which may take indefinitely long to yield a conclusion. Both are *logical demonstrations*: starting from certain assumption bases, they both proceed in a stepwise manner, deducing various intermediate propositions by sound manipulation of the assumption base, until the ultimate conclusion is finally obtained or some logical oversight is uncovered. The only difference is quantitative: where a $\mathcal{NDL}_0$ proof will promptly reach a verdict (either the desired conclusion or an error), a $\mathcal{NDL}_0^\omega$ proof might take much longer—indeed, in the extreme case it might take forever. But we stress that this does not constitute a difference in essence, at least no more than it does in the case of algorithms.

One could also perhaps argue that type-α proofs are *focused* because they have zero computational cost, whereas unrestricted type-ω proofs might wander around performing search, and that this lack of focus constitutes a qualitative difference. But we note that type-α proofs can also be terribly unfocused by way of detours and superfluous intermediate conclusions—consider a billion-line type-α proof that derives the proposition **true**. Focus is no more an essential feature of proofs than efficiency is of algorithms.

Of course, for engineering purposes, we may be perfectly justified in restricting attention to classes of proofs that have complexity properties appropriate for the application at hand. If we download a proof-carrying [25] executable and need to check the proof to make sure that the code does not corrupt any of our system's resources, we want the proof to be validated in no more than a few seconds. In that case it may well be reasonable to accept only type-α or type-β proofs. On the other hand, if someone hands us an alleged proof of Goldbach's conjecture we are probably willing to let a computer check that proof over a period of weeks, or even months if need be. But in any event we should resist the temptation to draw ontological conclusions about the nature of deduction on the basis of our own practical needs and limitations. The development of the theory of algorithms teaches us otherwise.

1.16 Implementation

In this section we present a complete implementation of the full $\mathcal{NDL}_0^\omega$ language in fewer than 400 lines of SML code. The implementation consists of three SML structures, `AbstractSyntax`, `Semantics`, and `AssumptionBase`. We do not include a parser here because we want to stress that the choice of concrete syntax is largely arbitrary and independent of the abstract syntax and semantics; it is an issue that should be sharply decoupled in the language design process.[13] Perhaps the simplest approach is to use a concrete syntax based s-expressions, but infix variants are also possible and we encourage the reader to experiment with different alternatives.[14]

The structure `AbstractSyntax` models the abstract syntax of propositions, deductions, expressions, and phrases. It is shown in Figure 1.15. The structure `AssumptionBase` models assumption bases and is defined thus:

```
structure AssumptionBase =
struct
val empty_ab = [];

fun member(P:AbstractSyntax.Prop,[]) = false
  | member(P,Q::rest) = if P = Q then true else member(P,rest);

fun members([],_) = true
  | members(P::rest,ab) = member(P,ab) andalso members(rest,ab);

val insert = op::;
val augment = op@;
end;
```

Thus assumption bases are modelled by lists that grow and shrink on the left side, in a stack-like fashion. This is clearly a naive implementation. More efficient implementations would result by representing assumption bases as tries, for example. It is interesting to note, however, that implementing assumption bases as simple lists is nowhere as bad as one would think at first. For instance, Athena (a type-ω DPL for polymorphic multi-sorted first-order logic) uses a heavily optimized implementation of assumption bases based on hash tables and balanced search trees. Yet when we compared the performance of that implementation with the performance of a naive list implementation we only found a substantial difference in cases involving very large assumption bases (containing thousands of propositions). In more common cases there was little difference, and in fact in some cases the naive implementation was slightly faster. A little reflection will reveal the reason for this. A typical deduction proceeds by inferring a lemma P, putting it in the assumption base, then another lemma Q, putting that in the assumption base too, and then applying some primitive method that requires P and Q, so P and Q will be looked up more or less immediately after they are put on the stack, while they are still at the top. Moreover, they will rarely ever be used again afterwards. Accordingly,

[13] David MacQueen [20] gives the following advice to language designers about concrete syntax: do it last. Getting bogged down in premature debates and dilemmas about concrete syntax can greatly hinder the design of a language. If the abstract syntax and the semantics are clean, the language will be readable and writable with pretty much any sensible concrete syntax. But if the designer does not get the abstract syntax and the formal semantics right, the best concrete syntax in the world will not save the language.

