Avoiding Spurious Causal Dependencies via Proof Irrelevance in a Concurrent Logical Framework

Ruy Ley-Wild[1] Frank Pfenning[2]

February 11, 2007
CMU-CS-07-107

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The Concurrent Logic Framework (CLF) is a foundational type theory for encoding concurrent computations by representing resources with linearity and encapsulating the effects of concurrency in a monad. The definition of concurrent equality via commuting conversions identifies computations differing only in the order of execution of independent steps, capturing a form of true concurrency in a proof-theoretic way. However, some example encodings suffer from spurious dependencies whereby independent computations cannot be reordered because they use the same shared resource but are not causally linked, or computations are distinguished even though they only differ in the use of isomorphic objects. We address these limitations by incorporating a linear proof irrelevance modality and adopting a richer definition of equality that admits reordering computations modulo proof irrelevant terms. We present several encodings of stateful concurrent systems to demonstrate the usefulness of this extension.

[1]This author was partially supported by a Bell Labs Graduate Fellowship.
[2]This author was partially supported by NSF Grant CCR-0306313.

Keywords: linear logic, logical frameworks, proof irrelevance, true concurrency

1 Introduction

A *logical framework* is a formal system that serves as a meta-language for defining and reasoning about logics and programming languages. A framework consists of a formal *meta-logic* or *type theory* that internalizes features that are common to deductive systems and an *informal representation methodology* for encoding object languages and establishing the adequacy of encodings. Logical frameworks are useful for formalizing meta-theoretic properties of deductive systems and, moreover, mechanized implementations can provide a logic programming environment for automated proof checking and proof search.

The Logical Framework (LF) [Harper et al., 1993] is a foundational intuitionistic type theory with an intrinsic notion of bound variable, α-conversion, and substitution. A deductive system is represented in LF as an object language specified by a *signature* consisting of constants at the object and type levels. Judgments pertaining to the logic and its meta-theory are encoded by types, while computations and deductions are encoded by objects. The Concurrent Logical Framework (CLF) [Watkins et al., 2004] augments the dependent type theory of LF with selected connectives from linear logic [Girard, 1987] as type constructors and a definition of *concurrent equality*. Linearity can be used to represent resources and thus model imperative computation, while the multiplicative conjunction connective of linear logic can represent concurrent execution. The synchronous fragment (e.g., multiplicative conjunction and related connectives) of the language is encapsulated in a monad to segregate the effects of concurrency. Therefore linear hypotheses model state, linear functions model imperative computations, and monadic objects model concurrent computations.

To capture *true concurrency* [Mazurkiewicz, 1995], semantic models of concurrency typically quotient the interleaving of events by an equivalence relation that allows permuting independent events, which makes the order of independent events indistinguishable. In modern presentations of LF, objects are restricted to β-normal, η-long *canonical form* and definitional equality is a structural congruence. In CLF, definitional equality achieve a form of *true concurrency* by extending the structural congruence to admit *commuting conversions*: monadic computations are identified if they only differ in the order of execution of independent steps. Adjacent computation steps are recognized as independent if they are not causally linked, that is, if the effects of one computation step do not affect the behavior of the other, and vice versa. This notion generalizes straightforwardly to sequences of steps. In terms of the framework's language, two concurrent computations are independent if the variables bound by one monadic computation are not used by the other, and vice versa. This definition of independence is too weak for some encodings and the resulting notion of equality is deficient in two ways. First, equality distinguishes computations that only differ in the use of distinct but isomorphic objects. Second, some computations suffer from spurious synchronizations whereby independent computations cannot be reordered because they use the same shared resource, even though the order of their occurrence is not causally related.

Proof irrelevance has been suggested as a solution to these limitations [Watkins et al., 2004], and here we develop a formulation of CLF with proof irrelevance and a richer concurrent equality that can eliminate spurious synchronization. In prior work, Pfenning [2001] developed an extension of the LF type theory with an *intuitionistic proof irrelevance* modality wherein objects of the same type are equated. The computational interpretation of proof irrelevant objects in that setting is that only their existence matters, which is a static property in the absence of stateful computation. In CLF, however, it is desirable to have *linear proof irrelevant* objects that can be produced and consumed as resources with the equational condition that proof irrelevant objects of the same type are indistinguishable.

In this paper, we extend CLF to CLF$^{\triangle}$ by incorporating such a linear proof irrelevance modality and adopting a richer definition of equality that admits reordering concurrent computations modulo proof irrelevant terms. Proof irrelevant objects of the same type are identified, so computations that use different but isomorphic proof irrelevant objects are also indistinguishable. Crucially, spurious causal dependencies can be eliminated by making the pertinent shared resources proof irrelevant in the encoding. Note that CLF$^{\triangle}$ itself does not identify which actions may commute but rather provides a framework for describing varying degrees of concurrency. We refer to certain causal dependencies in CLF encodings as spurious because they are an artifact of the framework, whereas they should not be regarded as spurious in CLF$^{\triangle}$ because their occurrence can be controlled.

In §2 we present the type theory and meta-theory of CLF$^{\triangle}$. In §3 we give encodings of concurrent systems and exhibit how proof irrelevance can be used to eliminate spurious synchronizations and ignore the use of isomorphic objects. In §4 we discuss future and related work and we conclude in §5.

2 CLF$^{\triangle}$

In this section we give an overview of CLF and present the CLF$^{\triangle}$ extension with proof irrelevance and a richer definition of concurrent equality. The syntax is presented in §2.1 and the typing judgements in §2.2; linear proof irrelevance is discussed in §2.3 and concurrent equality is discussed in §2.4. The reader may refer to Watkins et al. [2002] for a detailed treatment of CLF and to López et al. [2005] for an operational semantics of proof search. We illustrate CLF$^{\triangle}$'s constructs throughout this section with examples from the AI planning domain Blocks World (*cf.* §3.2).

CLF integrates an intuitionistic dependent type theory and a complement of connectives from linear logic for the succinct representation of concurrent computations with state. Types represent the state of computation and objects represent sequences of actions in a computation; under the semantics of proof search, types may also be considered as reachability predicates. Concurrent systems can, of course, already be encoded in LF alone, but it is awkward to describe stateful computation and difficult to work with the resulting encodings.

CLF incorporates connectives from linear logic which permit the representation of state with linear hypotheses. Moreover, imperative computation is represented with linear implication ($A_2 \multimap A_1$), nondeterminism with additive conjunction ($A_1 \mathbin{\&} A_2$), and arbitrary resource consumption with the additive unit ($\top$). Following the terminology of Andreoli [1992], we refer to the above connectives as *asynchronous* because of their behavior in proof search; the *synchronous* connectives are introduced below.

Example. The Blocks World domain consists of a virtual world where stacks of blocks are manipulated with robotic arms. The CLF encoding of Blocks World models the blocks and arms with the types blk and arm, respectively. For example, a system with two blocks named a and b and one arm named h is represented by: $a{:}\mathsf{blk}, b{:}\mathsf{blk}, h{:}\mathsf{arm}$. Here, a sequence separated by commas (",") represents a multiset of *unrestricted* variables—i.e., the variables can be used an unrestricted number of times.

The dynamic properties of the objects are modeled by linear variables of the following types:

type	description
$\mathsf{on}\, a\, b$	block a is on top of block b
$\mathsf{ont}\, a$	block a is on the table
$\mathsf{clr}\, a$	block a is reachable (nothing is on top of it)
$\mathsf{free}\, h$	arm h is available
$\mathsf{holds}\, h\, a$	arm h is holding block a

Since we consider arms indistinguishable, we henceforth ignore their names: we elide unrestricted variables of type arm. In particular, we omit them from the types:

type	description
free	an arm is available
$\mathsf{holds}\, a$	an is holding block a

For example, the configuration with a stacked on top of b and an empty-handed arm is represented by the linear variables of the appropriate type: $o \hat{:} \mathsf{on}\, a\, b \hat{;} f \hat{:} \mathsf{free}$. Here, the colon with a caret ($\hat{:}$) indicates the typing of linear variables and a sequence separated by semicolons with carets ("$\hat{;}$") represents a multiset of *linear* variables—i.e., each variable must be used exactly once. We use the following mnemonics for variables: f to witness an arm is free, hx for the arm holds block x, cx for block x is clear, o for a is stacked on b, tx for block x is on the table.

Consider an operator for picking up a reachable block a that is on top of a block b. Intuitively, the operator causes a transition from an initial state where a is on top of b ($\mathsf{on}\, a\, b$), a is reachable ($\mathsf{clr}\, a$), and a robotic arm is available (free), to a final state where the robotic arm is holding a ($\mathsf{holds}\, a$) and now b is reachable ($\mathsf{clr}\, b$). Since CLF includes the *linear function* type $A_2 \multimap A_1$ that consumes a linear argument of type A_2 and produces a result of type A_1,

the operator's transition can intuitively be represented as a linear function. The synchronous connectives alone, however, cannot directly express the (multiplicative) conjunction of the initial conditions transformed into the conjunction of the final conditions. A (multiplicative) conjunction of hypotheses can nonetheless be encoded with the linear function type by using a *continuation-passing style* with an abstract answer type res. More precisely, the operator can be encoded by the constant:

$$\mathsf{up_k} : (\mathsf{holds}\, x \multimap \mathsf{clr}\, y \multimap \mathsf{res}) \multimap (\mathsf{on}\, x\, y \multimap \mathsf{clr}\, x \multimap \mathsf{free} \multimap \mathsf{res})$$

that transforms a continuation of type $\mathsf{holds}\, x \multimap \mathsf{clr}\, y \multimap \mathsf{res}$ that consumes the final state into a continuation of type $\mathsf{on}\, x\, y \multimap \mathsf{clr}\, x \multimap \mathsf{free} \multimap \mathsf{res}$ that consumes the initial state.

Consider a configuration with block a on top of b and a free arm:

$$ca \mathbin{\hat{:}} \mathsf{clr}\, a \mathbin{\hat{;}} o \mathbin{\hat{:}} \mathsf{on}\, a\, b \mathbin{\hat{;}} tb \mathbin{\hat{:}} \mathsf{ont}\, b \mathbin{\hat{;}} f \mathbin{\hat{:}} \mathsf{free}.$$

Given a continuation: $k{:}\mathsf{holds}\, a \multimap \mathsf{clr}\, b \multimap \mathsf{ont}\, b \multimap \mathsf{res}$ that expects a to be held by the arm and b to be reachable and on the table, the term:

$$\mathsf{up_k} \mathbin{\hat{}} (\hat{\lambda} ha, cb.k \mathbin{\hat{}} ha \mathbin{\hat{}} cb \mathbin{\hat{}} tb) \mathbin{\hat{}} o \mathbin{\hat{}} ca \mathbin{\hat{}} f$$

has type res thus transitioning to a state where k can proceed. Here, the infix caret ($\hat{}$) is linear application and $\hat{\lambda} x_1, \ldots, x_n.N$ denotes the iterated abstraction of linear variables $x_1, \ldots, x_n$ from the term N.

