Abstract

Adjoint logic is a general approach to combining multiple logics with different structural properties, including linear, affine, strict, and (ordinary) intuitionistic logics, where each proposition has an intrinsic mode of truth. It has been defined in the form of a sequent calculus because the central concept of independence is most clearly understood in this form, and because it permits a proof of cut elimination following standard techniques.

In this paper we present a natural deduction formulation of adjoint logic and show how it is related to the sequent calculus. As a consequence, every provable proposition has a verification (sometimes called a long normal form). We also give a computational interpretation of adjoint logic in the form of a functional language and prove properties of computations that derive from the structure of modes, including freedom from garbage (for modes without weakening and contraction), strictness (for modes disallowing weakening), and erasure (based on a preorder between modes). Finally, we present a surprisingly subtle algorithm for type checking.

1 Introduction

A substructural logic provides fine control over the use of assumptions during reasoning. It usually does so by denying the general sequent calculus rules of contraction (which permits an antecedent to be used more than once) and weakening (which permits an antecedent not to be used). Instead, these rules become available only for antecedents of the form \(!A\). Ever since the inception of linear logic [Girard, 1987], researchers have found applications in programming languages, for example, to avoid garbage collection [Girard and Lafont, 1987], soundness of imperative update [Wadler, 1990], the chemical abstract machine [Abramsky, 1993], and session-typed communication [Caires and Pfenning, 2010, Wadler, 2012], to name just a few.

Besides linear logic, there are other substructural logics and type systems of interest. For example, affine logic denies general contraction but allows weakening and is the basis
for the type system of Alms [Tov and Pucella, 2011] (an affine functional language) and Rust (an imperative language aimed at systems programming).

If we deny weakening but accept contraction we obtain strict logic (a variant of relevance logic) where every assumption must be used at least once. On the programming language side, this corresponds to strictness, which allows optimizations in otherwise non-strict functional languages such as Haskell [Mycroft, 1980]. Interestingly, Church’s original $\lambda I$ calculus [Church, 1941] was also strict in this sense.

The question arises how we can combine such features, both in logics and in type systems. Recently, this question has been tackled through graded or quantitative type systems (see for example, Atkey [2018], Moon et al. [2021], Choudhury et al. [2021], Wood and Atkey [2022], Abel et al. [2023]). The essential idea is to track and reason explicitly about the usage of a given assumption through grades. This provides very fine-grained control and allows us to, for example, model linear, strict, and unrestricted usage of assumptions through graded modalities. In this paper, we pursue an alternative taking a proof-theoretic view with the goal of building a computational interpretation. There are three possible options that emerge from existing proof-theoretic explorations that could serve as a foundation of such a computational interpretation. The first one is by embedding. For example, we can embed (structural) intuitionistic logic in linear logic writing $!A \to B$ for $A \to B$. Similarly, we can embed affine logic in linear logic by mapping hypotheses $A$ to $A \& 1$ so they do not need to be used. The difficulties with such embeddings is that, often, they neither respect proof search properties such as focusing [Andreoli, 1992] nor do they achieve a desired computational interpretation.

A second approach is taken by subexponential linear logic [Nigam and Miller, 2009, Nigam et al., 2016, Kanovich et al., 2018] that defines multiple subexponential modalities $!^mA$, where each mode $m$ has a specific set of structural properties. As in linear logic, all inferences are carried out on linear formulas, so while it resolves some of the issues with embeddings, it still requires frequent movement into the linear layer using explicit subexponentials.

We pursue a third approach, pioneered by Benton [1994a] who symmetrically combined (structural) intuitionistic logic with (purely) linear intuitionistic logic. He employs two adjoint modalities that switch between the two layers and works out the proof-theoretic and categorical semantics. This approach has the advantage that one can natively reason and compute within the individual logics, so we preserve not only provability but the fine structure of proofs and proof reduction from each component. This has been generalized in prior work [Reed, 2009, Pruiksma et al., 2018] by incorporating from subexponential linear logic the idea to have a preorder between modes $m \geq k$ that must be compatible with the structural properties of $m$ and $k$ (explained in more detail in Section 2). This means we can now also model intuitionistic S4 [Pfenning and Davies, 2001] and lax logic [Benton et al., 1998], representing comonadic and monadic programming, respectively. We hence arrive at a unifying calculus firmly rooted in proof theory that is more general than previous graded modal type systems in that we can construct monads as well as comonads. We will briefly address dependently typed variations of the adjoint approach in Section 8.

Most substructural logics and many substructural type systems are most clearly formulated as sequent calculi. However, natural deduction has not only an important foundational role [Gentzen, 1935, Prawitz, 1965, Dummett, 1991], it also has provided a simple
and elegant notation for functional programs through the Curry-Howard correspondence [Howard, 1969]. We therefore develop a system of natural deduction for adjoint logic that, in a strong sense, corresponds to the original sequent formulation. It turns out to be surprisingly subtle because we have to manage not only the substructural properties that may be permitted or not, but also respect the preorder between modes. We show that the our calculus satisfies some expected properties like substitution and has a natural notion of verification that corresponds to proofs in long normal form, satisfying a subformula property.

In order to illustrate computational properties, we also give an abstract machine and show the consequences of the mode structure: freedom from garbage for linear modes (that is, modes admitting neither weakening nor contraction), strictness for modes that do not admit weakening, and erasure for modes that a final value may not depend on, based on the preorder of modes. We close with an algorithmic type checker for our language which, again, is surprisingly subtle.

## 2 Adjoint Sequent Calculus

We briefly review the adjoint sequent calculus from Pruiksma et al. [2018]. We start with a standard set of possibly substructural propositions, indexing each with a mode of truth, denoted by \( m, k, n, r \). Propositions are perhaps best understood by using their linear meaning as a guide, so we uniformly use the notation of linear logic. Also, for programming convenience, we generalize the usual binary and nullary disjunction (\( A \oplus B \) and 0) and conjunction (\( A \& B \) and \( \top \)) by using labeled disjunction \( \oplus\{\ell : A^\ell_m\}_{\ell \in L} \) and conjunction \( \&\{\ell : A^\ell_m\}_{\ell \in L} \). From the linear logical perspective, these are internal and external choice, respectively; from the programming perspective they are sums and products. We write \( P^m \) for atomic propositions of mode \( m \).

### Propositions

\[
A^m_m, B^m_m ::= \begin{align*}
P^m | A^m_m \rightarrow B^m_m | \&\{\ell : A^\ell_m\}_{\ell \in L} | \downarrow^m_k A^k_k & \quad \text{(negative)} \\
| A^m_m \otimes B^m_m | 1^m_m | \oplus\{\ell : A^\ell_m\}_{\ell \in L} | \uparrow^m_n A^n_n & \quad \text{(positive)}
\end{align*}
\]

### Contexts

\[
\Gamma ::= \cdot | \Gamma, x : A^m_m
\]

Each mode \( m \) comes with a set \( \sigma(m) \subseteq \{ W, C \} \) of structural properties, where \( W \) stands for weakening and \( C \) stands for contraction. We further have a preorder \( m \geq r \) that specifies that a proof of the succedent \( C_r \) may depend on an antecedent \( A_m \). This is enforced using the presupposition that in a sequent \( \Gamma \vdash C_r \), every antecedent \( A_m \) in \( \Gamma \) must satisfy \( m \geq r \), written as \( \Gamma \geq r \). We have the additional stipulation of monotonicity, namely that \( m \geq k \) implies \( \sigma(m) \supseteq \sigma(k) \). This is required for cut elimination to hold. Furthermore, we presuppose that in \( \downarrow^m_k A^k_k \) we have \( m \geq k \) and for \( \uparrow^m_n A^n_n \) we have \( n \geq m \). Also, contexts may not have any repeated variables and we will implicitly apply variable renaming to maintain this presupposition. Finally, we abbreviate \( \cdot, x : A \) as just \( x : A \).

In preparation for natural deduction, instead of explicit rules of weakening and contraction (see [Pruiksma et al., 2018] for such a system) we have a context merge operation \( \Gamma_1 ; \Gamma_2 \). Since, as usual in the sequent calculus, we read the rules bottom-up, it actually describes a nondeterministic split of the context that is pervasive in the presentations of
linear logic [Andreoli, 1992].

\[(\Gamma_1, x : A_m) ; (\Gamma_2, x : A_m) = (\Gamma_1 ; \Gamma_2, x : A_m) \text{ provided } C \in \sigma(m)\]

\[(\Gamma_1, x : A_m) ; \Gamma_2 = (\Gamma_1 ; \Gamma_2, x : A_m) \text{ provided } x \notin \text{dom}(\Gamma_2)\]

\[(\cdot) ; \Gamma_2 = \Gamma_2\]

\[(\cdot) = \Gamma_1\]

Note that the context merge is a partial operation, which prevents, for example, the use of an antecedent without contraction in both premises of the $\otimes R$ rule.

The complete set of rules can be found in Figure 1. In the rules, we write $\Gamma_W$ for a context in which weakening can be applied to every antecedent, that is, $W \in \sigma(m)$ for every antecedent $x : A_m$. Also, as is often the case in presentations of the sequent calculus, we omit explicit variable names that tag antecedents. We only discuss the rules for $\downarrow_m A_n$ because they illustrate the combined reasoning about structural properties and modes.

First, the $\downarrow R$ rule:

\[\begin{align*}
\Gamma' \geq n & \quad \Gamma' \vdash A_n \\
\Gamma_W ; \Gamma' \vdash \downarrow_m A_n & \quad \downarrow R
\end{align*}\]

Because we presuppose the conclusion is well-formed, we know $\Gamma_W ; \Gamma' \geq m$ since $\downarrow_m A_n$ has mode $m$. Again, by presupposition $n \geq m$ and we have to explicitly check that $\Gamma' \geq n$ because it doesn’t follow from knowing that $\Gamma_W ; \Gamma' \geq m$. There may be some antecedents $A_k$ in the conclusion such that $k \not\geq n$. If the mode $k$ admits weakening, we can sort them into $\Gamma_W$. If it does not, then the rule is simply not applicable.

On to the $\downarrow L$ rule:

\[\begin{align*}
\Gamma, A_n \vdash C_r & \quad \downarrow L \\
\Gamma; \downarrow_m A_n \vdash C_r
\end{align*}\]

By presupposition on the conclusion, we know $\Gamma; \downarrow_m A_n \geq r$ which means that $\Gamma \geq r$ and $m \geq r$. Since $n \geq m$ we have $n \geq r$ by transitivity, so $\Gamma, A_n \geq r$ and we do not need any explicit check. The formulation of the antecedents in the conclusion $\Gamma; \downarrow_m A_n$ means that if mode $m$ admits contraction, then the antecedent $\downarrow_m A_n$ may also occur in $\Gamma$, that is, it may be preserved by the rule. If $m$ does not admit contraction, this occurrence of $\downarrow_m A_n$ is not carried over to the premise.

This implicit sequent calculus satisfies the expected theorems, due to Reed [2009], Pruiksma et al. [2018] and, most closely reflecting the precise form of our formulation, Pruiksma [2024]. They follow standard patterns, modulated by the substructural properties and the preorder on modes.