[14] Nevertheless, for the sake of offering a complete working system, the online copy of this implementation, at `www.ai.mit.edu/projects/dynlangs/dpls/omega`, includes a lexer and a parser (for an s-expression concrete syntax). A source file for the SML-NJ compilation manager can also be found there.

we usually do not have to search the stack at any great depth, and thus very few comparisons are actually made. So the worst case where the entire list must be searched will rarely occur because of the way deductions are written: there is usually a very small gap between the derivation of a lemma and its first—and often last—use.

The structure `Semantics` is the gist of the system. Lexical environments (implemented as higher-order functions for simplicity) are used instead of substitutions. The interpreter `evPhrase` takes a phrase `phr`, an environment reference `env`, and an assumption base `ab` and evaluates `phr` in `env` and `ab`, ultimately producing an element of the datatype `value`, or else diverging or raising an exception `EvalError`. Environment references are used in order to implement recursion. In particular, functions and methods are represented as lexical closures that contain pointers to the corresponding environments. The implementation of `fix` puts this to use by resetting the environment pointer appropriately, thereby "tying the knot" in the usual manner (see Chapter 8 of "Functional Programming Languages" [28] and Chapter 5 of "Essentials of Programming Languages " [15] for a discussion of this technique). The relevant data types and the pattern matcher are shown in Figure 1.16. The expression interpreter appears in Figure 1.17, and the deduction interpreter in Figure 1.18.

Environment-based implementations of type-ω DPLs such as the one given here highlight the differences between assumption bases and lexical environments. The most important difference is that methods and functions are statically closed over environments but *dynamically* closed over assumption bases. In particular:

- The assumption base in which we evaluate a method is thrown away. By contrast, a pointer to the lexical environment in which we evaluate the method is retained in the method's closure.

- Symmetrically, the application of a method takes place in the environment that was stored in the method's closure, on one hand, and in the *current* assumption base on the other, i.e., the assumption base in which the application occurs.

In a sense, assumption bases are a cross between lexical environments and stores. Like environments and unlike stores, they grow and shrink in a context-free fashion and are free of side effects. But like stores and unlike environments, they are dynamic rather than static. A denotational-style formal semantics would clarify these differences rigorously, and would also serve to emphasize that the meaning of a deduction is a function over assumption bases.[15]

[15]Indeed, the reason why these differences are brought out in interpreters such as the one presented here is because these interpreters are essentially implementations of denotational semantics.

```
structure AbstractSyntax = struct

datatype prim_fun = eqFun | notFun | andFun | orFun | ifFun | iffFun;

datatype prim_method = claim | dn | mp | both | leftAnd | rightAnd | cd | leftEither
                     | rightEither | equiv | leftIff | rightIff | absurd | trueIntro;

datatype constant = primMethodConst of prim_method
                  | primFunConst of prim_fun
                  | propAtomConst of string
                  | trueConst
                  | falseConst;

type ide = string;

datatype prop =  atom of string
               | trueProp
               | falseProp
               | neg of prop
               | conj of prop * prop
               | disj of prop * prop
               | cond of prop * prop
               | biCond of prop * prop;

datatype exp = constExp of constant
             | idExp of ide
             | funExp of {params: ide list, body:exp}
             | methodExp of {params: ide list, body:ded}
             | funAppExp of exp * phrase list
             | letExp of ((ide * phrase) list) * exp
             | fixExp of ide * exp
             | matchExp of {discriminant: phrase, cases: (pat * exp) list}
             | seqExp of exp list
     and
         ded = methodAppDed of exp * (phrase list)
             | assumeDed of exp * ded
             | supAbDed of exp * ded
             | letDed of ((ide * phrase) list) * ded
             | byDed of exp * ded
             | seqDed of ded list
             | matchDed of {discriminant: phrase, cases: (pat * ded) list}
     and
         pat = idPat of ide
             | atomPat of string
             | truePat
             | falsePat
             | anyPat
             | negPat of pat
             | conjPat of pat * pat
             | disjPat of pat * pat
             | condPat of pat * pat
             | biCondPat of pat * pat
   and  phrase = expression of exp | deduction of ded;

end;
```