Blocks World exhibits concurrency in the presence of multiple stacks of blocks or arms. As an example where two arms can manipulate separate stacks concurrently, consider a configuration with block a_i on top of b_i ($i \in 1..2$) and two free arms:

$$ca_1 \mathbin{\hat{:}} \mathsf{clr}\, a_1 \mathbin{\hat{;}} o_1 \mathbin{\hat{:}} \mathsf{on}\, a_1\, b_1 \mathbin{\hat{;}} tb_1 \mathbin{\hat{:}} \mathsf{ont}\, b_1 \mathbin{\hat{;}} f_1 \mathbin{\hat{:}} \mathsf{free} \mathbin{\hat{;}}$$
$$ca_2 \mathbin{\hat{:}} \mathsf{clr}\, a_2 \mathbin{\hat{;}} o_2 \mathbin{\hat{:}} \mathsf{on}\, a_2\, b_2 \mathbin{\hat{;}} tb_2 \mathbin{\hat{:}} \mathsf{ont}\, b_2 \mathbin{\hat{;}} f_2 \mathbin{\hat{:}} \mathsf{free}.$$

Given a continuation:

$$k_{12}{:}\mathsf{holds}\, a_1 \multimap \mathsf{clr}\, b_1 \multimap \mathsf{ont}\, b_1 \multimap \mathsf{holds}\, a_2 \multimap \mathsf{clr}\, b_2 \multimap \mathsf{ont}\, b_2 \multimap \mathsf{res}$$

that expects each a_i to be held by an arm and each b_i on the table, both of the terms:

$$\mathsf{up_k} \mathbin{\hat{}} (\hat{\lambda} ha_1, cb_1.\mathsf{up_k} \mathbin{\hat{}} (\hat{\lambda} ha_2, cb_2.k_{12} \mathbin{\hat{}} ha_1 \mathbin{\hat{}} cb_1 \mathbin{\hat{}} tb_1 \mathbin{\hat{}} ha_2 \mathbin{\hat{}} cb_2 \mathbin{\hat{}} tb_2) \mathbin{\hat{}} o_2 \mathbin{\hat{}} ca_2 \mathbin{\hat{}} f_2) \mathbin{\hat{}} o_1 \mathbin{\hat{}} ca_1 \mathbin{\hat{}} f_1$$
$$\mathsf{up_k} \mathbin{\hat{}} (\hat{\lambda} ha_2, cb_2.\mathsf{up_k} \mathbin{\hat{}} (\hat{\lambda} ha_1, cb_1.k_{12} \mathbin{\hat{}} ha_1 \mathbin{\hat{}} cb_1 \mathbin{\hat{}} tb_1 \mathbin{\hat{}} ha_2 \mathbin{\hat{}} cb_2 \mathbin{\hat{}} tb_2) \mathbin{\hat{}} o_1 \mathbin{\hat{}} ca_1 \mathbin{\hat{}} f_1) \mathbin{\hat{}} o_2 \mathbin{\hat{}} ca_2 \mathbin{\hat{}} f_2$$

pick up a_1 and a_2 in different order and transition to a state where k_{12} can proceed. Intuitively, there is concurrency between the two $\mathsf{up_k}$ applications because the order in which a_1 and a_2 are picked up shouldn't matter. Nevertheless, structural equality would distinguish the two terms.

Representing stateful transitions with the asynchronous connectives, thus, requires an awkward continuation-passing style. In the presence of nondeterminism, this amounts to an interleaving semantics because the definitional equality of the asynchronous fragment does not identify computations that only differ in the order of independent steps. It is possible to define a judgment in the object language that equates traces modulo ordering of independent steps, but since this is a common feature of concurrent systems it is preferable to internalize this notion of equality in the meta-language. To this end, the framework also includes synchronous connectives from linear logic (multiplicative unit 1 and conjunction $A_1 \otimes A_2$, existential $\exists x{:}A.S$, unrestricted modality $!A$) and a definition of concurrent equality, which obviate the continuation-passing style representation and admit an intrinsic form of true concurrency semantics.

Expressions are similar to Mazurkiewicz traces [Mazurkiewicz, 1977] as the equivalence class of action sequences modulo an independence relation, where actions correspond to monadic objects and concurrent equality generates independence. Since commutativity of actions relies on variable dependence, $\text{CLF}^{\triangle}$ is also comparable to Pratt's pomsets [Pratt, 1984] where the dependence graph of the term language induces the partial ordering, and to Winskel's event structures [Winskel, 1980] where the causality relation is variable dependence and resource availability determines incompatibility between events.

The notion of a β-normal, η-long *canonical form* is central to the LF family of frameworks. Proving adequacy of an encoding requires demonstrating a bijective correspondence between the language of the deductive system and the canonical forms of its LF encoding. The introduction of synchronous connectives threatens the uniqueness of canonical forms at a given type because the effects of concurrency cannot be identified. CLF employs a monad [Moggi, 1989], written $\{S\}$, to encapsulate synchronous propositions S of the language and thus contain the effects of concurrency. Each monadic object corresponds to a step in a concurrent computation and a sequence of let-bindings corresponds to concurrent execution.

Example. The final state of the operator for picking up a block can be expressed directly using CLF's multiplicative conjunction type $A_1 \otimes A_2$. Moreover, the *monad* type $\{S\}$ is used to encapsulate the effectful computation of performing the transition. Therefore, the operator can be encoded by the constant:

$$\mathsf{up} : \mathsf{on}\, x\, y \multimap \mathsf{clr}\, x \multimap \mathsf{free} \multimap \{\mathsf{holds}\, x \otimes \mathsf{clr}\, y\}.$$

The function is linear because resources (the arguments) are consumed, and the result is monadic because it is an effectful concurrent computation: the arm is occupied holding x and simultaneously y is reachable.

Returning to the previous example where block a is on top b and there's a free arm, the term:

$$\mathsf{up}^{\wedge} o^{\wedge} ca^{\wedge} f$$

produces the desired state transition without the need of a continuation-passing style.

CLF extends syntactic equality to *concurrent equality* by admitting the reordering of independent monadic computations. A sequence of monadic let-bindings is considered independent if their reordering does not affect the structure of dependencies on bound variables. This notion of independence is too stringent because independent computations may use a shared resource, which causes a *spurious dependence* between otherwise causally unrelated computations. Moreover, computations that only differ in the use of distinct but isomorphic objects are also distinguished by CLF's equality. The type theory of CLF$^{\triangle}$ extends CLF with a linear proof irrelevance modality ($\triangle$) in the synchronous fragment and associated connectives for producing and consuming objects of proof irrelevant type. Equality of proof irrelevant terms of the same type is trivial, so adequate encodings can avoid spurious causal dependencies by using proof irrelevance where appropriate.

Example. In a CLF encoding of Blocks World, two arms independently manipulating disjoint stacks of blocks are concurrent because no resources are shared. Returning to the previous example where block a_i is on top b_i ($i \in 1..2$) and there are two free arms, suppose we have a term M that expects each a_i to be held by an arm and each b_i on the table. Then both of the terms (using let–binding notation):

$$\begin{array}{ll}
\mathsf{let}\ \{ha_1 \otimes cb_1\} = \mathsf{up}^{\wedge}o_1{}^{\wedge}ca_1{}^{\triangle}f_1 & \mathsf{let}\ \{ha_2 \otimes cb_2\} = \mathsf{up}^{\wedge}o_2{}^{\wedge}ca_2{}^{\triangle}f_2 \\
\quad \{ha_2 \otimes cb_2\} = \mathsf{up}^{\wedge}o_2{}^{\wedge}ca_2{}^{\triangle}f_2 & \quad \{ha_1 \otimes cb_1\} = \mathsf{up}^{\wedge}o_1{}^{\wedge}ca_1{}^{\triangle}f_1 \\
\mathsf{in}\ \ M & \mathsf{in}\ \ M
\end{array}$$

perform the transition to a state where M can proceed. Since reordering the bindings doesn't affect the dependencies on bound variables, CLF's concurrent equality allows the two bindings to commute and thus the two terms are equated.

Unfortunately, the CLF encoding suffers unwarranted synchronization when independent operations (e.g., moving different stacks) involve some shared resource (e.g., a single arm). For example, a single arm manipulating two blocks is shared and causes a spurious synchronization because CLF does not permit reordering the actions, even if they are on separate stacks and thus causally unrelated. Assume an additional operator for releasing a block on the table:

$$\mathsf{dnt} : \mathsf{holds}\, x \multimap \{\mathsf{ont}\, x \otimes \mathsf{clr}\, x \otimes \mathsf{free}\}$$

that transitions from a state where an arm holds a block x ($\mathsf{holds}\, x$) to a state where x is on the table ($\mathsf{ont}\, x$) and reachable ($\mathsf{clr}\, x$) and the arm is available (free). Consider a configuration with two stacks and a single arm: block a_i is reachable ($ca_i \hat{:} \mathsf{clr}\, a_i$) and on top of b_i ($o_i \hat{:} \mathsf{on}\, a_i\, b_i$) ($i \in 1..2$), and the arm is free ($f \hat{:} \mathsf{free}$). Then the terms:

$$\begin{array}{ll}
\mathsf{let}\ \{ha_1 \otimes cb_1\} = \mathsf{up}^{\wedge}o_1{}^{\wedge}ca_1{}^{\wedge}f & \mathsf{let}\ \{ha_2 \otimes cb_2\} = \mathsf{up}^{\wedge}o_2{}^{\wedge}ca_2{}^{\wedge}f \\
\quad \{ta_1 \otimes ca_1 \otimes f'\} = \mathsf{dnt}^{\wedge}ha_1 & \quad \{ta_2 \otimes ca_2 \otimes f'\} = \mathsf{dnt}^{\wedge}ha_2 \\
\quad \{ha_2 \otimes cb_2\} = \mathsf{up}^{\wedge}o_2{}^{\wedge}ca_2{}^{\wedge}f' & \quad \{ha_1 \otimes cb_1\} = \mathsf{up}^{\wedge}o_1{}^{\wedge}ca_1{}^{\wedge}f' \\
\quad \{ta_2 \otimes ca_2 \otimes f''\} = \mathsf{dnt}^{\wedge}ha_2 & \quad \{ta_1 \otimes ca_1 \otimes f''\} = \mathsf{dnt}^{\wedge}ha_1 \\
\mathsf{in}\ \ \ldots & \mathsf{in}\ \ \ldots
\end{array}$$