**Theorem 1 (Admissibility of Weakening and Contraction)** The following are admissible:

\[\begin{align*}
\Gamma_W \geq m & \quad \Gamma \vdash A_m \\
\Gamma_W ; \Gamma \vdash A_m \quad \text{weaken} & \quad C \in \sigma(m) \quad \Gamma, A_m, A_m \vdash C_r \\
\Gamma, A_m \vdash C_r \quad \text{contract}
\end{align*}\]

**Theorem 2 (Admissibility of Cut and Identity)**

(i) In the system without cut, cut is admissible.
Adjoint Natural Deduction

\[\frac{\Gamma \vdash A_m}{W \vdash A_m \vdash A_m} \text{id} \quad \frac{\Gamma \geq m \geq r \quad \Gamma \vdash A_m \quad \Gamma', A_m \vdash C_r}{\Gamma ; \Gamma' \vdash C_r} \text{cut}\]

\[\frac{\Gamma \vdash A_m \vdash B_m}{\Gamma \vdash A_m \vdash B_m} \text{\text{oR}} \quad \frac{\Gamma \vdash A_m \vdash B_m}{\Gamma ; \Gamma' \vdash A_m \vdash B_m} \text{\text{oL}} \]

\[\frac{\Gamma \vdash A_m (\forall \ell \in L) \quad \& \Gamma \vdash \{\ell : A_m^\ell \}_{\ell \in L}}{\Gamma \vdash \& \{\ell : A_m^\ell \}_{\ell \in L}} \text{\&R} \quad \frac{\Gamma \vdash A_m^\ell (\ell \in L) \quad \& \Gamma \vdash \{\ell : A_m^\ell \}_{\ell \in L}}{\Gamma \vdash \& \{\ell : A_m^\ell \}_{\ell \in L}} \text{\&L} \]

\[\frac{\Gamma \vdash A_m \quad \Gamma' \vdash B_m}{\Gamma ; \Gamma' \vdash A_m \otimes B_m} \text{\otimesR} \quad \frac{\Gamma \vdash A_m \quad B_m \vdash C_r}{\Gamma \vdash A_m \otimes B_m \vdash C_r} \text{\otimesL} \]

\[\frac{\Gamma \vdash A_m^\ell (\ell \in L) \quad \oplus \Gamma \vdash \{\ell : A_m^\ell \}_{\ell \in L}}{\Gamma \vdash \oplus \{\ell : A_m^\ell \}_{\ell \in L}} \text{\oplusR} \quad \frac{\Gamma \vdash A_m^\ell (\forall \ell \in L) \quad \oplus \Gamma \vdash \{\ell : A_m^\ell \}_{\ell \in L}}{\Gamma \vdash \oplus \{\ell : A_m^\ell \}_{\ell \in L}} \text{\oplusL} \]

\[\frac{\Gamma \vdash A_k}{\Gamma \vdash \uparrow_k^m A_k} \text{\upR} \quad \frac{k \geq r \quad \Gamma \vdash A_k}{\Gamma ; \Gamma \vdash \uparrow_k^m A_k \vdash C_r} \text{\upL} \]

\[\frac{\Gamma' \geq n \quad \Gamma' \vdash A_n}{W \vdash \Gamma' \vdash \downarrow^m_n A_n} \text{\downR} \quad \frac{\Gamma \vdash A_n}{\Gamma ; \Gamma \vdash \downarrow^m_n A_n \vdash C_r} \text{\downL} \]

Figure 1: Implicit Adjoint Sequent Calculus

(ii) In the system with identity restricted to atoms \(P_m\), the general identity is admissible.

We call a proof cut-free if it does not contain cut and long if the identity is restricted to atomic propositions \(P\). It is an immediate consequence of Theorem 2 that every derivable sequent has a long cut-free proof. The subformula property of cut-free proofs directly implies that a cut-free proof of a sequent \(\Gamma_m \vdash A_m\) where all subformulas are of mode \(m\) is directly a proof in the logic captured by the mode \(m\). Moreover, an arbitrary proof can be transformed into one of this form by cut elimination. These strong conservative extension properties are a hallmark of adjoint logic.

Since our main interest lies in natural deduction, we consider only three examples.

**Example 3 (G3)** We obtain the standard sequent calculus G3 [Kleene, 1952] for intuitionistic logic.
with a single mode $U$. All side conditions are automatically satisfied since $U \geq U$.

**Example 4 (LNL and DILL)** By specializing the rules to two modes, $U$ and $L$ with the order $U > L$, we obtain a minor variant of LNL in its parsimonious presentation [Benton, 1994b]. Our notation is $FX = \downarrow^U X$ and $GA = \uparrow^U A$. Significant here is that we do not just model provability, but the exact structure of proofs except that our structural rules remain implicit.

We obtain the sequent calculus formulation of dual intuitionistic linear logic (DILL) [Barber, 1996, Chang et al., 2003] by restricting the formulas of mode $U$ so that they only contain $\uparrow^U A$. In this version we have $!A = \downarrow^U \uparrow^U A$. Again, the rules of dual intuitionistic linear logic are modeled precisely.

**Example 5 (Intuitionistic Subexponential Linear Logic)** Subexponential linear logic [Nigam and Miller, 2009, Nigam et al., 2016] also uses a preorder of modes, each of which permits specific structural rules. We obtain a formulation of intuitionistic subexponential linear logic by adding a new distinguished mode $L$ with $m \geq L$ for all given subexponential modes $m$, retaining all the other relations. We further restrict all modes $m$ except for $L$ to contain only $\uparrow^m A$, forcing all logical inferences to take place at mode $L$.

Compared to Chaudhuri [2010] our system does not contain $?A$ and is not focused; compared to Kanovich et al. [2017], our base logic is linear rather than ordered. Also, all of our structural rules are implicit.

### 3 Adjoint Natural Deduction

Substructural sequent calculi have recently found interesting computational interpretations [Caires and Pfenning, 2010, Wadler, 2012, Caires et al., 2016, Pfenning and Pruiksma, 2023, Pruiksma and Pfenning, 2022], including adjoint logic [Pruiksma and Pfenning, 2021]. In this paper, we look instead at functional interpretations, which are most closely related to natural deduction. Some guide is provided by natural deduction systems for linear logic (see, for example, Abramsky [1993], Benton et al. [1993], Troelstra [1995]), but already they are not entirely straightforward. For example, some of these calculi do not satisfy subject reduction. The interplay between modes and substructural properties creates some further complications. The closest blueprint to follow is probably Benton’s [1994b, Figure 8], but his system does not exhibit the full generality of adjoint logic and is also not quite “parsimonious” in the sense of the LNL sequent calculus.

In the interest of economy, we present the calculus with proof terms and two bidirectional typing judgments, $\Delta \vdash e \iff A_m$ (expression $e$ checks against $A_m$) and $\Delta \vdash s \implies A_m$ (expression $s$ synthesizes $A_m$). The syntax for expressions can be found in Figure 2, the rules in Figure 3. The bidirectional nature will allow us to establish a precise relationship to the sequent calculus (Section 4), but it does not immediately yield a type checking algorithm since the context merge operation is highly nondeterministic when used to split contexts. An algorithmic system can be found in Section 7.

We obtain the vanilla typing judgment by replacing both checking and synthesis judgments with $\Delta \vdash e : A$, dropping the rules $\implies / \iff$ and $\iff / \implies$, and removing the syntactic form $(e : A_m)$. We further obtain a pure natural deduction system by removing the proof
Adjoint Natural Deduction

Checkable Exps.  \[ e ::= \lambda x. e \quad (\neg \neg) \]
\[ \{ \ell \Rightarrow e_\ell \}_{\ell \in L} \quad (\& \&) \]
\[ \text{susp } e \quad (\uparrow) \]
\[ (e_1, e_2) \quad (\otimes) \]
\[ \text{match } s \ ((x_1, x_2) \Rightarrow e') \quad (1) \]
\[ (\) \quad (\odot) \]
\[ \text{match } s \ ((\) \Rightarrow e') \]
\[ \ell(e) \]
\[ \text{match } s \ ((\ell(x) \Rightarrow e_\ell)_{\ell \in L}) \quad (\odot) \]
\[ \text{down } e \quad (\downarrow) \]
\[ \text{match } s \ (\text{down } x \Rightarrow e') \]
\[ s \]

Synthesizable Exps.  \[ s ::= x \]
\[ s e \quad (\neg \neg) \]
\[ s . \ell \quad (\& \&) \]
\[ \text{force } s \quad (\uparrow) \]
\[ (e : A_m) \]

Figure 2: Expressions for Bidirectional Natural Deduction

terms, although uses of the hypothesis rule then need to be annotated with variables in order to avoid any ambiguities.

The rules maintain a few important invariants, particularly independence:

(i) \( \Delta \vdash e \iff A_m \) presupposes \( \Delta \geq m \)

(ii) \( \Delta \vdash s \Rightarrow A_m \) presupposes \( \Delta \geq m \)

This is somewhat surprising because we think of the synthesis judgment \( s \Rightarrow A_m \) as proceeding top-down rather than bottom-up. Indeed, there are other choices with dependence and structural properties being checked in different places. We picked this particular form because we want general typing \( e : A_m \) to arise from collapsing the checking/synthesis distinction. This means that the two rules \( \Rightarrow / \iff \) and \( \iff / \Rightarrow \) should have no conditions because those would disappear. The algorithmic system in Section 7 checks the conditions in different places.

As an example of interesting rules we revisit \( \downarrow_m A_n \) (where \( n \geq m \) is presupposed). The introduction rule of natural deduction mirrors the right rule of the sequent calculus, which is the case throughout.

\[
\begin{align*}
\Gamma' \geq n \quad \Gamma' \vdash A_n & \quad \downarrow R \quad \Delta' \geq n \quad \Delta' \vdash e \iff A_m \\
\Gamma_W ; \Gamma' \vdash \downarrow_m A_n & \quad \Delta_W ; \Delta' \vdash \text{down } e \iff \downarrow_m A_n
\end{align*}
\]