Figure 1.15: The abstract syntax structure.

```
structure Semant = struct

structure A = AbstractSyntax;
structure AB = AssumptionBase;

datatype value = propVal of A.prop
               | primMethodVal of A.prim_method
               | primFunVal of A.prim_fun
               | funClosVal of {params:A.ide list,body:A.exp,env:environment ref}
               | methClosVal of {params:A.ide list,body:A.ded,env:environment ref}
and
   env_binding = unBound | bind of value

withtype environment = A.ide -> env_binding;

exception EvalError of string;
fun evError(str) = raise EvalError("\n"^str^"\n");

val empty_env = fn id => unBound;

fun extend(f,[]) = f
  | extend(f,(a,b)::rest) = extend(fn x => if x = a then b else f(x),rest);

fun augmentEnv(env1,env2) = fn id => (case env2(id) of unBound => env1(id) | b => b);

fun failMatch() = evError("Failed match.");

fun match(P,A.idPat(id),env) =
            (case env(id) of
               unBound => extend(env,[(id,bind(propVal(P)))])
             | bind(propVal(Q)) => if P = Q then env else failMatch()
             | _ => failMatch())
  | match(_,A.anyPat,env) = env
  | match(A.trueProp,A.truePat,env) = env
  | match((A.falseProp,A.falsePat,env) = env
  | match(A.atom(s1),A.atomPat(s2),env) = if s1 = s2 then env else failMatch()
  | match(A.neg(P),A.negPat(pat),env) = match(P,pat,env)
  | match(A.conj(P1,P2),A.conjPat(pat1,pat2),env) = thread(P1,pat1,P2,pat2,env)
  | match(A.disj(P1,P2),A.disjPat(pat1,pat2),env) = thread(P1,pat1,P2,pat2,env)
  | match(A.cond(P1,P2),A.condPat(pat1,pat2),env) = thread(P1,pat1,P2,pat2,env)
  | match(A.biCond(P1,P2),A.biCondPat(pat1,pat2),env) = thread(P1,pat1,P2,pat2,env)
  | match(_) = failMatch()
and thread(P1,pat1,P2,pat2,env) = match(P2,pat2,match(P1,pat1,env));

fun findMatch(P,cases) =
      let fun tryCases([]) = failMatch()
            | tryCases((pattern,phrase)::rest) =
                (phrase,match(P,pattern,empty_env)) handle _ => tryCases(rest)
      in
         tryCases(cases)
      end;
```