$$
\begin{array}{rll}
K & ::= \mathsf{type} \mid \Pi x{:}A.K & \text{Kinds} \\
A & ::= A_2 \multimap A_1 \mid A_2 \overset{\triangle}{\multimap} A_1 \mid \Pi x{:}A_2.A_1 & \text{Asynchronous types} \\
 & \quad \mid A_1 \mathbin{\&} A_2 \mid \top \mid \{S\} \mid P & \\
P & ::= a \mid P\, N & \text{Atomic type constructors} \\
S & ::= S_1 \otimes S_2 \mid 1 \mid \exists x{:}A.S & \text{Synchronous types} \\
 & \quad \mid A \mid\, !A \mid \triangle A & \\
N & ::= \hat{\lambda} x.N \mid \overset{\triangle}{\lambda} x.N \mid \lambda x.N & \text{Normal objects} \\
 & \quad \mid \langle N_1, N_2 \rangle \mid \langle\rangle \mid \{E\} \mid R & \\
R & ::= c \mid x \mid R^{\wedge} N \mid R^{\triangle} N \mid R\, N & \text{Atomic objects} \\
 & \quad \mid \pi_1 R \mid \pi_2 R & \\
E & ::= \mathsf{let}\ B\ \mathsf{in}\ M & \text{Expressions} \\
B & ::= \cdot \mid \{p\} = R, B & \text{Bindings} \\
M & ::= M_1 \otimes M_2 \mid 1 \mid [N, M] & \text{Monadic objects} \\
 & \quad \mid N \mid\, !N \mid \triangle N & \\
p & ::= p_1 \otimes p_2 \mid 1 \mid [x, p] \mid x \mid\, !x \mid \triangle x & \text{Patterns} \\
\Psi & ::= \cdot \mid p \overset{\wedge}{:} S, \Psi & \text{Pattern contexts} \\
\Gamma & ::= \cdot \mid \Gamma, x{:}A & \text{Unrestricted contexts} \\
\Delta & ::= \cdot \mid \Delta, x \overset{\wedge}{:} A & \text{Linear contexts} \\
\Omega & ::= \cdot \mid \Omega, x \overset{\triangle}{:} A & \text{Linear irrelevant contexts} \\
\Sigma & ::= \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A & \text{Signatures}
\end{array}
$$

Figure 1: Syntax of CLF$^{\triangle}$

encode the two possible sequences for picking up each a_i and placing it on the table—moving a_1 then a_2, or vice versa. CLF's notion of concurrent equality does not allow the terms to be equated, however, because there is a spurious causal dependency between the second binding—which releases the arm f'—and third binding—which consumes the free arm f'. In §3.2, we return to this example to show how linear proof irrelevance and a more general notion of concurrent equality allows these operations to commute.

2.1 Syntax

The syntax of CLF$^{\triangle}$ is given in Figure 1. The types and objects of CLF$^{\triangle}$ are stratified to respect the segregation of the synchronous fragment from the dependent and asynchronous fragment. Synchronous types S classify monadic objects M, which represent suspended concurrent computations. Dependent and asynchronous types A classify ($\beta\eta$-)normal objects N and atomic objects

R. The definition of binding B generalizes the standard presentation of nested bindings by explicitly collecting all bindings in a single let-block, this is done to simplify the definition of concurrent equality. An expression E of the form $\mathsf{let}\ \{p\} = R, B\ \mathsf{in}\ M$ should be read as the computation of R that binds the variables in p, which are then in scope for the computation of $\mathsf{let}\ B\ \mathsf{in}\ M$. We abbreviate $\Pi x{:}A_2.K$ as $A_2 \to K$ and $\Pi x{:}A_2.A$ as $A_2 \to A$, respectively, when x is not free in K and A. We only extend contexts with fresh variables that do not already appear in the context in order to avoid an ambiguous reading of the context.

The terms containing $\triangle$ correspond to the new language constructs used to support proof irrelevance. The synchronous type $\triangle A$ is the type of linear proof irrelevant objects introduced by the modal operator $\triangle N$ in the monad and pattern-matched by $\triangle x$. The linear irrelevant implication $A_2 \overset{\triangle}{\multimap} A_1$ classifies a linear function that consumes a linear irrelevant object of type A_2 and produces an object of type A_1, this type is introduced by the abstraction $\overset{\triangle}{\lambda} x.N$ over a linear irrelevant variable and eliminated by the application $R^{\triangle}N$. In addition to the unrestricted and linear contexts, CLF$^{\triangle}$ includes a context Ω for linear irrelevant variables.

Example. In our CLF$^{\triangle}$ encoding of Blocks World, the availability of a single arm has proof irrelevant type $\triangle\mathsf{free}$ to let the arm be a shared resource without incurring spurious synchronization. An operator for self-testing the arm would have type $\mathsf{free} \overset{\triangle}{\multimap} \{\triangle\mathsf{free}\}$ since the arm is available before and after the action.

2.2 Judgments

The type theory of CLF is carefully designed to make adequacy proofs expose the concurrent nature of computation. This is realized in the syntax by avoiding mutual dependence between objects and types, and in the typing rules by restricting objects to have canonical β-normal and η-long form[1]. The reader may refer to Watkins et al. [2002] for a detailed treatment of the meta-theory of CLF.

In the absence of $\beta\eta$-conversion, equality can be defined as α-equivalence augmented with concurrent equality. In classical formulations of LF and its variants, definitional equality and typing were mutually-defined. In particular, the following typing rule depends on equality:

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A = B : \mathsf{type}}{\Gamma \vdash M : B}$$

while equality for types $\Gamma \vdash A = B : \mathsf{type}$ and objects $\Gamma \vdash M = N : A$ is defined for well-kinded types and well-typed objects. In CLF, the judgement

[1]Throughout this paper, we use *canonical* to refer to terms that are β-normal and η-long. Because of commuting conversions and proof irrelevance as discussed below, canonical forms may be equal without being syntactically identical, contrary to some uses of this term in the literature.

$\boxed{A_1 =_{\mathrm{A}} A_2}$ $\boxed{P_1 =_{\mathrm{P}} P_2}$ $\boxed{S_1 =_{\mathrm{S}} S_2}$ [Type equality]

$$\frac{A =_{\mathrm{A}} A'}{(\triangle A) =_{\mathrm{A}} (\triangle A')} \qquad \frac{A_2 =_{\mathrm{A}} A_2' \qquad A_1 =_{\mathrm{A}} A_1'}{(A_2 \overset{\triangle}{\multimap} A_1) =_{\mathrm{A}} (A_2' \overset{\triangle}{\multimap} A_1')} \qquad \text{(All other congruences.)}$$

$\boxed{N_1 =_{\mathrm{N}} N_2}$ $\boxed{R_1 =_{\mathrm{R}} R_2}$ $\boxed{M_1 =_{\mathrm{M}} M_2}$ $\boxed{p_1 =_{\mathrm{p}} p_2}$ [Object and pattern equality]

$$\frac{N =_{\mathrm{N}} N'}{(\overset{\triangle}{\lambda} x.N) =_{\mathrm{N}} (\overset{\triangle}{\lambda} x.N')} \qquad \frac{R =_{\mathrm{R}} R'}{(R^{\triangle} N) =_{\mathrm{R}} (R'^{\triangle} N')} \qquad \frac{}{(\triangle N) =_{\mathrm{M}} (\triangle N')}$$

$$\frac{}{(\triangle x) =_{\mathrm{p}} (\triangle x)} \qquad \text{(All other congruences.)}$$

$\boxed{E_1 =_{\mathrm{E}} E_2}$ [Expression equality]

$$\frac{\sigma \in \mathrm{Sym}(|B|) \qquad \sigma B =_{\mathrm{B}} B' \qquad M =_{\mathrm{M}} M'}{(\mathsf{let}\ B\ \mathsf{in}\ M) =_{\mathrm{E}} (\mathsf{let}\ B'\ \mathsf{in}\ M')}\ {}^{a}$$

$\boxed{B_1 =_{\mathrm{B}} B_2}$ [Binding equality]

$$\frac{}{\cdot =_{\mathrm{B}} \cdot} \qquad \frac{p =_{\mathrm{p}} p' \qquad R =_{\mathrm{R}} R' \qquad B =_{\mathrm{B}} B'}{(\{p\} = R, B) =_{\mathrm{B}} (\{p'\} = R', B')}$$

Figure 2: CLF$^{\triangle}$ equality rules

[a] cf. side condition in §2.4

$C =_{\mathrm{C}} C'$, for each syntactic category C, denotes the equality of the terms C and C' without referring to their type or the ambient context. This eliminates the mutual dependency between equality and typing (*cf.* rule $\Rightarrow_{\mathrm{R}}\Leftarrow_{\mathrm{N}}$ in Figure 6), but imposes the caveat that *a derivation of equality only makes sense when applied to well-typed objects of the same type in the same hypothetical context.*

In CLF, equality is defined as a structural congruence for each syntactic category: two terms are equal if their subterms are equal. In addition, expressions are equal modulo the reordering of independent steps (*cf.* concurrent equality in §2.4). The definition of CLF$^{\triangle}$ equality is given in Figure 2 with explicit rules for the proof irrelevance connectives and their proof constructors, concurrent equality, and bindings. Objects in irrelevant position (N in $R^{\triangle}N$ and $\triangle N$) are *always* equal, but equality of irrelevant types ($\triangle A$ and $A_2 \overset{\triangle}{\multimap} A_1$) is structural. Concurrent equality for expressions allows reordering independent steps, subject to a side condition on variables in the bindings (*cf.* § 2.4) that allows proof irrelevant variables to be reordered. Bindings for irrelevant objects (x in $\overset{\triangle}{\lambda} x.N$ and $\triangle x$) are *not* irrelevant occurrence positions and therefore they are only equated when the variables match.