As is typical for these translations, the elimination rules turns the left rule “upside down” because (like all rules in natural deduction) the principal formula is on the right-hand
\[ \Delta \vdash s \implies A_m \quad \Delta \vdash e \iff A_m \quad \Delta \vdash (e : A_m) \implies A_m \quad \text{hyp} \]

\[ \Delta, x : A_m \vdash e \iff B_m \quad \Delta \vdash \lambda x. e \iff A_m \twoheadrightarrow B_m \quad \text{\textup{\textsc{\textbeta}}I} \]

\[ \Delta \vdash s \implies A_m \twoheadrightarrow B_m \quad \Delta' \vdash e \iff A_m \quad \text{\textup{\textsc{\textbeta}}E} \]

\[ \Delta \vdash \{ \ell \Rightarrow e_{\ell} \}_{\ell \in L} \iff \& \{ \ell : A_{m,}\}_{\ell \in L} \quad \text{\&I} \]

\[ \Delta \vdash \{ \ell \Rightarrow e_{\ell} \}_{\ell \in L} \iff \& \{ \ell : A_{m,}\}_{\ell \in L} \quad \text{\&E} \]

\[ \Delta \vdash e \iff A_k \quad \Delta \vdash \text{\textup{\textit{suspend}}} e \iff \uparrow_k A_k \quad \uparrow I \]

\[ \Delta \vdash \text{\textup{\textit{force}}} s \iff A_k \quad \uparrow E \]

\[ \Delta \vdash e_1 \iff A_m \quad \Delta' \vdash e_2 \iff B_m \quad \text{\textup{\textbf{\textbf{\&I}}} I} \]

\[ \Delta \vdash s \implies A_m \otimes B_m \quad \Delta \vdash (e_1, e_2) \iff A_m \otimes B_m \quad \otimes I \]

\[ \Delta \vdash s \implies A_m \otimes B_m \quad \Delta \geq m \geq r \quad \Delta', x_1 : A_m, x_2 : B_m \vdash e' \iff C_r \quad \text{\textup{\textbf{\textbf{\&E}}} \text{\textup{\textsc{\textbeta}}} E} \]

\[ \Delta \vdash \text{\textup{\textbf{\textbf{\textit{match}}}}} s ((x_1, x_2) \Rightarrow e') \iff C_r \quad \text{\textup{\textbf{\textit{match}}} \text{\textbf{\textbf{\textit{\&}}} E}} \]

\[ \Delta \vdash s \implies 1_m \quad \Delta \geq m \geq r \quad \Delta' \vdash e' \iff C_r \quad \text{\textup{\textbf{\textit{match}}} \text{\textbf{\textbf{\textit{\&}}} E}} \]

\[ \Delta \vdash e \iff A_m \quad \Delta \vdash \ell(e) \iff \oplus \{ \ell : A_{m,}\}_{\ell \in L} \quad \oplus I \]

\[ \Delta \vdash s \implies \oplus \{ \ell : A_{m,}\}_{\ell \in L} \quad \Delta \geq m \geq r \quad \Delta', x : A_{m,}\vdash e_{\ell} \iff C_r \quad (\forall \ell \in L) \quad \text{\textup{\textbf{\textbf{\&E}}} \text{\textbf{\textbf{\&}}} E} \]

\[ \Delta \vdash \text{\textup{\textbf{\textbf{\textit{match}}}}} s ((\ell(x) \Rightarrow e_{\ell})_{\ell \in L} \iff C_r \quad \text{\textup{\textbf{\textbf{\&}}} E} \]

\[ \Delta \geq n \quad \Delta' \vdash e \iff A_n \quad \Delta \vdash s \iff \downarrow_n A_n \quad \Delta \geq m \geq r \quad \Delta', x : A_n \vdash e' \iff C_r \quad \downarrow E \]

\[ \Delta \vdash \text{\textup{\textbf{\textbf{\textit{match}}}}} s (\downarrow x \Rightarrow e') \iff C_r \quad \downarrow E \]

\[ \Delta_W ; \Delta' \vdash \downarrow_n A_n \quad \Delta \vdash \text{\textup{\textbf{\textbf{\textit{match}}}}} s (\downarrow x \Rightarrow e') \iff C_r \quad \downarrow E \]

Figure 3: Implicit Bidirectional Natural Deduction
side of judgment, not the left as in the sequent calculus. This means we now have some conditions to check.

\[
\frac{\Gamma, A_n \vdash C_r}{\Gamma; \downarrow_m^n A_n \vdash C_r} \quad \downarrow L
\]

\[
\Delta \vdash s \implies \downarrow_m^n A_n \quad \Delta \geq m \geq r \quad \Delta', x : A_n \vdash e' \iff C_r \quad \downarrow E
\]

\(\Delta \geq m\) is needed to enforce independence on the first premise. \(m \geq r\) together with \(n \geq m\) enforces independence on the second premise. Similar restrictions appear in the other elimination rules for the positive connectives \((\otimes, 1, \oplus)\).

In general we see the following patterns in the correctness proofs below:

- The identity corresponds to \(\Rightarrow/\Leftarrow\)
- Cut corresponds to \(\Leftarrow/\Rightarrow\)
- Right rules correspond to introduction rules
- Left rules correspond to upside-down elimination rules
  - For negative connectives \((-\circ, \otimes, \uparrow)\) they are just reversed
  - For positive connectives \((\otimes, 1, \oplus, \downarrow)\) in addition a new hypothesis is introduced in a second premise

From the last point we see that the hypothesis \(x : A_m\) should just be read as \(x \Rightarrow A_m\).

We often say a natural deduction is normal, which means that it cannot be reduced, but under which collection of reductions? The difficulty here is that rewrite rules that reduce an introduction of a connective immediately followed by its elimination are not sufficient to achieve deductions that are analytic in the sense that they satisfy the subformula property. To obtain analytic deductions, we have to add permuting conversions.

We follow a different approach by directly characterizing verifications [Dummett, 1991, Martin-Löf, 1983], which are proofs that can be seen as constructed by applying introduction rules bottom-up and elimination rules top-down. By definition, verifications satisfy the subformula property and are therefore analytic and a suitable “normal form” even without defining a set of reductions.

How does this play out here? It turns out that if \(\Delta \vdash e \iff A_m\) then the corresponding proof of \(A_m\) (obtained by erasure of expressions) is a verification if the \(\Leftarrow/\Rightarrow\) rule is disallowed and the \(\Rightarrow/\Leftarrow\) rule is restricted to atomic propositions \(P\). By our above correspondence that corresponds precisely to a cut-free sequent calculus proof where the identity is restricted to atomic propositions. Proof-theoretically, the meaning of a proposition is determined by its verifications, which, by definition, only decompose the given proposition into its components. Compare this with general proofs that do not obey such a restriction.

In the next section we will prove that every proposition that has a proof also has a verification by relating the sequent calculus and natural deduction.

**Example 6 (Church’s \(\lambda I\) calculus)** Church [1941, Chapter II] introduced the \(\lambda I\) calculus in which each bound variable requires at least one occurrence. We obtain the simply-typed \(\lambda I\) calculus with one mode \(S\) with \(\sigma(S) = \{C\}\) and using \(A_S \rightarrow B_S\) as the only type constructor.
Similarly, we obtain the simply-typed \( \lambda \)-calculus with a single mode \( U \) with \( \sigma(U) = \{ W, C \} \) and the simply-typed linear \( \lambda \)-calculus with a single mode \( L \) with \( \sigma(L) = \{ \} \), using \( A \rightarrow B \) as the only type constructor.

**Example 7 (Intuitionistic Natural Deduction)** We obtain (structural) intuitionistic natural deduction with a single mode \( U \) with \( \sigma(U) = \{ W, C \} \), where we can define \( A \lor B = \oplus \{ \text{inl} : A, \text{inr} : B \} \) and \( \bot = \oplus \{ \} \), \( A \land B = \& \{ \pi_1 : A, \pi_2 : B \} \) and \( \top = \& \{ \} \) and \( A \rightarrow B = A \rightarrow C \).

**Example 8 (Intuitionistic S4)** We obtain the fragment of intuitionistic S4 in its dual formulation [Pfenning and Davies, 2001] without possibility \( (\Diamond A) \) with two modes \( V \) and \( U \) with \( V \rightarrow U \) and \( \sigma(V) = \sigma(U) = \{ W, C \} \). As in the DILL example of the adjoint sequent calculus, the mode \( V \) is inhabited only by types \( \top \downarrow A_U \) and we define \( \Box A_U = \downarrow \top \uparrow V A_U \), which is a comonad. The judgment \( \Delta ; \Gamma \vdash C \) true with valid hypotheses \( \Delta \) and true hypothesis \( \Gamma \) is modeled by \( \Delta \downarrow \gamma, \Gamma_U \vdash C_U \).

The structure of verifications is modeled almost exactly with one small exception: we allow hypotheses \( B \downarrow C_U \). Because any proposition \( B \downarrow C_U \), there is only one applicable rule to construct a verification of this judgment: \( \top \downarrow I \) (which, not coincidentally, is invertible).

**Example 9 (Lax Logic)** We obtain natural deduction for lax logic [Benton et al., 1998, Pfenning and Davies, 2001] with two modes, \( U \) and \( X \), with \( U \rightarrow X \) and \( \sigma(U) = \sigma(X) = \{ W, C \} \). The mode \( X \) is inhabited only by \( \downarrow X_A_U \). We define \( \bigcirc A_U = \top \downarrow X_A_U \), which is a strong monad [Benton et al., 1998].

We model the rules of Pfenning and Davies [2001] exactly, except that we allow hypotheses \( B \downarrow X \), which must have the form \( \downarrow X_A_U \). We can eagerly apply \( \downarrow E \) to obtain \( A_U \), which again does not lose completeness by the invertibility of \( \downarrow L \) in the sequent calculus.

We can also obtain linear versions of these relationships following [Benton and Wadler, 1996], although the term calculi do not match up exactly.

### 4 Relating Sequent Calculus and Natural Deduction

Rather than trying to find a complete set of proof reductions for natural deduction, we translate a proof to the sequent calculus, apply cut and identity elimination, and then translate the resulting proof back to natural deduction. This is not essential, but it simultaneously proves the soundness and completeness of natural deduction for adjoint logic and the completeness of verifications. This allows us to focus on the computational interpretation in Section 5 that is a form of substructural functional programming.

For completeness of natural deduction, one might expect to prove that \( \Gamma \vdash C \) in the sequent calculus implies \( \Gamma \vdash e \iff C \) in natural deduction. While this holds, a direct proof would not generate a verification from a cut-free proof. Intuitively, the way the proof proceeds instead is to take a sequent \( x_1 : A_1, \ldots, x_n : A_n \vdash C \) (ignoring modes for the moment) and annotate each antecedent with a synthesizing term and the succedent with an expression \( s_1 \Rightarrow A_1, \ldots, s_n \Rightarrow A_n \vdash e \iff C \). This means we have to account for the variables in \( s_i \), and we do this with a substitution \( \theta \) assigning synthesizing terms to each antecedent in \( \Gamma \). We therefore define substitutions as mapping from variables to synthesizing terms.

Substitutions \( \theta ::= \cdot \mid \theta, x \mapsto s \)
We type substitutions with the judgment $\Delta \vdash \theta \implies \Gamma$, where $\Delta$ contains the free variables in $\theta$. This judgment must respect independence and the structural properties of each antecedent in $\Gamma$, as defined by the following rules:


\[
\begin{array}{c}
\Delta \vdash \theta \implies \Gamma \\
\Delta' \geq m \\
\Delta' \vdash s \implies A_m
\end{array}
\]

\[
\begin{array}{c}
\cdot \vdash () \implies () \\
\Delta ; \Delta' \vdash (\theta, x \mapsto s) \implies (\Gamma, x : A_m)
\end{array}
\]

We will use silently that if $\Delta \vdash \theta \implies \Gamma$ and $\Gamma \geq m$ then $\Delta \geq m$.

We write $e(x)$ and $s'(x)$ for terms with (possibly multiple, possibly no) occurrences of $x$ and $e(s)$ and $s'(s)$ for the result of substituting $s$ for $x$, respectively. Because variables $x : A$ synthesize their types $x \implies A$, the following admissible rules are straightforward assuming the premises satisfy our presuppositions.