Figure 1.16: The semantics structure, part 1.

```
fun withProp(propVal(P),f) => f(P)
  | withProp(_) = evError("Wrong kind of value; a proposition was expected here.");

fun evalMethArgs(args,env,ab,eval) =
 let fun f([],vals,lemmas) = (rev(vals),lemmas)
       | f(arg as A.expression(_)::rest,vals,lemmas) =
           f(rest,eval(arg,env,ab)::vals,lemmmas)
       | f(arg as A.deduction(_)::rest,vals,lemmas) =
           withProp(eval(arg,env,ab),fn P => f(rest,propVal(P)::vals,P::lemmas))
 in
    f(args,[],[])
 end;

fun evExp(A.constExp(A.primMethodConst(m)),_,_) = primMethodVal(m)
  | evExp(A.constExp(A.primFunConst(f)),_,_) = primFunVal(f)
  | evExp(A.constExp(A.trueConst),_,_) = propVal(A.trueProp)
  | evExp(A.constExp(A.falseConst),_,_) = propVal(A.falseProp)
  | evExp(A.constExp(A.propAtomConst(A)),_,_) = propVal(A.atom(A))
  | evExp(A.idExp(id),env,_) =
      (case (!env id) of unBound => evError("Unbound identifier: "^id)
                       | bind(v) => v)
  | evExp(A.funExp({p,b}),env,_) = funClosVal({params=p,body=b,env=env})
  | evExp(A.methodExp({p,b}),env,_) = methClosVal({params=p,body=b,env=env})
  | evExp(A.funAppExp(fexp,args),env,ab) =
     let val (fval,vals) = (evExp(fexp,env,ab),map (fn p => evPhrase(p,env,ab)) args)
     in
       (case fval of
          primFunVal(f) => applyPrimFun(f,arg_vals,ab)
        | funClosVal({params,body,clos_env}) =>
           evExp(body,ref(extend(!clos_env,zip(params,map bind arg_vals))),ab)
        | _ => funAppError())
     end
  | evExp(A.letExp([],body),env,ab) = evExp(body,env,ab)
  | evExp(A.letExp((id,p)::rest,body),env,ab) =
     evExp(A.letExp(rest,body),ref(extend(!env,[(id,bind(evPhrase(p,env,ab)))])),ab)
  | evExp(A.fixExp(x,body),env,ab) =
     let val rec_env = ref(!env)
         val rec_value = evExp(body,rec_env,ab)
         val new_env = extend(!env,[(x,bind rec_value)])
     in
       (rec_env := new_env;rec_value)
     end
  | evExp(A.matchExp({discriminant,cases}),env,ab) =
     withProp(evPhrase(discriminant,env,ab),
              fn P => let fun f(pat,e) => (pat,A.expression(e))
                            val (env',e) = findMatch(P,map f cases)
                      in
                        evPhrase(e,ref(augmentEnv(!env,env')),ab)
                      end)
  | evExp(A.seqExp([e]),env,ab) = evExp(e,env,ab)
  | evExp(A.seqExp(e::rest),env,ab) = (evExp(e,env,ab);evExp(A.seqExp(rest),env,ab))
and
```

Figure 1.17: The semantics structure, part 2.

```
evDed(A.methodAppDed(mexp,args),env,ab) =
   let val mval = evExp(mexp,env,ab)
       val (arg_vals,lemmas) = evalMethArgs(args,ab,evPhrase)
       val new_ab = AB.augment(ab,lemmas)
   in
     (case mval of
       primMethodVal(m) => applyPrimMethod(m,arg_vals,new_ab)
     | methClosVal({params,body,env=clos_env}) =>
        evDed(body,ref(extend(!clos_env,zip(params,map bind arg_vals))),new_ab)
     | _ => methAppError())
   end
| evDed(A.assumeDed(hyp,body),env,ab) =
   withProp(evExp(hyp,env,ab),fn P => withProp(evDed(body,env,AB.insert(P,ab)),
                                            fn Q => propVal(A.cond(P,Q))))
| evDed(A.supAbDed(hyp,body),env,ab) =
   withProp(evExp(hyp,env,ab),
            fn P => withProp(evDed(body,env,AB.insert(P,ab)),
                             fn Q => if Q = A.falseProp then propVal(A.neg(P))
                                     else supposeAbsurdError()))
| evDed(A.letDed([],body),env,ab) = evDed(body,env,ab)
| evDed(A.letDed((id,A.expression(e))::rest,body),env,ab) =
    evDed(A.letDed(rest,body),ref(extend(!env,[(id,bind(evExp(e,env,ab)))])),ab)
| evDed(A.letDed((id,A.deduction(d))::rest,body),env,ab) =
    withProp(evDed(d,env,ab),
             fn P => evDed(A.letDed(rest,body),
                           ref(extend(!env,[(id,bind(propVal P))])),
                           AB.insert(P,ab)))
| evDed(A.byDed(e,d),env,ab) =
   (case (evDed(d,env,ab),evExp(e,env,ab)) of
     (v as propVal(P),propVal(Q)) => if P = Q then v else evError(by_errror))
| evDed(A.seqDed([d]),env,ab) = evDed(d,env,ab)
| evDed(A.seqDed(d::rest),env,ab) =
    withProp(evDed(d,env,ab),fn P => evDed(A.seqDed(rest),env,AB.insert(P,ab)))
| evDed(A.matchDed({discriminant,cases}),env,ab) =
    withProp(evPhrase(discriminant,env,ab),
             fn P => let fun f(pat,d) = (pat,A.deduction(d))
                         val (chosen_ded,match_env) = findMatch(P,map f cases)
                     in
                       evPhrase(chosen_ded,ref(augmentEnv(!env,match_env)),ab)
                     end
and
  evPhrase(A.expression(e),env,ab) = evExp(e,env,ab)
| evPhrase(A.deduction(d),env,ab) = evDed(d,env,ab);

end;
```