Since concurrent equality implies structural equality of expressions and equality of irrelevant objects is trivial, structural equality need not be explicitly given for expressions or irrelevant objects. Thus equality can be characterized as the least congruence containing α-equivalence, concurrent equality, and trivial equality on irrelevant objects.

Alternatively, we can define a homomorphic translation $(\cdot)^*$ of CLF$^{\triangle}$ terms that replaces objects in irrelevant position with a distinguished constant $*$ but preserves the remaining structure of terms. The cases for irrelevant connectives and constructors are given below, the cases for the remaining terms are defined by congruence.

$$\begin{array}{cc} (\triangle N)^* \equiv \triangle * & (\triangle A)^* \equiv \triangle (A^*) \\ (\overset{\triangle}{\lambda} x.N)^* \equiv \overset{\triangle}{\lambda} x.(N^*) & (A_2 \overset{\triangle}{\multimap} A_1)^* \equiv (A_2^*) \overset{\triangle}{\multimap} (A_1^*) \\ (R^{\triangle} N)^* \equiv (R^*)^{\triangle} * & (\triangle x)^* \equiv \triangle x \end{array}$$

It follows that equality is the least congruence containing α-equivalence and concurrent equality of the translated terms.

Example. The availability of each arm is represented by a corresponding object of type $\triangle\mathsf{free}$. Two available arms can be used indistinguishably because their respective $\triangle\mathsf{free}$ witnesses must occur in irrelevant position. Consider an encoding with a self-testing operation test, a suspension operation susp, an operation $\mathsf{turnoff}$ for shutting down all unused arms:

$$\mathsf{test}{:}\mathsf{free} \overset{\triangle}{\multimap} \{\triangle\mathsf{free}\} \qquad \mathsf{susp}{:}\mathsf{free} \overset{\triangle}{\multimap} \{\triangle\mathsf{down}\} \qquad \mathsf{turnoff}{:}\top \multimap \mathsf{off}$$

where the type down denotes an inactive arm and the type off denotes shutting down any number of unused arms. Assume there are two available arms $x, y \overset{\triangle}{:} \mathsf{free}$. Note that by the typing rule $\top\mathbf{I}$ (Figure 6), $\mathsf{turnoff}$ acts as a sink for any number of unused linear resources. The expressions:

$$\mathsf{let}\ \{\triangle w\} = \mathsf{test}^{\triangle} x\ \mathsf{in}\ \triangle w \otimes \triangle y \quad \text{and} \quad \mathsf{let}\ \{\triangle w\} = \mathsf{test}^{\triangle} y\ \mathsf{in}\ \triangle w \otimes \triangle x$$

test different arms (x and y) and leave the other untested, but they are equal because they have the same type $\triangle\mathsf{free}\otimes\triangle\mathsf{free}$ and only differ by proof irrelevant terms: $(\mathsf{test}^{\triangle} x) =_{\mathrm{R}} (\mathsf{test}^{\triangle} y)$ and $(\triangle w \otimes \triangle y) =_{\mathrm{M}} (\triangle w \otimes \triangle x)$. Terms with different free variables may still be equal if they typecheck under the same context: $\triangle x \otimes (\mathsf{turnoff}^{\wedge}\langle\rangle)$ and $\triangle y \otimes (\mathsf{turnoff}^{\wedge}\langle\rangle)$ have different free variables, but they are equal because they have the same type $\triangle\mathsf{free} \otimes \mathsf{off}$ in context $x, y \overset{\triangle}{:} \mathsf{free}$ and only differ by proof irrelevant terms: $\triangle x =_{\mathrm{M}} \triangle y$. Although both:

$$\mathsf{let}\ \{\triangle y\} = \mathsf{test}^{\triangle} x\ \mathsf{in}\ \mathsf{turnoff}^{\wedge}\langle\rangle \quad \text{and} \quad \mathsf{let}\ \{\triangle y\} = \mathsf{susp}^{\triangle} x\ \mathsf{in}\ \mathsf{turnoff}^{\wedge}\langle\rangle$$

have type off and the binding patterns coincide, but the expressions are not equated because the bound objects and their types differ:

$$\mathsf{test}^{\triangle} x \neq_{\mathrm{R}} \mathsf{susp}^{\triangle} x \qquad \{\triangle\mathsf{free}\} \neq_{\mathrm{A}} \{\triangle\mathsf{down}\}.$$

Observe that in this encoding $\mathsf{test}^{\triangle} x$ and $\mathsf{test}^{\triangle} y$ are equated because proof irrelevance does not distinguish the arms. Token synchronization (*cf.* write synchronization in §3.3) is an encoding methodology that can be used to enforce selective synchronization of individual operators or between operators.

In this presentation of CLF, the typing rules (*cf.* rules $\Pi\mathbf{E}$ in Figure 6 and $\exists\mathbf{I}$ in Figure 7) employ *hereditary substitutions* to perform a standard substitution and additionally preserve ($\beta\eta$-)canonical forms. Hereditary substitutions are realized by an *instantiation* operator $[N_0/x]^{\mathrm{C}}_{A_0} C \equiv C'$, for each syntactic category C, which denotes the capture-avoiding substitution of object N_0 of type A_0 for the free occurrences of x in C, which produces result C' with the necessary β-reductions and η-expansions to maintain canonical form. If both C and N_0 are canonical then C' is guaranteed to be canonical, but if either C or N_0 fail to be canonical then C' may not be canonical either.

Instantiation additionally relies on the following auxiliary *reduction* and *matching* operators. When the head of R is x, the *reduction* operator $\mathsf{reduce}_{A_0}(x.R, N_0) \equiv N$ computes the canonical form N of the substitution of object N_0 of type A_0 for x in R, and the *type reduction* operator $\mathsf{treduce}_{A_0}(x.R) \equiv A$ computes the associated type A of N. The *matching* operators $\mathsf{match\_e}_{S_0}(p.E, E_0) \equiv E'$ and $\mathsf{match\_m}_{S_0}(p.E, M_0) \equiv E'$ compute the instantiation E' resulting from the substituend-directed hereditary substitution in E resulting from binding E_0 or M_0 of type S_0 to pattern p.

Furthermore, we define an *expansion* operator denoted $\mathsf{expand}_A(R) \equiv N$, which produces the type-directed η-long form N of the atomic object R of type A, and an auxiliary *pattern expansion* operator $\mathsf{pexpand}_S(p) \equiv M$ for generating the canonical expansion of patterns.

The definition of the hereditary substitution operators for the linear proof irrelevance connectives and constructors of CLF$^{\triangle}$ are given in Figure 3, the definition for the rest of the language is the same as in CLF. The cases of instantiation, matching, and pattern expansion for the proof irrelevance modality are straightforward recursions and mimic the corresponding cases of the unrestricted modality !. The cases of instantiation and expansion for irrelevant implication, instantiation, type reduction, and reduction for irrelevant application, and instantiation for irrelevant abstraction are similar to the respective cases of the linear versions.

The typing judgments of CLF$^{\triangle}$ (Figure 4) are bidirectional and syntax-directed in the objects. Typing derivations are parameterized in a CLF$^{\triangle}$ signature Σ of constants. Each judgment has an unrestricted context Γ of variables that may be used any number of times, a linear context Δ of variables that must be used exactly once, and a linear irrelevant context Ω of variables that must be used exactly once in a proof irrelevant object. The full typing rules are shown in Figures 6 and 7. The irrelevant context, like the linear context, is used additively or multiplicatively in the typing rules according to the connective under consideration. The typing rules for the linear proof irrelevant modality, implication, abstraction, and application constructors are discussed in §2.3. Observe that objects may only refer to variables in the unrestricted context (rule $x!$) and

$$
\begin{aligned}
[N_0/x]^{\mathrm{A}}_{A_0}(A_2 \overset{\triangle}{\multimap} A_1) &\equiv ([N_0/x]^{\mathrm{A}}_{A_0} A_2) \overset{\triangle}{\multimap} ([N_0/x]^{\mathrm{A}}_{A_0} A_1) \\
[N_0/x]^{\mathrm{S}}_{A_0}(\triangle A) &\equiv \triangle([N_0/x]^{\mathrm{A}}_{A_0} A) \\
[N_0/x]^{\mathrm{N}}_{A_0}(\overset{\triangle}{\lambda} y.N) &\equiv \overset{\triangle}{\lambda} y.([N_0/x]^{\mathrm{N}}_{A_0} N) \\
&\quad \text{where } y \neq x, y \notin \mathrm{FV}(N_0) \\
[N_0/x]^{\mathrm{R}}_{A_0}(R^{\triangle} N) &\equiv ([N_0/x]^{\mathrm{R}}_{A_0} R)^{\triangle}([N_0/x]^{\mathrm{N}}_{A_0} N) \\
[N_0/x]^{\mathrm{M}}_{A_0}(\triangle N) &\equiv \triangle([N_0/x]^{\mathrm{N}}_{A_0} N) \\
\mathsf{treduce}_{A_0}(x.R^{\triangle} N) &\equiv A_1 \\
&\quad \text{where } \mathsf{treduce}_{A_0}(x.R) \equiv A_2 \overset{\triangle}{\multimap} A_1 \\
\mathsf{reduce}_{A_0}(x.R^{\triangle} N) &\equiv [[N_0/x]^{\mathrm{N}}_{A_0} N/y]^{\mathrm{N}}_{A_2} N' \\
&\quad \text{where } \mathsf{treduce}_{A_0}(x.R, N_0) \equiv A_2 \overset{\triangle}{\multimap} A_1, \\
&\quad \mathsf{reduce}_{A_0}(x.R, N_0) \equiv \overset{\triangle}{\lambda} y.N', \\
&\quad y \neq x, y \notin \mathrm{FV}([N_0/x]^{\mathrm{N}}_{A_0} N) \\
\mathsf{match\_m}_{\triangle A_0}(\triangle x.E, \triangle N_0) &\equiv \triangle([N_0/x]^{\mathrm{E}}_{A_0} E) \\
\mathsf{expand}_{A_2 \overset{\triangle}{\multimap} A_1}(R) &\equiv \overset{\triangle}{\lambda} x.\mathsf{expand}_{A_1}(R^{\triangle}\mathsf{expand}_{A_2}(x)) \\
&\quad \text{where } x \notin \mathrm{FV}(R) \\
\mathsf{pexpand}_{\triangle A}(\triangle x) &\equiv \triangle \mathsf{expand}_A(x)
\end{aligned}
$$