**Theorem 10 (Substitution Property)** The following are admissible:

\[
\begin{array}{c}
\Delta \vdash s \implies A_m \\
\Delta', x : A_m \vdash e(x) \iff C_r
\end{array}
\]

\[
\begin{array}{c}
\Delta ; \Delta' \vdash e(s) \iff C_r
\end{array}
\]

\[
\begin{array}{c}
\Delta \vdash s \implies A_m \\
\Delta', x : A_m \vdash s'(x) \implies B_k
\end{array}
\]

\[
\begin{array}{c}
\Delta ; \Delta' \vdash s'(s) \implies B_k
\end{array}
\]

**Proof:** By a straightforward simultaneous rule induction on the second given derivation. In some cases we need to apply monotonicity. For example, if $m$ admits contraction and $\Delta \geq m$, then each hypothesis in $\Delta$ must also admit contraction. □

**Lemma 11 (Substitution Split)** If $\Delta \vdash \theta \implies (\Gamma; \Gamma')$ then there exists $\theta_1$ and $\theta_2$ and $\Delta_1$ and $\Delta_2$ such that $\Delta = \Delta_1; \Delta_2$ and $\Delta_1 \vdash \theta_1 \implies \Gamma$ and $\Delta_2 \vdash \theta_2 \implies \Gamma'$.

**Proof:** By case analysis on the definition of context merge operation and induction on $\Delta \vdash \theta \implies (\Gamma; \Gamma')$. We rely on associativity and commutativity of context merge. We show two cases.

**Case:** $(\Gamma_1, x : A_m ; \Gamma_2, x : A_m) = (\Gamma_1 ; \Gamma_2), x : A_m$ and $C \in \sigma(m)$

\[
\Delta \vdash \theta_{12} \implies \Gamma_1 ; \Gamma_2 \\
\Delta' \geq k \\
\Delta' \vdash s \implies A_m
\]

\[
\Delta ; \Delta' \vdash (\theta_{12}, x \mapsto s) \implies (\Gamma_1 ; \Gamma_2), x : A_m
\]

$\Delta_1 \vdash \theta_1 \implies \Gamma_1$ and

$\Delta_2 \vdash \theta_2 \implies \Gamma_2$ and

$\Delta = \Delta_1 ; \Delta_2$

$\Delta_1 ; \Delta' \vdash \theta_1, x \mapsto s \implies \Gamma_1, x : A_m$

$\Delta_2 ; \Delta' \vdash \theta_2, x \mapsto s \implies \Gamma_2, x : A_m$

by IH

by rule

by rule

since $C \in \sigma(m)$ and $\Delta' \geq m$, we have $C \in \sigma(k)$ for any $B_k \in \Delta'$ by monotonicity

by previous line

$(\Delta_1 ; \Delta') ; (\Delta_2 ; \Delta') = (\Delta_1 ; \Delta_2) ; \Delta' = \Delta ; \Delta'$ by previous line
Case: $\Gamma_1 ; (\Gamma_2, x : A_m) = (\Gamma_1 ; \Gamma_2), x : A_m$ and $x \not\in \text{dom}(\Gamma_1)$

$$\Delta \vdash \theta_{12} \implies \Gamma_1 ; \Gamma_2 \quad \Delta' \geq k \quad \Delta' \vdash s \implies A_m$$

$$\Delta ; \Delta' \vdash (\theta_{12}, x \mapsto s) \implies ((\Gamma_1 ; \Gamma_2), x \mapsto A_m)$$

$$\Delta_1 \vdash \theta_1 \implies \Gamma_1$$

$$\Delta_2 \vdash \theta_2 \implies \Gamma_2$$

$$\Delta = \Delta_1 ; \Delta_2$$

$$\Delta_2 ; \Delta' \vdash \theta_2, x \mapsto s \implies \Gamma_2, x \mapsto A_m$$

$$\Delta_1 ; (\Delta_2 ; \Delta') = (\Delta_1 ; \Delta_2) ; \Delta' = \Delta ; \Delta'$$

by associativity of context merge

Now we have the pieces in place to prove the translation from the sequent calculus to natural deduction.

**Theorem 12 (From Sequent Calculus to Natural Deduction)**

If $\Gamma \vdash A_r$ and $\Delta \vdash \theta \implies \Gamma$ then $\Delta \vdash e \iff A_r$ for some $e$.

**Proof:** By rule induction on the derivation $D$ of $\Gamma \vdash A_r$ and applications of inversion on the definition of substitution. We present several indicative cases. In this proof we write out the variables labeling the antecedents in sequents to avoid ambiguities.

**Case:** $D$ ends in the identity.

$$D = \Gamma_W \vdash x : A_m \vdash A_m \quad \text{id}$$

$$\Delta \vdash \theta \implies (\Gamma_W ; x : A_m)$$

Given

$$\theta = (\theta_W, x \mapsto s)$$

By inversion

$$\Delta = (\Delta_W ; \Delta')$$

with

$$\Delta_W \vdash \theta_W \implies \Gamma_W \quad \text{and} \quad \Delta' \vdash s \implies A$$

By context split

$$\Delta_W$$

satisfies weakening

$$\Delta' \vdash s \iff A$$

By monotonicity

$$\Delta_W ; \Delta' \vdash s \iff A$$

By rule $\Rightarrow / \Leftarrow$

$$\Delta \vdash s \iff A$$

By weakening

Since $\Delta = (\Delta_W ; \Delta')$

**Case:** $D$ ends in cut.

$$D = \begin{array}{c}
\Delta_1 \geq m \geq r \\
\Delta_1 \vdash e_1 \iff A_m \\
\Delta_1 \vdash (e_1 : A_m) \implies A_m \\
\Delta_2, x : A_m \vdash (\theta_2, x \mapsto x) \implies (\Gamma_2, x : A_m) \\
\Delta_2, x : A_m \vdash e_2(x) \iff C_r \\
\Delta_1 ; \Delta_2 \vdash e_2(e_1 : A_m) \iff C_r \\
\end{array}$$

Given

$$\Delta = (\Delta_1 ; \Delta_2), \theta = (\theta_1, \theta_2)$$

with

$$\Delta_1 \vdash \theta_1 \implies \Gamma_1$$

and

$$\Delta_2 \vdash \theta_2 \implies \Gamma_2$$

By context split

$$\Delta_1 \vdash e_1 \iff A_m$$

By IH on $D_1$

$$\Delta_1 \vdash (e_1 : A_m) \implies A_m$$

by rule $\Leftarrow / \Rightarrow$

$$\Delta_2, x : A_m \vdash (\theta_2, x \mapsto x) \implies (\Gamma_2, x : A_m)$$

by subst. rule

$$\Delta_2, x : A_m \vdash e_2(x) \iff C_r$$

By IH on $D_2$

$$\Delta_1 ; \Delta_2 \vdash e_2(e_1 : A_m) \iff C_r$$

by substitution (Theorem 10)
Case: $\mathcal{D}$ ends in $\uparrow L$.

\[
\begin{array}{c}
\Gamma, y : A_k \vdash C_r \\
\mathcal{D}' \\
\end{array}
\]

\[
\mathcal{D} = \Gamma ; x : \uparrow_m^k A_k \vdash C_r \uparrow L
\]

We consider two subcases: $x : \uparrow_m^k A_k \in \Gamma$ and $x : \uparrow_m^k A_k \not\in \Gamma$.

Subcase: $x : \uparrow_m^k A_k \in \Gamma$ and so $C \in \sigma(m)$. Hence, $\Gamma ; x : \uparrow_m^k A_k = \Gamma'$, $x : \uparrow_m^k A_k$.

\[
\begin{array}{c}
\Gamma ; x : \uparrow_m^k A_k = \Gamma \geq r \\
m \geq r \\
\Delta \vdash \theta \Rightarrow \Gamma' \\
\Delta_1 \vdash \theta' \Rightarrow \Gamma' \text{ and } \Delta_2 \vdash s \Rightarrow \uparrow_m^k A_k \text{ and } \Delta_2 \geq m \\
\end{array}
\]

by inversion on subst. rule

where $\Delta = \Delta_1 \uparrow \Delta_2$ and $\theta = (\theta', x \mapsto s)$

\[
\Delta_2 \vdash \text{force } s \Rightarrow A_k
\]

by rule $\uparrow E$

\[
\Delta : \Delta_2 \vdash (\theta, y \mapsto \text{force } s) \Rightarrow (\Gamma, y : A_k)
\]

by subst. rule

$\Delta : \Delta_2 \vdash e \iff C_r$ for some $e$

by IH on $\mathcal{D}'$

$\Delta : \Delta_2 = \Delta$ since $C \in \sigma(m)$ and $\Delta_2 \geq m$, we have $C \in \sigma(j)$ for any $B_j \in \Delta_2$

$\Delta \vdash e \iff C_r$

by previous line

Subcase: $x : \uparrow_m^k A_k \not\in \Gamma$.

\[
\begin{array}{c}
\Gamma, x : \uparrow_m^k A_k \geq r \\
\Gamma \geq r \text{ and } m \geq r \\
\Delta \vdash \theta \Rightarrow (\Gamma, x : \uparrow_m^k A_k) \\
\Delta_1 \vdash \theta' \Rightarrow \Gamma \text{ and } \Delta_2 \vdash s \Rightarrow \uparrow_m^k A_k \text{ and } \Delta_2 \geq m \\
\end{array}
\]

by previous line

by assumption

by inversion on subst. rule

where $\Delta = \Delta_1 \uparrow \Delta_2$ and $\theta = (\theta', x \mapsto s)$

\[
\Delta_2 \vdash \text{force } s \Rightarrow A_k
\]

by rule $\uparrow E$

\[
\Delta_1 : \Delta_2 \vdash (\theta', x \mapsto \text{force } s) \Rightarrow (\Gamma, y : A_k)
\]

by subst. rule

$\Delta_1 : \Delta_2 \vdash e \iff C_r$

by IH on $\mathcal{D}'$

$\Delta \vdash e \iff C_r$

since $\Delta = (\Delta_1 \uparrow \Delta_2)$

Case: $\mathcal{D}$ ends in $\downarrow L$. Similarly to the previous case, we need to distinguish whether $x : \downarrow_m^k A_k$ appears in $\Gamma$.

\[
\begin{array}{c}
\Gamma, y : A_k \vdash C_r \\
\mathcal{D}' \\
\end{array}
\]

\[
\mathcal{D} = \Gamma ; x : \downarrow_m^k A_k \vdash C_r \downarrow L
\]

We only show the case where $x : \downarrow_m^k A_k$ is in $\Gamma$; the other follows similar reasoning.