Figure 1.18: The semantics structure, part 3.

Bibliography

[1] K. Arkoudas. Athena: a formal system integrating deduction and computation. Forthcoming, at `www.ai.mit.edu/projects/express`.

[2] K. Arkoudas. A case comparison of Athena and HOL. Forthcoming MIT AI memo.

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

[4] K. Arkoudas. Denotational Proof Languages. PhD thesis, MIT, 2000, available from `http://www.ai.mit.edu/projects/dynlangs/dpls/dpl-thesis.ps`.

[5] K. Arkoudas. Type-α DPLs. MIT AI Memo 2001-25.

[6] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Background: Computational structures*, volume 2 of *Handbook of Logic in Computer Science*. Oxford Science Publications, 1992.

[7] R. S. Boyer and J. S. Moore. *A computational logic handbook*. Academic Press, New York, 1988.

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

[9] L. Cardelli. Basic polymorphic type checking. *Science of Computer Programming*, 8(2):147–172, 1987.

[10] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, EngleWood Cliffs, New Jersey, 1986.

[11] T. Coquand. Metamathetical investigations of a Calculus of Constructions. In P. Odifreddi, editor, *Logic and Computer Science*, pages 91–122. Academic Press, London, 1990.

[12] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[13] G. Cousineau and G. Huet. The caml primer. Technical report, INRIA, Rocquencourt, France, 1990.

[14] L. Damas and R. Milner. Principal type schemes for functional programs. In *Ninth Annual Symposium of Principles of Programming Languages*, pages 207–212, 1982.

[15] D. P. Friedman et al. *Essentials of Programming Languages*. McGraw-Hill, 1992.

[16] M. J. Gordon, A. J. Miller, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[17] M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a theorem proving environment for higher-order logic*. Cambridge University Press, Cambridge, England, 1993.

[18] 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.

[19] 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.

[20] David B. MacQueen. Reflections on standard ML. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693, pages 32–46. Springer Verlag, 1994.

[21] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Computer Science Series, 1974.

[22] David A. McAllester and Kostas Arkoudas. Walther recursion. In *Conference on Automated Deduction*, pages 643–657, 1996.

[23] A. R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52:87–122, 1982.

[24] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[25] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[26] 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.

[27] L. C. Paulson. *ML for the working programmer*. Cambridge University Press, Cambridge, England, 2nd edition, 1996.

[28] C. Reade. *Elements of functional programming*. Addison Wesley, International Computer Science Series, 1989.

[29] H. Rogers. *Theory of recursive functions and effective computability*. McGraw-Hill Book Company, 1967.