Figure 3: Hereditary substitution operators for proof irrelevant connectives

$\Gamma; \Delta; \Omega \vdash_\Sigma N \Leftarrow_{\mathrm{N}} A$	Normal object checking
$\Gamma; \Delta; \Omega \vdash_\Sigma R \Rightarrow_{\mathrm{R}} A$	Atomic object inference
$\Gamma; \Delta; \Omega \vdash_\Sigma E \Leftarrow_{\mathrm{E}} S$	Expression checking
$\Gamma; \Delta; \Omega \vdash_\Sigma B \Rightarrow_{\mathrm{B}} \Gamma'; \Delta'; \Omega'$	Binding typing
$\Gamma; \Delta; \Omega; \Psi \vdash_\Sigma E \leftarrow_{\mathrm{p}} S$	Pattern expansion
$\Gamma; \Delta; \Omega \vdash_\Sigma M \Leftarrow_{\mathrm{M}} S$	Monadic object checking

Figure 4: Typing judgments of CLF$^{\triangle}$

linear context (rule x). Variables in the irrelevant context may only be used if they appear in proof irrelevant position (rule $\triangle\mathbf{I}$), which requires defining mutually inverse translations of context promotion $\Omega^{\oplus} \equiv \Delta$ and context demotion $\Delta^{\ominus} \equiv \Omega$ between irrelevant and linear contexts (Figure 5).

$$\cdot^{\oplus} \equiv \cdot \qquad\qquad \cdot^{\ominus} \equiv \cdot$$

$$(\Omega, x \overset{\triangle}{:} A)^{\oplus} \equiv \Omega^{\oplus}, x \overset{\wedge}{:} A \qquad\qquad (\Delta, x \overset{\wedge}{:} A)^{\ominus} \equiv \Delta^{\ominus}, x \overset{\triangle}{:} A$$

Figure 5: CLF$^{\triangle}$ context promotion and demotion

$\boxed{\Gamma; \Delta; \Omega \vdash_{\Sigma} N \Leftarrow_{\mathrm{N}} A}$ [Normal object checking]

$$\frac{\Gamma; \Delta, x \overset{\wedge}{:} A_2; \Omega \vdash N \Leftarrow_{\mathrm{N}} A_1}{\Gamma; \Delta; \Omega \vdash \hat{\lambda} x.N \Leftarrow_{\mathrm{N}} A_2 \multimap A_1} \multimap \mathbf{I} \qquad \frac{\Gamma; \Delta; \Omega, x \overset{\triangle}{:} A_2 \vdash N \Leftarrow_{\mathrm{N}} A_1}{\Gamma; \Delta; \Omega \vdash \overset{\triangle}{\lambda} x.N \Leftarrow_{\mathrm{N}} A_2 \overset{\triangle}{\multimap} A_1} \overset{\triangle}{\multimap} \mathbf{I}$$

$$\frac{\Gamma, x{:}A_2; \Delta; \Omega \vdash N \Leftarrow_{\mathrm{N}} A_1}{\Gamma; \Delta; \Omega \vdash \lambda x.N \Leftarrow_{\mathrm{N}} \Pi x{:}A_2.A_1} \Pi\mathbf{I}$$

$$\frac{\Gamma; \Delta; \Omega \vdash N_1 \Leftarrow_{\mathrm{N}} A_1 \qquad \Gamma; \Delta; \Omega \vdash N_2 \Leftarrow_{\mathrm{N}} A_2}{\Gamma; \Delta; \Omega \vdash \langle N_1, N_2 \rangle \Leftarrow_{\mathrm{N}} A_1 \,\&\, A_2} \,\&\, \mathbf{I} \qquad \frac{}{\Gamma; \Delta; \Omega \vdash \langle\rangle \Leftarrow_{\mathrm{N}} \top} \top\mathbf{I}$$

$$\frac{\Gamma; \Delta; \Omega \vdash E \Leftarrow_{\mathrm{E}} S}{\Gamma; \Delta; \Omega \vdash \{E\} \Leftarrow_{\mathrm{N}} \{S\}} \{\}\mathbf{I} \qquad \frac{\Gamma; \Delta; \Omega \vdash R \Rightarrow_{\mathrm{R}} P' \qquad P' =_{\mathrm{P}} P}{\Gamma; \Delta; \Omega \vdash R \Leftarrow_{\mathrm{N}} P} \Rightarrow_{\mathrm{R}} \Leftarrow_{\mathrm{N}}$$

$\boxed{\Gamma; \Delta; \Omega \vdash_{\Sigma} R \Rightarrow_{\mathrm{R}} A}$ [Atomic object inference]

$$\frac{}{\Gamma; \cdot; \cdot \vdash c \Rightarrow_{\mathrm{R}} \Sigma(c)} \, c \qquad \frac{}{\Gamma; x \overset{\wedge}{:} A; \cdot \vdash x \Rightarrow_{\mathrm{R}} A} \, x \qquad \frac{}{\Gamma; \cdot; \cdot \vdash x \Rightarrow_{\mathrm{R}} \Gamma(x)} \, x!$$

$$\frac{\Gamma; \Delta_1; \Omega_1 \vdash R \Rightarrow_{\mathrm{R}} A_2 \multimap A_1 \qquad \Gamma; \Delta_2; \Omega_2 \vdash N \Leftarrow_{\mathrm{N}} A_2}{\Gamma; \Delta_1, \Delta_2; \Omega_1, \Omega_2 \vdash R \hat{} N \Rightarrow_{\mathrm{R}} A_1} \multimap \mathbf{E}$$

$$\frac{\Gamma; \Delta; \Omega_1 \vdash R \Rightarrow_{\mathrm{R}} A_2 \overset{\triangle}{\multimap} A_1 \qquad \Gamma; \Omega_2{}^{\oplus}; \cdot \vdash N \Leftarrow_{\mathrm{N}} A_2}{\Gamma; \Delta; \Omega_1, \Omega_2 \vdash R \overset{\triangle}{} N \Rightarrow_{\mathrm{R}} A_1} \overset{\triangle}{\multimap} \mathbf{E}$$

$$\frac{\Gamma; \Delta; \Omega \vdash R \Rightarrow_{\mathrm{R}} \Pi x{:}A_2.A_1 \qquad \Gamma; \cdot; \cdot \vdash N \Leftarrow_{\mathrm{N}} A_2}{\Gamma; \Delta; \Omega \vdash R\, N \Rightarrow_{\mathrm{R}} [N/x]^{\mathrm{A}}_{A_2} A_1} \Pi\mathbf{E}$$

$$\frac{\Gamma; \Delta; \Omega \vdash R \Rightarrow_{\mathrm{R}} A_1 \,\&\, A_2}{\Gamma; \Delta; \Omega \vdash \pi_1 R \Rightarrow_{\mathrm{R}} A_1} \,\&\, \mathbf{E}_1 \qquad \frac{\Gamma; \Delta; \Omega \vdash R \Rightarrow_{\mathrm{R}} A_1 \,\&\, A_2}{\Gamma; \Delta; \Omega \vdash \pi_2 R \Rightarrow_{\mathrm{R}} A_2} \,\&\, \mathbf{E}_2$$

Figure 6: CLF$^{\triangle}$ typing rules for asynchronous objects

$\boxed{\Gamma;\Delta;\Omega \vdash_\Sigma E \Leftarrow_{\mathrm{E}} S}$ [Expression checking]

$$\frac{\Gamma;\Delta;\Omega \vdash M \Leftarrow_{\mathrm{M}} S}{\Gamma;\Delta;\Omega \vdash \mathsf{let}\ \cdot\ \mathsf{in}\ M \Leftarrow_{\mathrm{E}} S}\ {\Leftarrow_{\mathrm{M}}}{\Leftarrow_{\mathrm{E}}}$$

$$\frac{\Gamma;\Delta_1;\Omega_1 \vdash R \Rightarrow_{\mathrm{R}} \{S_0\} \qquad \Gamma;\Delta_2;\Omega_2, p \hat{:} S_0 \vdash \mathsf{let}\ B\ \mathsf{in}\ M \Leftarrow_{\mathrm{M}} S}{\Gamma;\Delta_1,\Delta_2;\Omega_1,\Omega_2 \vdash \mathsf{let}\ \{p\} = R, B\ \mathsf{in}\ M \Leftarrow_{\mathrm{E}} S}\ \{\}\mathbf{E}$$

$\boxed{\Gamma;\Delta;\Omega;\Psi \vdash_\Sigma E \leftarrow_{\mathrm{p}} S}$ [Pattern expansion]

$$\frac{\Gamma;\Delta;\Omega \vdash E \Leftarrow_{\mathrm{E}} S}{\Gamma;\Delta;\Omega;\cdot \vdash E \leftarrow_{\mathrm{p}} S}\ {\Leftarrow_{\mathrm{E}}}{\leftarrow_{\mathrm{p}}} \qquad \frac{\Gamma;\Delta;\Omega;p_1 \hat{:} S_1, p_2 \hat{:} S_2, \Psi \vdash E \leftarrow_{\mathrm{p}} S}{\Gamma;\Delta;\Omega;p_1 \otimes p_2 \hat{:} S_1 \otimes S_2, \Psi \vdash E \leftarrow_{\mathrm{p}} S}\ \otimes\mathbf{L}$$

$$\frac{\Gamma;\Delta;\Omega;\Psi \vdash E \leftarrow_{\mathrm{p}} S}{\Gamma;\Delta;\Omega;1 \hat{:} 1, \Psi \vdash E \leftarrow_{\mathrm{p}} S}\ 1\mathbf{L} \qquad \frac{\Gamma, x{:}A;\Delta;\Omega;p \hat{:} S_0, \Psi \vdash E \leftarrow_{\mathrm{p}} S}{\Gamma;\Delta;\Omega;[x,p] \hat{:} \exists x{:}A.S_0, \Psi \vdash E \leftarrow_{\mathrm{p}} S}\ \exists\mathbf{L}$$

$$\frac{\Gamma;\Delta,x \hat{:} A;\Omega;\Psi \vdash E \leftarrow_{\mathrm{p}} S}{\Gamma;\Delta;\Omega;x \hat{:} A, \Psi \vdash E \leftarrow_{\mathrm{p}} S}\ A\mathbf{L} \qquad \frac{\Gamma;\Delta;\Omega,x \hat{\triangle} A;\Psi \vdash E \leftarrow_{\mathrm{p}} S}{\Gamma;\Delta;\Omega;\triangle x \hat{:} \triangle A, \Psi \vdash E \leftarrow_{\mathrm{p}} S}\ \triangle\mathbf{L}$$

$$\frac{\Gamma, x{:}A;\Delta;\Omega;\Psi \vdash E \leftarrow_{\mathrm{p}} S}{\Gamma;\Delta;\Omega;!x \hat{:} !A, \Psi \vdash E \leftarrow_{\mathrm{p}} S}\ !\mathbf{L}$$