Subcase: $x : \downarrow_m^k A_k \in \Gamma$ and so $C \in \sigma(m)$. Hence, $\Gamma ; x : \downarrow_m^k A_k = \Gamma' \uparrow \Gamma'$, $x : \downarrow_m^k A_k$.

\[
\begin{array}{c}
\Gamma \geq r \\
k \geq m \geq r \\
\Delta \vdash \theta \Rightarrow \Gamma ; x : \downarrow_m^k A_k \\
\Delta_1 \vdash \theta' \Rightarrow \Gamma' \text{ and } \Delta_2 \vdash s \Rightarrow \downarrow_m^k A_k \text{ and } \Delta_2 \geq m \\
\end{array}
\]

by presupposition

by presuppositions

by assumption

by inversion on subst. rule

where $\Delta = \Delta_1 \downarrow \Delta_2$ and $\theta = (\theta', x \mapsto s)$

$\text{force } y : A_k \vdash x \Rightarrow A_k$

by hyp
Δ, y: A_k ⊢ (θ, x ↦ x) ⊢ (Γ, y : A_k) \quad \text{by subst. rule (using } \Delta \vdash y : A_k = \Delta, y : A_k) \\
Δ, y : A_k \vdash e' \iff C_r \quad \text{for some } e' \quad \text{by IH on } D' \\
Δ_2 \geq m \geq r \quad \text{by previous lines} \\
\Delta_2 : \Delta \vdash \text{match } s \quad (\text{down } y \Rightarrow e') \iff C_r \quad \text{by rule } \downarrow \text{E} \\
\Delta : \Delta_2 = \Delta \quad \text{since } C \in \sigma(m) \text{ and } \Delta_2 \geq m, \text{ we have } C \in \sigma(j) \text{ for any } B_j \in \Delta_2 \\
\Delta \vdash \text{match } s \quad (\text{down } y \Rightarrow e') \iff C_r \quad \text{by previous line} \\
\square

While there are no substitutions involved, the other direction has to take care to introduce a cut only for uses of the \( \iff \Rightarrow \) rule, and identity only for uses of the \( \Rightarrow \iff \) rule. This requires a generalization of the induction hypothesis so that the elimination rules can be turned “upside down”.

**Theorem 13 (From Natural Deduction to Sequent Calculus)**

(i) If \( \Delta \vdash e \iff C_r \) then \( \Delta \vdash C_r \)

(ii) If \( \Delta \vdash s \Rightarrow A_m \) and \( \Delta', x : A_m \vdash C_r \) and \( \Delta \geq r \) then \( \Delta ; \Delta' \vdash C_r \)

**Proof:** By simultaneous rule induction on \( \Delta \vdash e \iff C_r \) and \( \Delta \vdash s \Rightarrow A_m \). We provide four sample cases.

**Case:** The derivation ends in \( \Rightarrow \iff \).

\[
D' = \frac{\Delta \vdash e \iff A_m}{\Delta \vdash \left( e : A_m \right) \Rightarrow A_m} \quad \text{by identity rule}
\]

\[
x : A_m \vdash A_m \\
\Delta \vdash A_m \\
\Delta \vdash A_m \quad \text{By IH(ii) with } \Delta' = (\cdot)
\]

**Case:** The derivation ends in \( \iff \Rightarrow \).

\[
D' = \frac{\Delta \vdash e \iff A_m}{\Delta \vdash \left( e : A_m \right) \iff A_m} \quad \text{by IH(i) on } D' \\
\Delta', x : A_m \vdash C_r \quad \text{Assumption} \\
\Delta \vdash A_m \\
\Delta ; \Delta' \vdash C_m \quad \text{By rule of cut}
\]

**Case:** The derivation ends in \( \uparrow \text{E} \)

\[
D' = \frac{\Delta \geq m \quad \Delta \vdash \left( s : A_k \right) \Rightarrow A_k}{\Delta_W ; \Delta \vdash \left( s : A_k \right) \uparrow \text{E}} \\
\]

**Extended Version**

**February 2, 2024**
As mentioned above, verifications are the foundational equivalent of normal forms in natural deduction. Using the two translations above we can show that every provable proposition has a verification. While we have not written the translations out as functions, they constitute the computational contents of our constructive proof of Theorem 12 and Theorem 13.

**Theorem 14** If $\Delta \vdash e \iff A_m$ then there exists a verification of $\Delta \vdash e \iff A_m$.

**Proof:** Given an arbitrary deduction of $\Delta \vdash e \iff A_m$, we can use Theorem 13 (ii) to translate it to a sequent derivation of $\Delta \vdash A_m$.

By the admissibility of cut and identity (Theorem 2), we can obtain a long cut-free proof of $\Delta \vdash A_m$.

We observe that the translation of Theorem 12 translates only cut to $\iff/\Rightarrow$ and only identity to $\Rightarrow/\iff$. Using the translation back to natural deduction from a long cut-free proof therefore results in a verification.

\[\Delta \vdash e : A \iff A_m \]
Adjoint Natural Deduction

- \(\text{susp}_k^m e : \uparrow_k^m A_k\) if \(e : A_k\)
- \(\text{force}_k^m e : A_k\) if \(e : \uparrow_k^m A_k\)
- \(\text{down}_m^a e : \downarrow_m^a A_n\) if \(e : A_n\)
- \(\text{match}_m e M_r : C_r\) if \(e : A_m\)

We give a sequential call-by-value semantics similar to the K machine (e.g., [Harper, 2016, Chapter 28]), but maintaining a global environment similar to the Milner Abstract Machine [Accattoli et al., 2014]. There are two forms of state in the machine:

- \(\eta ; K \triangleright_m e\) (evaluate \(e\) of mode \(m\) under continuation stack \(K\) and environment \(\eta\))
- \(\eta ; K \triangleleft_m v\) (pass value \(v\) of mode \(m\) to continuation stack \(K\) in environment \(\eta\))

In the first, \(e\) is an expression to be evaluated and \(K\) is a stack of continuations that the value of \(e\) is passed to for further computation. The second then passes this value \(v\) to the continuation stack.

The global environment \(\eta\) maps variables to values, but these values may again reference other variables. In this way it is like Launchbury’s [1993] heap, a connection we exploit in Section 6 to model call-by-need. Because we maintain a global environment, we do not need to build closures, nor do we need to substitute values for variables. Instead, we only (implicitly) rename variables to make them globally unique. This form of specification allows us to isolate the dynamic use of variables, which means we can observe the computational consequences of modes and their substructural nature. We could also use the translation to the sequent calculus and then observe the consequence with an explicit heap [Pruiksma and Pfenning, 2022, Pfenning and Pruiksma, 2023], but in this paper we study natural deduction and functional computation more directly.

The syntax for continuations, environments, values, and machine states is summarized in Figure 4. Although not explicitly polarized (as in Levy [2006]), values of negative type \((\rightarrow, \&, \uparrow)\) are lazy in the sense that they abstract over unevaluated expressions, while values of positive types \((\otimes, 1, \oplus, \downarrow)\) are constructed from other values. This will be significant in our analysis of the computational properties of modes. Continuation frames just reflect the left-to-right call-by-value nature of evaluation.

Values are typed as expressions. Frames are typed with \(\Gamma \vdash f : B_k < A_m\), which means \(f\) takes a value of type \(A_m\) and passes a value of type \(B_k\) further up the continuation stack. We show sample rules for a negative (\(\rightarrow\)) and a positive (\(\oplus\)) type.

\[
\Delta \vdash e_2 : A_m \\
\Delta \vdash \_e_2 : B_m < (A_m \rightarrow B_m)
\]

\[
\Delta \vdash v_1 : A_m \rightarrow B_m \\
\Delta \vdash v_1 \_ : B_m < A_m
\]

\[
\Delta_w \vdash \ell(\_) : \oplus\{\ell : A_m^\ell\}_{\ell \in L} < A_m \land
\Delta \vdash \text{match}_m \_ (\ell(x) \Rightarrow e(x)) : C_r < \oplus\{\ell : A_m^\ell\}_{\ell \in L}
\]

\[
\Delta_w \vdash e : A_m < A_m \\
\Delta \vdash K : C_r < B_k \land
\Delta' \vdash f : B_k < A_m
\]

\[
\Delta ; \Delta' \vdash K \cdot f : C_r < A_m
\]
Regarding environments we face a fundamental choice. One possibility is to extend the term language of natural deduction with explicit constructs for weakening and contraction. Then, similar to Girard and Lafont [1987], no garbage collection would be required during evaluation since uniqueness of references to variables would be maintained.

We pursue here an alternative that leads to slightly deeper properties. We leave the structural rules implicit as in the rules so far. This means that variables of linear mode (that is, a mode that allows neither weakening nor contraction) have uniqueness of reference and their bindings can be deallocated when dereferenced. Variables of structural mode (that is, a mode that allows both weakening and contraction) are simply persistent in the dynamics and therefore could be subject to an explicit garbage collection algorithm.

A difficulty arises with variables that only admit contraction but not weakening. After they are dereferenced the first time, they may or may not be dereferenced again. That is, they could be implicitly weakened after the first access. In order to capture this we introduce a new form of typing \([x : A_m]\) and binding \([x \mapsto v]\) we call provisional. A provisional
binding does not need to be referenced even if \( m \) does not admit weakening. The important new property is that an “ordinary” variable \( y : A_k \) that does not admit weakening can not appear in a binding \( [x \mapsto v] \). In addition, all the usual independence requirements have to be observed.

The rules for typing expressions, continuations, etc. are extended in the obvious way, allowing variables \([x : A_m]\) to be used or ignored (as a part of some \( \Delta_W \)). We extend the context merge operation as follows, keeping in mind that \( x \) of \( x \) \( \sigma(m)\), while \([x : A_m]\) does not.

\[
\begin{align*}
(\Delta_1, [x : A_m]) ; (\Delta_2, [x : A_m]) &= (\Delta_1 ; \Delta_2), [x : A_m] \quad \text{provided } C \in \sigma(m) \\
(\Delta_1, x : A_m) ; (\Delta_2, [x : A_m]) &= (\Delta_1 ; \Delta_2), x : A_m \quad \text{provided } C \in \sigma(m) \\
(\Delta_1, [x : A_m]) ; (\Delta_2, x : A_m) &= (\Delta_1 ; \Delta_2), x : A_m \quad \text{provided } C \in \sigma(m) \\
(\Delta_1, [x : A_m]) ; (\Delta_2) &= (\Delta_1 ; \Delta_2), [x : A_m] \quad \text{provided } x \notin \text{dom}(\Delta_2) \\
\Delta_1 ; (\Delta_2, [x : A_m]) &= (\Delta_1 ; \Delta_2), [x : A_m] \quad \text{provided } x \notin \text{dom}(\Delta_1)
\end{align*}
\]

We have the following typing rules for environments. \( \Delta_W \) now means that every declaration in \( \Delta \) can be weakened, either explicitly because its mode allows weakening, or implicitly because it is provisional.

\[
\begin{align*}
\eta : (\Delta ; \Delta') &\quad \Delta' \geq m \quad \Delta' \vdash v : A_m \quad \eta : (\Delta ; \Delta') &\quad \Delta' \geq m \quad \Delta'_W \vdash v : A_m \\
(\cdot) : (\cdot) &\quad (\eta, x \mapsto v) : (\Delta, x : A_m) &\quad (\eta, x \mapsto v) : (\Delta, [x : A_m])
\end{align*}
\]

As an example, consider \( \eta_0 = (x \mapsto (\lambda f. f x), y \mapsto \lambda f. f x) \) where the mode of variables is immaterial, but let’s fix them to be \( L \) with \( \sigma(L) = \{ \} \).

\[
\begin{align*}
(\cdot) : (\cdot) &\quad \vdash (\cdot) : 1_l \\
(x \mapsto ()) : (x : 1_l) &\quad x : 1_l \vdash \lambda f. f x : (1_l \rightarrow A_l) \rightarrow A_l \\
(x \mapsto (), y \mapsto \lambda f. f x) &\quad (y : (1_l \rightarrow A_l) \rightarrow A_l)
\end{align*}
\]

We observe that the binding of \( x \mapsto () \) does not contribute a declaration \( x : 1 \) to the result context due to the occurrence of \( x \) in the value of \( y \).

Now consider a slightly modified version where the mode of both \( x \) and \( y \) is \( S \) with \( \sigma(S) = \{ C \} \), and the binding of \( y \mapsto \ldots \) becomes provisional. This modified example is no longer well-typed.

\[
\begin{align*}
(\cdot) : (\cdot) &\quad \vdash (\cdot) : 1_s \\
(x \mapsto ()) : (x : 1_s) &\quad x : 1_s \vdash \lambda f. f x : (1_s \rightarrow A_s) \rightarrow A_s \\
(x \mapsto (), y \mapsto \lambda f. f x) &\quad (y : [(1_s \rightarrow A_s) \rightarrow A_s])
\end{align*}
\]

The problem is at the rule application marked by ??. The variable \( y \) does not need to be used, despite its mode, because the binding is provisional. This means that \( x \) might also not be used because its only occurrence is in the value of \( y \). But that is not legal, since the mode of \( x \) does not admit weakening and the binding is not provisional.
We type abstract machine states with the type of their final answer, that is $s : C_r$.

\[
\begin{align*}
\eta : (\Delta ; \Delta') & \quad \Delta \vdash K : C_r < A_m \quad \Delta' \vdash e : A_m \\
(\eta ; K \triangleright_m e) & : C_r \\
\eta : (\Delta ; \Delta') & \quad \Delta \vdash K : C_r < A_m \quad \Delta' \vdash v : A_m \\
(\eta ; K \triangleleft_m v) & : C_r
\end{align*}
\]

We now continue with the computational rules for our abstract machine. The rules are in Figure 5. We factor out passing a value to a match, $\eta ; v \triangleright_m M = \eta' ; e'$ that produces a (possibly extended) environment $\eta'$ and expression $e'$. In all cases below, we presuppose the variable names are chosen so the extended environment has unique bindings for each variable. For an extension with mutual recursion, see Section 6.

We obtain the following expected theorems of preservation and progress.

**Theorem 15 (Preservation)** If $S : A$ and $S \rightarrow S'$ then $S' : A$.

**Proof:** By cases on $S \rightarrow S'$, applying inversion to the typing of $S$ and assembling a typing derivation of $S'$ from the resulting information.

The trickiest case involves dereferencing a variable $x \mapsto v$ admitting contraction. It is sound because every variable $y$ occurring in $v$ must also admit contraction by monotonicity and, furthermore, such variables still have an occurrence in the value $v$ that is being returned. Therefore in the typing of the environment we can now type $[x \mapsto v]$ with $[x : A_m]$.

A machine state is final if it has the form $\eta ; \epsilon \triangleleft_m v$, that is, if a value is returned to the empty continuation in some global environment $\eta$. In order to prove progress, we need to characterize values of a given type using a canonical forms property. Note that we allow a context $\Delta$ to provide for the variables that may be embedded in a value of negative type ($\sim$, $\&$, $\uparrow$), but that a variable by itself does not count as a value.

**Theorem 16 (Canonical Forms)** If $\Delta \vdash v : A_m$ then one of the following applies:

1. if $A_m = B_m \rightarrow C_m$ then $v = \lambda x. e(x)$ for some $e$
2. if $A_m = \&\{\ell : A_m^\ell\}_{\ell \in L}$ then $v = \{\ell \Rightarrow e_\ell\}_{\ell \in L}$ for some set $e_\ell$
3. if $A_m = \uparrow^m_k B_k$ then $v = \text{susp}_k^m e$
4. if $A_m = B_m \otimes C_m$ then $v = (v_1, v_2)$ for values $v_1$ and $v_2$
5. if $A_m = 1$ then $v = ()$
6. if $A_m = \oplus\{\ell : B_m^\ell\}_{\ell \in L}$ then $v = \ell(v')$ for some $\ell \in L$ and value $v'$
7. if $A_m = \downarrow^m_m A_n$ then $v = \text{down}_m^m v'$ for some value $v'$

**Theorem 17 (Progress)** If $S : C_r$ then either $S$ is final or $S \rightarrow S'$ for some $S'$.
\[
\eta ; (v_1, v_2) \triangleright_m ((x_1, x_2) \Rightarrow e'(x_1, x_2)) = \eta, x_1 \mapsto v_1, x_2 \mapsto v_2 ; e'(x_1, x_2)
\]
\[
\eta ; \ell(v) \triangleright_m (\ell(x) \Rightarrow e'(x))_{\ell \in L} = \eta, x \mapsto v ; e'(x)
\]
\[
\eta ; \text{down}^n_m v \triangleright_m (\text{down}(x) \Rightarrow e'(x)) = \eta, x \mapsto v ; e'(x)
\]
\[
\eta, x \mapsto v, \eta' ; K \triangleright_m x \rightarrow \eta, \eta' ; K \downarrow_m v \quad (C \notin \sigma(m))
\]
\[
\eta, y \mapsto v, \eta' ; K \triangleright_m x \rightarrow \eta, y \mapsto v, \eta' ; \text{down}_m v \quad (C \in \sigma(m))
\]
\[
\eta ; K \triangleright_r \text{match}_m e M_r \rightarrow \eta ; K \cdot \text{match}_m \downarrow_m e \quad (\odot, 1, \oplus, \downarrow)
\]
\[
\eta ; K \cdot (\text{match}_m \downarrow_m M_r) \downarrow_m v \rightarrow \eta' ; K \triangleright_r e' \quad \text{where } \eta ; v \downarrow_m M_r = \eta' ; e'
\]
\[
\eta ; K \triangleright_m \lambda x.e(x) \rightarrow \eta ; K \downarrow_m \lambda x.e(x) \quad (-\circ)
\]
\[
\eta ; K \triangleright_m (e_1, e_2) \rightarrow \eta ; K \cdot (\_, e_2) \triangleright_m e_1
\]
\[
\eta ; K \cdot (\_, e_2) \downarrow_m v_1 \rightarrow \eta ; K \cdot (v_1, \_) \triangleright_m e_2
\]

---

**Figure 5: Computation Rules**

**Proof:** By cases on the typing derivation for the configuration and inversion on the typing of the embedded frames, values, and expressions. We apply the canonical forms theorem when we need the shape of a value.

Purely positive types play an important role because we view values of these types as *directly observable*, while values of negative types can only be observed indirectly through...
their elimination forms.

Purely positive types \( A^+, B^+ \ ::= A^+ \otimes B^+ | 1 | \oplus \{ \ell : A^+_\ell \}_{\ell \in L} | \downarrow A^+ \)

Values of purely positive types are closed, even if values of negative types may not be.

**Lemma 18 (Positive Values)** If \( \Delta \vdash v : A^+_r \) then \( \cdot \vdash v : A^+_r \) and all declarations in \( \Delta \) admit weakening (either due to their mode or because they are provisional).

**Proof:** By induction on the structure of the typing derivation, recalling that variables are not values.

We call a variable \( x : A_m \) linear if \( \sigma(m) = \{ \} \), that is, the mode \( m \) admits neither weakening or contraction. We extend this term to types, bindings in the environment, etc. in the obvious way.

**Theorem 19 (Freedom from Garbage)** If \( \cdot \vdash e : A^+_r \) and \( \cdot ; \epsilon \triangleright_r e \rightarrow^* \eta ; \epsilon \triangleright_r v \), then \( \eta \) does not contain a binding \( x \mapsto v \) with \( \sigma(m) = \{ \} \).

**Proof:** Because \( A^+_r \) is purely positive, we know by **Lemma 18** that \( v \) is closed.

When the continuation \( K \) is empty, the typing rule for valid states implies that \( \eta : \Delta \) and \( \Delta \vdash v : A^+_r \) for some \( \Delta \). Since \( v \) is closed, \( \Delta \) cannot contain any linear variables.

Then we prove by induction on the typing of \( \eta \) that none of variables in \( \eta \) can be linear. In the inductive case

\[
\eta' : (\Delta ; \Delta') \quad \Delta' \geq m \quad \Delta' \vdash v : A_m
\]

\[
(\eta', x \mapsto v) : (\Delta, x : A_m)
\]

we know that \( m \) must admit weakening or contraction or both. Since \( \Delta' \geq m \), by monotonicity, \( \Delta' \) must also admit weakening or contraction and we can apply the induction hypothesis to \( \eta' : (\Delta ; \Delta') \).

We call a variable \( x_m, \) an expression \( e : A_m, \) or a binding \( x \mapsto v \) strict if \( \sigma(m) \subseteq \{C\} \), that is, \( m \) does not admit weakening.

**Theorem 20 (Strictness)** If \( \cdot \vdash e : A^+_r \) and \( \cdot ; \epsilon \triangleright_r e \rightarrow^* \eta ; \epsilon \triangleright_r v \), then every strict binding in \( \eta \) is of the form \([x \mapsto v]\).

**Proof:** Because \( A^+_r \) is purely positive, we know by **Lemma 18** that \( v \) is closed.

When the continuation \( K \) is empty, the typing rule for valid states implies \( \eta : \Delta \) and \( \Delta \vdash v : A^+_r \) for some \( \Delta \). Since \( v \) is closed, \( \Delta \) contains strict variables only in the form \([x : A_m]\).

We prove by induction on the typing of \( \eta \) all strict variables in \( \eta \) have the form \([x \mapsto w]\). There are two inductive cases.

\[
\eta' : (\Delta ; \Delta') \quad \Delta' \geq m \quad \Delta' \vdash w : A_m
\]

\[
(\eta', x \mapsto w) : (\Delta, x : A_m)
\]

Since \( m \) is not strict, it must admit weakening. Since \( \Delta' \geq m \), every variable in \( \Delta' \) must also admit weakening by monotonicity, so we can apply the induction hypothesis to \( \Delta;\Delta' \).
$$\eta' : (\Delta ; \Delta'_W) \quad \Delta'_W \geq m \quad \Delta'_W \vdash w : A_m$$

$$\vdash (\eta', [x \mapsto v]) : (\Delta, [x : A_m])$$

Any declaration in $\Delta'_W$ either directly admits weakening or is of the form $[y : A_k]$ for a strict $k$ so we can apply the induction hypothesis to $\eta' : (\Delta ; \Delta'_W)$.

In this context of call-by-value, this property expresses that every strict variable will be read at least once, since a binding $[x \mapsto v]$ arises only from reading the value of $x$.

**Theorem 21 (Dead Code)** If $\cdot \vdash e : A^+_m$ and $\cdot \triangleright_r e \rightarrow^* \eta ; e \blacktriangleleft_r v$ then every state during the computation either evaluates $\triangleright_m$ or returns $\blacktriangleleft_m$ for $m \geq r$.

**Proof:** Most rule do not change the subject’s mode. Several rules potentially raise the mode, name evaluating a *match*, a *force*, or a *down*. For each of these there is a corresponding rule lowering the mode back to its original, namely return a value to a *match*, to a *force*, or to a *down*.

We say the mode of a frame $f$ is the mode of the following state after a value is returned to $f$. We prove by induction over the computation that in all states, all continuation frames and subjects have modes $m \geq r$.