$\boxed{\Gamma;\Delta;\Omega \vdash_\Sigma M \Leftarrow_{\mathrm{M}} S}$ [Monadic object checking]

$$\frac{\Gamma;\Delta_1;\Omega_1 \vdash M_1 \Leftarrow_{\mathrm{M}} S_1 \qquad \Gamma;\Delta_2;\Omega_2 \vdash M_2 \Leftarrow_{\mathrm{M}} S_2}{\Gamma;\Delta_1,\Delta_2;\Omega_1,\Omega_2 \vdash M_1 \otimes M_2 \Leftarrow_{\mathrm{M}} S_1 \otimes S_2}\ \otimes\mathbf{I} \qquad \frac{}{\Gamma;\cdot;\cdot \vdash 1 \Leftarrow_{\mathrm{M}} 1}\ 1\mathbf{I}$$

$$\frac{\Gamma;\cdot;\cdot \vdash N \Leftarrow_{\mathrm{N}} A \qquad \Gamma;\Delta;\Omega \vdash M \Leftarrow_{\mathrm{M}} [N/x]^{\mathrm{S}}_A S}{\Gamma;\Delta;\Omega \vdash [N,M] \Leftarrow_{\mathrm{M}} \exists x{:}A.S}\ \exists\mathbf{I} \qquad \frac{\Gamma;\Delta;\Omega \vdash N \Leftarrow_{\mathrm{N}} A}{\Gamma;\Delta;\Omega \vdash N \Leftarrow_{\mathrm{M}} A}\ A\mathbf{I}$$

$$\frac{\Gamma;\Omega^{\oplus};\cdot \vdash N \Leftarrow_{\mathrm{N}} A}{\Gamma;\cdot;\Omega \vdash \triangle N \Leftarrow_{\mathrm{M}} \triangle A}\ \triangle\mathbf{I} \qquad \frac{\Gamma;\cdot;\cdot \vdash N \Leftarrow_{\mathrm{N}} A}{\Gamma;\cdot;\cdot \vdash !N \Leftarrow_{\mathrm{M}} !A}\ !\mathbf{I}$$

Figure 7: CLF$^{\triangle}$ typing rules for synchronous objects

2.3 Linear Proof Irrelevance

The type theory developed by Pfenning [2001] extends LF with an *intuitionistic* proof irrelevance modality. That type theory does not consider linearity, so the LF context is used for both (unrestricted) normal and proof irrelevant variables. The hypothetical judgments permit typing objects both at a normal type or at an irrelevant type. An object has an irrelevant type $\triangle A$ if it has the same underlying type A when all irrelevant variables in the context are promoted to be unrestricted variables. The hypothetical judgment for irrelevant types is internalized in the language with dependent function types, abstraction over irrelevantly-typed variables, and irrelevant application. However, there is no first-class modal operator for proof irrelevance and it is suggested that commuting conversions, which are absent in LF's definitional equality, would be necessary.

The linear proof irrelevance modality of CLF$^{\triangle}$ combines properties of proof irrelevance in the intuitionistic setting with support for linearity, thus permitting the representation of resources without distinguishing objects of the same irrelevant type. Instead of using the linear context for both linear and irrelevant variables, we use a separate irrelevant context to simplify the presentation. We roughly follow the intuitionistic treatment of irrelevant abstraction and application but, since CLF lacks linear dependent function types, the irrelevant function type constructor $\overset{\triangle}{\multimap}$ isn't dependent either. Furthermore, we do incorporate a proof irrelevance modal operator $\triangle$ in the monad for lifting normal objects to an irrelevant type but preserving linearity, similar to how the unrestricted modal operator ! makes objects persistent by lifting them from linear to unrestricted type.

The computational interpretation of an intuitionistic proof irrelevant type is that the underlying type is *provable*, that is, the type is inhabited by some object but the proof witness itself is unimportant. Objects with the same proof irrelevant type are equated and interchangeable, thus justifying the rule for equality of irrelevant objects in §2.2. However, the computational interpretation of linear proof irrelevance must be refined to support resource accounting, which is revealed in the typing rules. Given the concurrent nature of computation in CLF, the distinction between linear and linear irrelevant variables is that the former represents *a resource and its computational origin* (which computation step created the resource), whereas the latter only embodies the *existence of a resource.*

Following the introduction rule for linear abstraction, the introduction rule for linear irrelevant implication:

$$\frac{\Gamma; \Delta; \Omega, x \overset{\triangle}{:} A_2 \vdash_{\Sigma} N \Leftarrow_{\mathrm{N}} A_1}{\Gamma; \Delta; \Omega \vdash_{\Sigma} \overset{\triangle}{\lambda} x.N \Leftarrow_{\mathrm{N}} A_2 \overset{\triangle}{\multimap} A_1} \overset{\triangle}{\multimap}\mathbf{I}$$

typechecks an irrelevant abstraction with an irrelevant implication type if the body typechecks at the result type and the bound variable is used exactly once in irrelevant position, which is guaranteed by its presence in the irrelevant context.

The corresponding elimination rule:

$$\frac{\Gamma; \Delta; \Omega_1 \vdash_\Sigma R \Rightarrow_\mathrm{R} A_2 \stackrel{\triangle}{\multimap} A_1 \qquad \Gamma; \Omega_2{}^{\oplus}; \cdot \vdash_\Sigma N \Leftarrow_\mathrm{N} A_2}{\Gamma; \Delta; \Omega_1, \Omega_2 \vdash_\Sigma R^{\triangle} N \Rightarrow_\mathrm{R} A_1} \stackrel{\triangle}{\multimap}\mathbf{E}$$

typechecks an irrelevant application provided the irrelevant context can be partitioned such that the function typechecks with all of the linear context and part of the irrelevant context, and the argument typechecks with the rest of the irrelevant context promoted to be linear. This follows the intuitionistic case in that the irrelevant context becomes accessible to objects in irrelevant position (recall that there is no rule for variable lookup in the irrelevant context), and differs from the elimination rule for linear implication in that the linear context is inaccessible, preventing irrelevant terms from consuming (relevant) linear variables.

The left rule for the linear proof irrelevance modality:

$$\frac{\Gamma; \Delta; \Omega, x \stackrel{\triangle}{:} A; \Psi \vdash_\Sigma E \leftarrow_\mathrm{p} S}{\Gamma; \Delta; \Omega; \triangle x \hat{:} \triangle A, \Psi \vdash_\Sigma E \leftarrow_\mathrm{p} S} \triangle\mathbf{L}$$

inserts the pattern-matching variable into the irrelevant context just as the $!\mathbf{L}$ rule inserts the variable into the unrestricted context.

The corresponding introduction rule:

$$\frac{\Gamma; \Omega^{\oplus}; \cdot \vdash_\Sigma N \Leftarrow_\mathrm{N} A}{\Gamma; \cdot; \Omega \vdash_\Sigma \triangle N \Leftarrow_\mathrm{M} \triangle A} \triangle\mathbf{I}$$

typechecks an irrelevant object with irrelevant type if the linear context is empty and the object typechecks with the irrelevant context promoted to be linear. Again, the reason is that irrelevant terms may consume linear irrelevant variables but not linear variables.

2.4 Concurrent Equality

The basis for concurrent equality is the notion of a commuting conversion. In the simplest case, two let-bindings can be swapped if the reordering does not affect the dependence on bound variables. We state this equality using nested bindings to emphasize the binding structure, but subsequently use a single let-block.

$$\begin{aligned}&(\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E)\\ &\quad =_\mathrm{E} (\mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ \mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E)\end{aligned}$$

In order to ensure the independence of the concurrent computations R_1 and R_2, this rule has the following side conditions: the variables bound by p_1 and p_2 are disjoint, no variable bound by p_1 appears free in R_2, and vice versa.

In the original formulation of CLF, the definition of concurrent equality used the auxiliary notion of a *concurrent context* to generalize a sequence of let-bindings with a hole. That rule can be stated with the single let-block notation by requiring the two binding sequences to be equal up to some permutation σ, which need not be unique (this rule also appears in Figure 2):

$$\frac{\sigma \in \mathrm{Sym}(|B|) \qquad \sigma B =_{\mathrm{B}} B' \qquad M =_{\mathrm{M}} M'}{(\mathsf{let}\ B\ \mathsf{in}\ M) =_{\mathrm{E}} (\mathsf{let}\ B'\ \mathsf{in}\ M')}$$

We write $|B|$ for the number of bindings in B, $\mathrm{Sym}(n)$ for the complete permutation group on n elements, and σB for the corresponding permutation of the bindings. The side conditions generalize as follows: if B is a sequence of bindings $\{p_i\} = R_i$, then for each $1 \leq i < j \leq |B|$ with $\sigma(i) > \sigma(j)$, the variables bound by p_i and p_j must be disjoint, no variable bound by p_i appears free in R_j, and vice versa. The hereditary definition of binding equality respects the trivial equality between proof irrelevant terms, but the above side conditions prevent equating computations that use different proof irrelevant terms.

Example. Suppose M has type S in an context with only $w \overset{\triangle}{:} \mathsf{free}$, and the expressions:

$$\begin{array}{ll} \mathsf{let}\ \{\triangle y\} = s^{\triangle} x & \mathsf{let}\ \{\triangle z\} = t^{\triangle} x \\ \quad \{\triangle z\} = t^{\triangle} y\ \mathsf{in}\ M_z & \quad \{\triangle y\} = s^{\triangle} z\ \mathsf{in}\ M_y \end{array}$$

have the same type in a context with two self-test operations $s, t{:}(\mathsf{free} \overset{\triangle}{\multimap} \{\triangle\mathsf{free}\})$ and a free arm $x \overset{\triangle}{:} \mathsf{free}$, where $M_y = [y/w]^{\mathrm{M}}_{\mathsf{free}} M$, $M_z = [z/w]^{\mathrm{M}}_{\mathsf{free}} M$. Then $y, z \overset{\triangle}{:} \mathsf{free}$ in both cases and each must necessarily occur in irrelevant position in the corresponding monadic objects M_y, M_z. Since the function arguments occur in irrelevant position, the above expressions should be considered equal, but CLF's concurrent equality distinguishes them because the second condition fails: swapping the two bindings of the left expression ($\sigma = (12)$) leads to y being free in $t^{\triangle} y$.