**Corollary 22 (Erasure)** Assume $\cdot \vdash e : A^+_m$ and $\cdot \triangleright_r e \rightarrow^* \eta ; e \blacktriangleleft_r v$. Let $\Omega$ be a new term of every type and no transition rule.

If we obtain $e'$ by replacing all subterms $e' : B_k$ for $k \nleq r$ with $\Omega$, then evaluation $e'$ still terminates in a final state. This final state differs from $v$ in that subterms of mode $k \nleq r$ are also replaced by $\Omega$.

**Proof:** The computation of $e'$ parallels that of $e$. It would only get stuck for a state $\eta' ; K' \triangleright_k \Omega$, but that is impossible by the preceding dead code theorem since $k \nleq r$.

### 6 Adding Recursion and Call-by-Need

To add recursion at the level of types we allow a global environment $\Sigma$ of equirecursive type definition $t_m = A_m$ that may be mutually recursive and restrict $A_m$ to be contractive, that is, start with a type constructor and not a type name. Since we view them as equirecursive we can silently unfold them, and the only modification is in the rule for directional change. The typing judgment is now parameterized by a signature $\Sigma$, but since it is fixed we generally omit it.

$$\Delta \vdash s \Rightarrow A'_m \quad A'_m = A_m$$

$$\Delta \vdash s \Leftarrow A_m \quad \Rightarrow/\Leftarrow$$

Type equality follows the standard coinductive definition [Amadio and Cardelli, 1993]. It can also be replaced by subtyping $A'_m \leq A_m$ without affecting soundness, but a formal treatment is beyond the scope of this paper.

In order to model recursion at the level of expressions, we allow *contextual definitions* [Nanevski et al., 2008] in the signature. They have the form $f[\Delta] : A_m = e$ and may be
mutually recursive, so all definitions are checked with respect to the same global signature \( \Sigma \), using the \( \vdash \Sigma' \, \text{sig} \) judgment. We use here a form of checkable substitution \( \Delta' \vdash \theta \iff \Delta \) which we did not require earlier, and which has the obvious pointwise definition.

\[
\begin{align*}
\vdash \Sigma' \, \text{sig} & \quad \vdash \Sigma' \, A_m \, \text{type} \quad A_m \, \text{contractive} & \vdash \Sigma', t_m = A_m \\
\vdash \Sigma' \vdash \Delta \vdash \Sigma' \, e \iff A_m & \quad \vdash \Sigma' \, f[\Delta] : A_m = e \, \text{sig} & \vdash \Sigma' \, (\cdot) \, \text{sig}
\end{align*}
\]

\[
\begin{align*}
\Delta' \vdash \Sigma \vdash f[\theta] & \iff A_m \\
\end{align*}
\]

The reason that definitions are contextual is partly pragmatic. We are used to reifying, say, \( f(x : A)(y : B) : C = e(x, y) \) as \( f : A \to B \to C = \lambda x. \lambda y. e(x, y) \). But because \( A_m \to B_m \) requires the same mode on both sides, we would have to insert shifts, or generalize the function type and thereby introduce multiple mode-shifting constructors. By using contextual definitions, we can directly express dependence on a mixed context. For example, \( f[x : A_m, y : B_k] : C_r = e(x, y) \) which, by presupposition, requires \( m, k \geq r \) so that the judgment \( x : A_m, y : B_k \vdash e(x, y) \iff C_r \) is well-formed.

In order to demonstrate call-by-need, let’s assume that contextual definitions are indeed treated in call-by-need manner. In order to model this, we create a new form of environment entry \( x \mapsto e \) where \( e \) is an unevaluated expression. A substitution \( \theta = (x_1 \mapsto e_1, \ldots, x_n \mapsto e_n) \) then also represents an environment in which none of the expressions \( e_i \) depend on any of the variables \( x_i \). We also need to new continuation frame \( x \mapsto \_ \) to create a value binding in the environment. We have the following new rules:

\[
\begin{align*}
\eta ; K \triangleright_m f[\theta] & \quad \rightarrow \quad \eta, \theta ; K \triangleright_m e & \text{for } f[\Delta] : A_m = e \in \Sigma \\
\eta, x \mapsto e, \eta' ; K \triangleright_m x & \quad \rightarrow \quad \eta, [x \mapsto e], \eta' ; K \cdot (x \mapsto \_) \triangleright_m e \\
\eta, [x \mapsto e], \eta' ; K \cdot (x \mapsto \_) & \blacktriangleright_m v \quad \rightarrow \quad \eta, x \mapsto v, \eta' ; K \blacktriangleright_m v
\end{align*}
\]

We can further simplify these rules if \( m \) does not admit contraction, dropping \([x \mapsto e]\) and \([x \mapsto v]\). We could also model “black holes” by rebinding not \([x \mapsto e]\) in the second rule, but \([x \mapsto \Omega]\) where \( \Omega \) has every type, but does not reduce.

The theorems from the previous section continue to hold. They imply that in a final state of observable type there will be no binding \( x \mapsto e \) in the environment if the mode of \( x \) is strict (that is, does not admit contraction). In other words, strict expressions are always evaluated.

### 7 Algorithmic Type Checking

The bidirectional type system of Section 3 is not yet algorithmic, among other things because splitting a given context into \( \Delta = (\Delta_1 ; \Delta_2) \) is nondeterministic. One standard solution is track which are hypotheses are used in one premise (which ends up \( \Delta_1 \)), subtract them from the available ones, and pass the remainder into the second premise (which ends up \( \Delta_2 \) together with an overall remainder) [Cervesato et al., 2000]. This originated in proof search, but here when we actually have a proof terms available to check, other options are available. Additive resource management computes the used hypotheses (rather
than the unused ones) and merges ("adds") them [Atkey, 2018], which is conceptually slightly simpler and also has been shown to be more efficient [Hughes and Orchard, 2020].

The main complication in the additive approach are internal and external choice, more specifically, the $\&R$ and $\oplus L$ rules when the choice is empty. For example, while check $\Delta \vdash \{\} \iff \&\{\}$ any subset of $\Delta$ could be used. We reuse the idea from the dynamics to have hypothesis $[x : A_m]$ which may or may not have been used. Unfortunately, this leads to a plethora of different judgments, but it is not clear how to simplify them. In defining the additive approach, the main judgment is

$$\Gamma \vdash e \iff A_m / \Xi$$

where $\Gamma$ is a plain (that is, free of provisional hypotheses) context containing all variables that might occur in $e$ (regardless of mode or structural properties) and $\Xi$ is a context that may contain provisional hypotheses. We maintain the mode invariant $\Xi \geq m$ (even if it may be the case that $\Gamma \not\geq m$).

Because we keep the contexts $\Delta$ free of provisional hypotheses, we define the relation $\Xi \equiv \Delta$ which may remove or keep provisional hypotheses.

$$\begin{align*}
(\Xi, x : A_m) & \equiv (\Delta, x : A) \quad \text{if} \quad \Xi \not\equiv \Delta \\
(\Xi, [x : A_m]) & \equiv (\Delta, x : A) \quad \text{if} \quad \Xi \not\equiv \Delta \\
(\Xi, [x : A_m]) & \equiv \Delta \quad \text{if} \quad \Xi \equiv \Delta \\
(\cdot) & \equiv (\cdot)
\end{align*}$$

With this relation, we can state the soundness of algorithmic typing (postponing the proof).

**Theorem 23 (Soundness of Algorithmic Typing)**

If $\Gamma \vdash e \iff A_m / \Xi$ and $\Xi \equiv \Delta$ then $\Delta \vdash e \iff A_m$.

For completeness we need a different relation $\Delta \geq \Xi$ which means that $\Xi$ contains a legal subset of the hypotheses in $\Delta$. This means hypotheses in $\Delta$ might be in $\Xi$ (possibly provisional) or not, but then only if they can be weakened.

$$\begin{align*}
(\Delta, x : A_m) & \geq (\Xi, x : A_m) \quad \text{if} \quad \Delta \geq \Xi \\
(\Delta, x : A_m) & \geq ([\Xi, x : A_m]) \quad \text{if} \quad \Delta \geq \Xi \\
(\Delta, x : A_m) & \geq \Xi \quad \text{if} \quad \Delta \geq \Xi \quad \text{provided} \ W \in \sigma(m) \\
(\cdot) & \geq (\cdot)
\end{align*}$$

With this relation we can state the completeness of algorithmic typing (also postponing the proof).

**Theorem 24 (Completeness of Algorithmic Typing)**

If $\Delta \vdash e \iff A_m$ then $\Delta \vdash e \iff A_m / \Xi$ for some $\Xi$ with $\Delta \geq \Xi$.

For the algorithm itself we need several operations. The first, $\Xi \setminus x : A$ removes $x : A$ from $\Xi$ if this is legal operation. Its prototypical use is in the $\rightarrow I$ rule:

$$\begin{align*}
\Gamma, x : A_m \vdash e & \iff B_m / \Xi \\
\Gamma \vdash \lambda x. e & \iff A_m \rightarrow B_m / ([\Xi \setminus x : A_m]) \quad \rightarrow I
\end{align*}$$
For this rule application to be correct, \( x : A_m \) must either have been used and therefore occur in \( \Xi \), or the mode \( m \) must allow weakening.

\[
\begin{align*}
(\Xi, x : A_m) \ \parallel x : A_m &= \Xi \\
(\Xi, [x : A_m]) \ \parallel x : A_m &= \Xi \\
(\Xi, y : B_k) \ \parallel x : A_m &= (\Xi \setminus x : A_m), y : B_k \text{ provided } y \neq x \\
(\Xi, [y : B_k]) \ \parallel x : A_m &= (\Xi \setminus x : A_m), y : B_k \text{ provided } y \neq x \\
(\cdot) \ \parallel x : A &= (\cdot)
\end{align*}
\]

We have the following fundamental property (modulo exchange on contexts, as usual).

**Lemma 25** If \( \Xi \setminus x : A_m = \Xi' \) then

(i) either \( \Xi = (\Xi', x : A_m) \)

(ii) or \( \Xi = (\Xi', [x : A_m]) \)

(iii) or \( \Xi = \Xi' \) with \( x \not\in \text{dom}(\Xi) \) and \( W \in \sigma(m) \).

We also need two forms of context restriction. The first \( \Xi \parallel m \) removes all hypotheses whose mode is not greater or equal to \( m \) to restore our invariant. It fails if \( \Xi \) contains a used hypothesis \( B_r \) with \( r \not\geq m \). It is used only in the \( \uparrow I \) rule to restore the invariant,

\[
\begin{align*}
\Gamma \vdash e &\iff A_k / \Xi \\
\Gamma \vdash \text{suspend } e &\iff \uparrow_{m} \rightarrow A_k / \Xi \parallel m \uparrow I \\
(\Xi, x : A_k) \parallel m &= \Xi \parallel m, x : A_k \text{ provided } k \geq m \\
(\Xi, [x : A_k]) \parallel m &= \Xi \parallel m, [x : A_k] \text{ provided } k \geq m \\
(\Xi, [x : A_k]) \parallel m &= \Xi \parallel m \text{ provided } k \not\geq m \\
(\cdot) \parallel m &= (\cdot)
\end{align*}
\]

It has the following defining property.

**Lemma 26** If \( \Delta \geq m \) and \( \Delta \geq \Xi \) then \( \Delta \geq \Xi \parallel m \).

The second form of context restriction occurs in the case of an empty internal or external choice. All of the hypothesis that are allowed by the independence principle could be considered used, but they might also not. We write \([\Gamma]_m\) and define:

\[
\begin{align*}
([\Gamma, x : A_k])_m &= [\Gamma]_m, [x : A_k] \text{ provided } k \geq m \\
([\Gamma, x : A_k])_m &= [\Gamma]_m \text{ provided } k \not\geq m \\
([\cdot])_m &= (\cdot)
\end{align*}
\]

It is used only in the nullary case for internal and external choice.

\[
\begin{align*}
\Gamma \vdash \{ \} &\iff \&_m \{ \} / [\Gamma]_m \&_0 J_0 \\
\Gamma \vdash \text{match } s (\cdot) &\iff C_r / \Xi ; [\Gamma]_r \oplus E_0
\end{align*}
\]
Adjoint Natural Deduction

We come to the final operation $\Xi_1 \sqcup \Xi_2$ which is needed for $\& I$ and $\oplus E$. We show the case of $\& I_2$, that is, the binary version.

\[
\Gamma \vdash e_1 \iff A_m / \Xi_1 \quad \Gamma \vdash e_2 \iff B_m / \Xi_2
\]

\[
\Gamma \vdash \{
\pi_1 \Rightarrow e_1, \pi_2 \Rightarrow e_2
\} \iff \&\{\pi_1 : A_m, \pi_2 : B_m\} / \Xi_1 \sqcup \Xi_2
\]

Variables $y : B_k$ that are definitely used in $\Xi_1$ or $\Xi_2$ must also be used in the other, or be available for weakening. This could be because they are provisional $[y : B_k]$ or because mode $k$ admits weakening. This idea is captured by the following definition.

\[
(\Xi_1, x : A_m) \sqcup (\Xi_2, x : A_m) = (\Xi_1 \sqcup \Xi_2), x : A_m
\]

\[
(\Xi_1, [x : A_m]) \sqcup (\Xi_2, x : A_m) = (\Xi_1 \sqcup \Xi_2), x : A_m
\]

\[
(\Xi_1, [x : A_m]) \sqcup (\Xi_2, [x : A_m]) = (\Xi_1 \sqcup \Xi_2), x : A_m
\]

\[
\Xi_1 \sqcup (\Xi_2, [x : A_m]) = (\Xi_1 \sqcup \Xi_2), x : A_m
\]

\[
(\Xi_1, x : A_m) \sqcup \Xi_2 = (\Xi_1 \sqcup \Xi_2), x : A_m \quad \text{for} \ x \notin \text{dom}(\Xi_2), \ W \in \sigma(m)
\]

\[
\Xi_1 \sqcup (\Xi_2, x : A_m) = (\Xi_1 \sqcup \Xi_2), x : A_m \quad \text{for} \ x \notin \text{dom}(\Xi_1), \ W \in \sigma(m)
\]

\[
(\Xi_1, [x : A_m]) \sqcup \Xi_2 = \Xi_1 \sqcup \Xi_2 \quad \text{for} \ x \notin \text{dom}(\Xi_2)
\]

\[
\Xi_1 \sqcup \Xi_2, [x : A_m] = \Xi_1 \sqcup \Xi_2 \quad \text{for} \ x \notin \text{dom}(\Xi_1)
\]

\[
(\cdot) \sqcup (\cdot) = (\cdot)
\]

We have the following two properties, expressing the property of a least upper bound in two slightly asymmetric judgments.

**Lemma 27**

(i) If $\Delta \geq \Xi_1$ and $\Delta \geq \Xi_2$ then $\Delta \geq \Xi_1 \sqcup \Xi_2$.

(ii) If $\Xi_1 \sqcup \Xi_2 \sqsupseteq \Delta$ then for some $\Delta_1$, $\Delta_1^1$, $\Delta_2$, $\Delta_2^1$, $\Delta_2^2$ we have $\Xi_1 \sqsupseteq \Delta_1$, $\Xi_2 \sqsupseteq \Delta_2$ and $\Delta = \Delta_1^1 \setminus \Delta_2^2 \sqcup \Delta_2^2 = \Delta_2^2 \sqcup \Delta_2^2$.

The complete set of rules for algorithmic typing can be found in Figure 6.

**Lemma 28** If $\Gamma \vdash e \iff A_m / \Xi$ then $\Xi \geq m$

**Proof:** By straightforward rule induction. □

**Lemma 29 (Context Extension)** If $\Gamma \vdash e \iff A_m / \Xi$ and $\Xi \sqsupseteq \Delta$ and $\Gamma' \sqsupseteq \Gamma$ then $\Gamma' \vdash e \iff A_m / \Xi'$ for some $\Xi'$ with $\Xi' \sqsupseteq \Delta$.

**Proof:** By rule induction on the given derivation and inversion on the definition of $\Xi \sqsupseteq \Delta$. □

**Proof:** (of soundness, Theorem 23) By rule induction on the algorithmic typing derivation and inversion of the $\Xi \sqsupseteq \Delta$ judgment. □

**Proof:** (of completeness, Theorem 24) By rule induction on the given bidirectional typing. □

**Corollary 30** $\cdot \vdash e \iff A_m$ iff $\cdot \vdash e \iff A_m/ \cdot$. 

EXTENDED VERSION
8 Conclusion

There have been recent proposals to extend the adjoint approach to combining logics to dependent types. Licata and Shulman [2016], Licata et al. [2017] permit dependent types and richer connections between the logics that are combined, but certain properties such as independence are no longer fundamental, but have to be proved in each case where they apply. While they mostly stay within a sequent calculus, they also briefly introduce natural deduction. They also provide a categorical semantics. Hanukaev and Eades [2023] also permit dependent types and use the graded/algebraic approach to defining their system. However, their approach to dependency appears incompatible with control of contraction, so their adjoint structure is not nearly as general as ours. They also omit empty internal choice (and external choice altogether), which created some of the trickiest issues in our system. Neither of these proposes an algorithm for type checking or an operational semantics that would exploit the substructural and mode properties to obtain “free theorems” about well-typed programs.

We are pursuing several avenues of future work building on the results of this paper. On the foundational side, we are looking for a direct algorithm to convert an arbitrary natural deduction into a verification. On the programming side, we are considering mode polymorphism, that is, type-checking the same expression against multiple different modes to avoid code duplication. On the application side, we are considering staged computation, quotation, and metaprogramming, decomposing the usual type □A or its contextual analogue along the lines of Example 8.

References


Philip Wadler. Linear types can change the world. In IFIP TC, volume 2, pages 347–359, 1990.


\[
\begin{align*}
\Gamma \vdash s \Rightarrow A_m / \Xi & \quad \vdash e \Leftarrow A_m / \Xi & \quad \vdash x : A_m \in \Gamma \\
\Gamma \vdash s \Leftarrow A_m / \Xi & \quad \vdash (e : A_m) \Rightarrow A_m / \Xi & \quad \vdash (x : A_m) \\
\Gamma, x : A_m \vdash e \Leftarrow B_m / \Xi & \quad \vdash \lambda x. e \Leftarrow A_m \Rightarrow (\Xi \setminus x : A_m) & \quad \vdash \neg \text{I} \\
\Gamma, \ell \vdash A_m \Rightarrow B_m / \Xi & \quad \vdash e \Leftarrow A_m / \Xi' & \quad \vdash \neg \text{E} \\
\Gamma \vdash \{\} \Leftarrow \& \text{m} \{\} / [\Gamma|_m] & \quad \Gamma \vdash \ell \ell \Leftarrow A^{\ell}_m / \Xi_\ell \quad (\forall \ell \in L \neq \emptyset) & \quad \& \text{I} \\
\Gamma \vdash e \Rightarrow \& \{\ell : A^{\ell}_m\}_{\ell \in L} / \Xi \quad (\ell \in L) & \quad \Gamma \vdash e. \ell \Rightarrow A^{\ell}_m / \Xi & \quad \& \text{E} \\
\Gamma \vdash \text{susp} \ e \Leftarrow \uparrow^{\ell}_k A_k / \Xi|_k & \quad \Gamma \vdash \text{force} \ s \Rightarrow A_k / \Xi & \quad \uparrow \text{E} \\
\Gamma \vdash e_1 \Leftarrow A_m / \Xi & \quad \Gamma \vdash e_2 \Leftarrow B_m / \Xi' & \quad \Gamma \vdash (e_1, e_2) \Leftarrow A_m \otimes B_m / \Xi ; \Xi' & \quad \otimes \text{I} \\
\Gamma \vdash s \Rightarrow A_m \otimes B_m / \Xi & \quad m \geq r & \quad \Gamma, x : A_m, x_2 : B_m \vdash e' \Leftarrow C_r / \Xi' & \quad \otimes \text{E} \\
\Gamma \vdash \text{match} \ s ((x_1, x_2) \Rightarrow e') \Leftarrow C_r / \Xi ; (\Xi' \setminus x_1 : A_m \setminus x_2 : B_m) & \quad \vdash \emptyset & \quad \text{I} \\
\Gamma \vdash () \Leftarrow 1_m / () & \quad \Gamma \vdash s \Rightarrow 1_m / \Xi & \quad m \geq r & \quad \Gamma \vdash e' \Leftarrow C_r / \Xi' & \quad 1 \text{E} \\
\Gamma \vdash e \Leftarrow A^{\ell}_m / \Xi \quad (\ell \in L) & \quad \Gamma \vdash s \Rightarrow \oplus \{\ell : A^{\ell}_m\}_{\ell \in L} / \Xi & \quad m \geq r & \quad \Gamma \vdash \text{match} \ s () \Rightarrow C_r / \Xi ; [\Gamma|_r] & \quad \oplus \text{E}_0 \\
\Gamma \vdash s \Rightarrow \oplus \{\ell : A^{\ell}_m\}_{\ell \in L} / \Xi & \quad m \geq r & \quad \Gamma, x : A^{\ell}_m \vdash e_\ell \Leftarrow C_r / \Xi_\ell \quad (\forall \ell \in L \neq \emptyset) & \quad \oplus \text{E} \\
\Gamma \vdash \text{match} \ s (\ell(x) \Rightarrow e_\ell)_{\ell \in L} \Leftarrow C_r / \Xi ; \cup_{\ell \in L} (\Xi_\ell \setminus x : A_{\ell}) & \quad \vdash \emptyset & \quad \text{I} \\
\Gamma \vdash e \Leftarrow A_n / \Xi & \quad \Gamma \vdash \text{down} \ e \Leftarrow \downarrow^m_m A_n / \Xi & \quad \vdash \emptyset & \quad \text{I} \\
\Gamma \vdash s \Rightarrow \downarrow^m_m A_n / \Xi & \quad m \geq r & \quad \Gamma, x : A_n \vdash e' \Leftarrow C_r / \Xi' & \quad \down \text{E} \\
\Gamma \vdash \text{match} \ s (\text{down} \ x \Rightarrow e') \Leftarrow C_r / \Xi ; (\Xi' \setminus x : A_n) & \quad \vdash \emptyset & \quad \text{I} \\
\end{align*}
\]

Figure 6: Algorithmic Typing for Natural Deduction