We remedy this by using the same concurrent equality rule except that the side condition on p_i, R_j (and dually p_j, R_i) does not apply to linear irrelevant variables. The side condition of concurrent equality becomes: if B is a sequence of bindings $\{p_i\} = R_i$, then for each $1 \leq i < j \leq |B|$ with $\sigma(i) > \sigma(j)$, the variables bound by p_i and p_j must be disjoint, no *unrestricted or linear* variable bound by p_i appears free in R_j, and vice versa. Moreover, the trivial equality on irrelevant subterms permits $\sigma B =_{\mathrm{B}} B'$ to change objects occurring in proof irrelevant position. The new notion of equality admits reordering computations modulo proof irrelevant terms. In §3 we explore the consequences of this modification by considering more examples.

Example. The above expressions are equated in $\mathrm{CLF}^{\triangle}$ as they only differ by proof irrelevant terms. Since both expressions have type S in the same context and the variable disjointness conditions are satisfied. The derivation of Figure 8 witnesses the equality.

$$\sigma = (12) \in \mathrm{Sym}(2)$$

$$\dfrac{\dfrac{\dfrac{}{\triangle z =_{\mathrm{p}} \triangle z} \quad \dfrac{\dfrac{}{t =_{\mathrm{R}} t}}{t^{\triangle} y =_{\mathrm{R}} t^{\triangle} x} \quad \dfrac{\dfrac{}{\triangle y =_{\mathrm{p}} \triangle y} \quad \dfrac{\dfrac{}{s =_{\mathrm{R}} s}}{s^{\triangle} x =_{\mathrm{R}} s^{\triangle} z} \quad \dfrac{}{\cdot =_{\mathrm{B}} \cdot}}{(\{\triangle y\} = s^{\triangle} x) =_{\mathrm{B}} (\{\triangle y\} = s^{\triangle} z)}}{\sigma(\{\triangle y\} = s^{\triangle} x, \{\triangle z\} = t^{\triangle} y) =_{\mathrm{B}} (\{\triangle z\} = t^{\triangle} x, \{\triangle y\} = s^{\triangle} z)} \quad \vdots}{M_z =_{\mathrm{M}} M_y}}{(\mathsf{let}\ \{\triangle y\} = s^{\triangle} x, \{\triangle z\} = t^{\triangle} y\ \mathsf{in}\ M_z) =_{\mathrm{E}} (\mathsf{let}\ \{\triangle z\} = t^{\triangle} x, \{\triangle y\} = s^{\triangle} z\ \mathsf{in}\ M_y)}$$

Figure 8: Derivation of equality

The choice of representing expressions with single let-blocks is directly related to the concurrent equality rule. Using concurrent contexts to represent reordering is equivalent to restricting permutations to be reverse rotations of a subsequence of bindings, that is, permutations of the form $(\ell + k \ldots \ell)$ where $1 \leq \ell \leq \ell + k \leq |B|$. One difficulty with this representation is that it obscures the complexity of determining equality of expressions. Using a single permutation makes explicit that at most $(|B|)!$ possibilities need to be considered and avoids having to factor a general permutation into the restricted form. More importantly, the restricted permutations of concurrent contexts are insufficient to represent certain justifiable reorderings, especially when a spurious synchronization occurs due to use of a shared resource between otherwise independent computations. Even though permutations can be factored into the restricted form (i.e., the composition of transpositions), the side conditions may hold for the general permutation but not separately for each permutation of the factorization. Commuting two sequences of computations simultaneously is a more powerful transformation than reordering the computation steps individually, in some cases (*cf.* §3.2) only the former can be performed because a resource is held throughout each computation sequence, so reordering one computation step at a time does not yield a well-typed expression.

2.5 Meta-theory

As argued by Watkins et al. [2004], the capture-avoiding substitution of a normal object for a variable in a canonical term does not preserve canonicity in the general case, and a variable of higher type is not canonical at that type. Therefore it is necessary to prove the *identity* and *substitution* principles explicitly.

Lemma 1 (Expansion)**.** Asynchronous case: *If* $\Gamma; \Delta; \Omega \vdash R \Rightarrow_{\mathrm{R}} A$ *is derivable, then* $\Gamma; \Delta; \Omega \vdash \mathsf{expand}_A(R) \Leftarrow_{\mathrm{N}} A$ *is derivable.*

Synchronous case: *For any* Γ *and* S, $\Gamma; \cdot; \cdot; p \hat{:} S \vdash \mathsf{let}\ \cdot\ \mathsf{in}\ \mathsf{pexpand}_S(p) \leftarrow_{\mathrm{p}} S$ *is derivable for some* p.

Proof. By structural induction on the type. □

Theorem 2 (Identity)**.** Unrestricted case: *For any* Γ *and* A, $\Gamma, x{:}A; \cdot; \cdot \vdash \mathsf{expand}_A(x) \Leftarrow_{\mathrm{N}} A$ *is derivable.*

Linear case: *For any* Γ *and* A, $\Gamma; x \hat{:} A; \cdot \vdash \mathsf{expand}_A(x) \Leftarrow_{\mathrm{N}} A$ *is derivable.*

Proof. By the typing rules for variables and the Expansion lemma. □

Theorem 3 (Substitution)**.** Unrestricted case: *If* $\Gamma_L; \cdot; \cdot \vdash N_0 \Leftarrow_{\mathrm{N}} A_0$ *and* $\Gamma_L, x{:}A_0, \Gamma_R; \Delta; \Omega \vdash N \Leftarrow_{\mathrm{N}} A$ *are derivable, and* $[N_0/x]^{\mathrm{A}}_{A_0}\Gamma_R \equiv \Gamma'_R$, $[N_0/x]^{\mathrm{A}}_{A_0}\Delta \equiv \Delta'$, $[N_0/x]^{\mathrm{A}}_{A_0}\Omega \equiv \Omega'$, *and* $[N_0/x]^{\mathrm{A}}_{A_0}A \equiv A'$ *are defined, then* $[N_0/x]^{\mathrm{N}}_{A_0}N \equiv N'$ *is defined and* $\Gamma_L, \Gamma'_R; \Delta'; \Omega' \vdash N' \Leftarrow_{\mathrm{N}} A'$ *is derivable.*

Linear case: *If* $\Gamma; \Delta_0; \Omega_0 \vdash N_0 \Leftarrow_{\mathrm{N}} A_0$ *and* $\Gamma; \Delta, x \hat{:} A_0; \Omega \vdash N \Leftarrow_{\mathrm{N}} A$ *are derivable, then* $[N_0/x]^{\mathrm{N}}_{A_0}N \equiv N'$ *is defined and* $\Gamma; \Delta_0, \Delta; \Omega_0, \Omega \vdash N' \Leftarrow_{\mathrm{N}} A$ *is derivable.*

Linear irrelevant case: *If* $\Gamma; \Delta_0; \cdot \vdash N_0 \Leftarrow_{\mathrm{N}} A_0$ *and* $\Gamma; \Delta; \Omega, x \overset{\triangle}{:} A_0 \vdash N \Leftarrow_{\mathrm{N}} A$ *are derivable, then* $[N_0/x]^{\mathrm{N}}_{A_0}N \equiv N'$ *is defined and* $\Gamma; \Delta; \Omega, {\Delta_0}^{\ominus} \vdash N' \Leftarrow_{\mathrm{N}} A$ *is derivable.*

Proof. The theorem must be strengthened with analogous statements for the other syntactic categories. By structural induction on the second derivation of each case. □

The proofs closely follow the structure of the corresponding proofs for CLF given by Watkins [2003]. The cases for the connectives of CLF only need to carry the extra linear irrelevant context around. The cases for the proof irrelevance connectives rely on the bijective translation between linear and irrelevant contexts, but are otherwise straightforward. There is no identity principle for the irrelevant context because there is no variable lookup in that context.

The extensions given by CLF$^{\triangle}$ over CLF preserve the decidability and upper complexity bounds of hereditary substitutions, equality, and typing.

Theorem 4 (Decidability)**.** *Instantiation, expansion, equality, and typing are decidable.*

Proof. The hereditary substitution operators are syntax-directed and terminate on all inputs.

Decidability of equality is proved by simultaneous structural induction on the terms being equated.

Decidability of typing is proved by structural induction on the sequent being proved, appealing to the decidability of instantiation in the case of $\Pi\mathbf{E}$ and $\exists\mathbf{I}$ and of equality in the case of $\Rightarrow_{\mathrm{R}}\Leftarrow_{\mathrm{N}}$. □

Theorem 5 (Complexity)**.** *The complexity of hereditary substitutions, equality testing, and typing in CLF*$^{\triangle}$ *coincides with that of CLF.*

Proof. In CLF and CLF$^{\triangle}$, the hereditary substitution operators are syntax-directed and have complexity linear in the size of the inputs.

By simultaneous structural induction on the terms being equated, the upper complexity bound of equality testing is factorial in the size of the terms. The

fundamental difference between CLF and CLF$^{\triangle}$ is the definition of concurrent equality. However, since every permutation can be factored into the composition of transpositions, which satisfy the restricted form, the aggregate complexity of applying CLF's concurrent equality to nested let-bindings is the same as the complexity of applying CLF$^{\triangle}$'s concurrent equality once to a single let-block.

The typing rules for the proof irrelevance connectives do not introduce additional complexity as they parallel the definitions of the unrestricted modality and the linear implication connectives. The irrelevant context is used similarly to the linear context and does not affect the complexity of typing. By structural induction on the sequent under consideration, the complexity of typing in CLF and CLF$^{\triangle}$ coincide, relying on the corresponding agreement of hereditary substitutions and equality. □

3 Examples

We consider CLF$^{\triangle}$ encodings of Petri nets, a logical AI planning domain, and an imperative parallel language. Throughout, the corresponding CLF encodings can be obtained by replacing irrelevant application $^{\triangle}$ with linear application $^{\wedge}$, promoting irrelevant contexts to linear contexts, and deleting all remaining occurrences of $\triangle$. We omit Π quantifiers on free variables in constant declarations, and also leave the corresponding instantiations implicit in the examples. Where convenient, we include explicit typing annotations in binding patterns. CLF$^{\triangle}$ terms in sans-serif typeface denote constants and terms in *italic* typeface denote variables in the meta-language.

3.1 Petri Nets

The CLF encoding of Petri nets [Petri, 1962] was the motivating example for incorporating proof irrelevance into the framework. Petri nets can be represented in CLF by encoding each place p with a type constant p, each token in place p is encoded by a linear hypothesis $x \mathbin{\hat{:}} \mathsf{p}$, and each transition of tokens from places $p_1, \ldots, p_m$ to places $q_1, \ldots, q_n$ is represented as a linear function of type $\mathsf{p}_1 \multimap \cdots \multimap \mathsf{p}_m \multimap \{\mathsf{q}_1 \otimes \cdots \otimes \mathsf{q}_n\}$. The definitional equality of CLF distinguishes tokens at the same place, therefore encoding can only represent labeled Petri nets [Watkins et al., 2004]. We address this limitation by instead encoding each token with a linear irrelevant hypothesis $x \mathbin{\overset{\triangle}{:}} \mathsf{p}$ and each transition as a linear irrelevant function of type $\mathsf{p}_1 \overset{\triangle}{\multimap} \cdots \overset{\triangle}{\multimap} \mathsf{p}_m \overset{\triangle}{\multimap} \{\triangle\mathsf{q}_1 \otimes \cdots \otimes \triangle\mathsf{q}_n\}$. Therefore general Petri nets can be encoded in CLF$^{\triangle}$ without distinguishing computations that are identical except for the names of tokens.

Returning to CLF, a simple Petri net with two places p, q and a single transition of one token from p to q has CLF signature:

$$\mathsf{p}, \mathsf{q} : \mathsf{type} \qquad \mathsf{t} : \mathsf{p} \multimap \{\mathsf{q}\}$$

If the initial marking has two tokens at p, then the encoding begins with the linear hypotheses $x, y \mathbin{\hat{:}} \mathsf{p}$. There are two possible firings depending on whether

token x or y moves to q. The firings are represented by the expressions:

$$\begin{array}{ll} \mathsf{let}\ \{x' \hat{:} \mathsf{q}\} = \mathsf{t}^{\wedge} x & \mathsf{let}\ \{y' \hat{:} \mathsf{q}\} = \mathsf{t}^{\wedge} y \\ \mathsf{in}\ x' \otimes y & \mathsf{in}\ y' \otimes x \end{array}$$

The expressions have the same final state $\mathsf{q} \otimes \mathsf{p}$, but the bound objects differ ($\mathsf{t}^{\wedge} x \neq_{\mathrm{R}} \mathsf{t}^{\wedge} y$) and CLF distinguishes the expressions even though t acts on isomorphic objects.

In CLF$^{\triangle}$, the transition has type $\mathsf{t}{:}(\mathsf{p} \overset{\triangle}{\multimap} \{\triangle \mathsf{q}\})$, the initial state has the irrelevant hypotheses $x, y \overset{\triangle}{:} \mathsf{p}$, and the two firings are witnessed by the expressions:

$$\begin{array}{ll} \mathsf{let}\ \{\triangle x' \hat{:} \triangle \mathsf{q}\} = \mathsf{t}^{\triangle} x & \mathsf{let}\ \{\triangle y' \hat{:} \triangle \mathsf{q}\} = \mathsf{t}^{\triangle} y \\ \mathsf{in}\ \triangle x' \otimes \triangle y & \mathsf{in}\ \triangle y' \otimes \triangle x \end{array}$$

We can α-rename the bindings for x' and y' to the variable z':

$$\begin{array}{ll} \mathsf{let}\ \{\triangle z' \hat{:} \triangle \mathsf{q}\} = \mathsf{t}^{\triangle} x & \mathsf{let}\ \{\triangle z' \hat{:} \triangle \mathsf{q}\} = \mathsf{t}^{\triangle} y \\ \mathsf{in}\ \triangle z' \otimes \triangle y & \mathsf{in}\ \triangle z' \otimes \triangle x \end{array}$$

Now the the firings ($\mathsf{t}^{\triangle} x =_{\mathrm{R}} \mathsf{t}^{\triangle} y$) and final states ($\triangle z' \otimes \triangle y =_{\mathrm{M}} \triangle z' \otimes \triangle x$) are equal modulo irrelevant terms, and the variable bindings $\triangle z' =_{\mathrm{p}} \triangle z'$ are identical. Therefore, concurrent equality in CLF$^{\triangle}$ makes the two possible firings indistinguishable.

3.2 Blocks World

The classical AI planning domain Blocks World consists of a virtual 2D world with stacks of blocks on a table and robotic arms that can unstack, hold, or stack individual blocks. The encoding of the domain as a CLF$^{\triangle}$ signature is given in Figure 9. The type blk encodes blocks as unrestricted variables, their relative position (on and ont, the latter encoding a block on the table) and accessibility (clr) are encoded by linear variables:

$$\begin{array}{rl} \mathsf{blk} & : \mathsf{type} \\ \mathsf{on} & : \mathsf{blk} \to \mathsf{blk} \to \mathsf{type} \\ \mathsf{ont} & : \mathsf{blk} \to \mathsf{type} \\ \mathsf{clr} & : \mathsf{blk} \to \mathsf{type} \end{array}$$

The availability of the robotic arm (free) is represented by an irrelevant variable, but if it is unavailable (holds) then there is a linear variable witnessing which block it is holding:

$$\begin{array}{rl} \mathsf{free} & : \mathsf{type} \\ \mathsf{holds} & : \mathsf{blk} \to \mathsf{type} \\ \mathsf{on} & : \mathsf{blk} \to \mathsf{blk} \to \mathsf{type} \\ \mathsf{ont} & : \mathsf{blk} \to \mathsf{type} \end{array}$$

The ability to stack or unstack blocks is encoded by operators for picking up (up and upt) and putting down (dn and dnt) an unobstructed block relative to another block or the table:

$$
\begin{aligned}
\mathsf{blk} &: \mathsf{type} \\
\mathsf{on} &: \mathsf{blk} \to \mathsf{blk} \to \mathsf{type} \\
\mathsf{ont} &: \mathsf{blk} \to \mathsf{type} \\
\mathsf{clr} &: \mathsf{blk} \to \mathsf{type} \\
\mathsf{free} &: \mathsf{type} \\
\mathsf{holds} &: \mathsf{blk} \to \mathsf{type} \\
\mathsf{up} &: \mathsf{on}\, x\, y \multimap \mathsf{clr}\, x \multimap \mathsf{free} \overset{\triangle}{\multimap} \{\mathsf{holds}\, x \otimes \mathsf{clr}\, y\} \\
\mathsf{dn} &: \mathsf{holds}\, x \multimap \mathsf{clr}\, y \multimap \{\mathsf{on}\, x\, y \otimes \mathsf{clr}\, x \otimes \triangle\mathsf{free}\} \\
\mathsf{upt} &: \mathsf{ont}\, x \multimap \mathsf{clr}\, x \multimap \mathsf{free} \overset{\triangle}{\multimap} \{\mathsf{holds}\, x\} \\
\mathsf{dnt} &: \mathsf{holds}\, x \multimap \{\mathsf{ont}\, x \otimes \mathsf{clr}\, x \otimes \triangle\mathsf{free}\}
\end{aligned}
$$

Figure 9: Signature of Blocks World

$$
\begin{aligned}
\mathsf{up} &: \mathsf{on}\, x\, y \multimap \mathsf{clr}\, x \multimap \mathsf{free} \overset{\triangle}{\multimap} \{\mathsf{holds}\, x \otimes \mathsf{clr}\, y\} \\
\mathsf{dn} &: \mathsf{holds}\, x \multimap \mathsf{clr}\, y \multimap \{\mathsf{on}\, x\, y \otimes \mathsf{clr}\, x \otimes \triangle\mathsf{free}\} \\
\mathsf{upt} &: \mathsf{ont}\, x \multimap \mathsf{clr}\, x \multimap \mathsf{free} \overset{\triangle}{\multimap} \{\mathsf{holds}\, x\} \\
\mathsf{dnt} &: \mathsf{holds}\, x \multimap \{\mathsf{ont}\, x \otimes \mathsf{clr}\, x \otimes \triangle\mathsf{free}\}
\end{aligned}
$$

The table is implicitly encoded by the ont type and the operators upt and dnt.

This domain is sufficient to exhibit a spurious synchronization and justify the need for general permutations. In particular, consider a world with one robotic arm (one object $f \overset{\triangle}{:} \mathsf{free}$) and blocks a_1, b_1, a_2, b_2 with a_i stacked on b_i ($i \in 1..2$). This is represented by the contexts:

$$
\begin{aligned}
\Gamma &= a_1, b_1, a_2, b_2 {:}\mathsf{blk} \\
\Delta &= ca_1 \hat{:} \mathsf{clr}\, a_1 \hat{;} o_1 \hat{:} \mathsf{on}\, a_1\, b_1 \hat{;} tb_1 \hat{:} \mathsf{ont}\, b_1 \hat{;} ca_2 \hat{:} \mathsf{clr}\, a_2 \hat{;} o_2 \hat{:} \mathsf{on}\, a_2\, b_2 \hat{;} tb_2 \hat{:} \mathsf{ont}\, b_2 \\
\Omega &= f \overset{\triangle}{:} \mathsf{free}
\end{aligned}
$$

If we want to move each a_i to the table, we have to pick each a_i up from b_i and put it down on the table. Since the arm is unavailable while moving a block, this gives two possible interleavings: move a_1 then a_2, or move a_2 then a_1. Computationally, this is witnessed by the expressions below. We omit typing annotations from the patterns, but they can easily be inferred from the type of the operators. Suppose we are given some computation M that expects all blocks on the table, i.e., that typechecks in the context $\Gamma; \Delta'; \Omega$ where $\Delta' = ca_1 \hat{;} cb_1 \hat{;} ca_2 \hat{;} cb_2 \hat{;} ta_1 \hat{;} tb_1 \hat{;} ta_2 \hat{;} tb_2$. Let $M' = [f'/f]^{\mathrm{M}}_{\mathsf{free}} M$ and $M'' = [f''/f]^{\mathrm{M}}_{\mathsf{free}} M$. Then the following well-typed expressions represent two ways of transitioning from the configuration Δ to the configuration Δ' where M can proceed:

$$
\begin{array}{ll}
\mathsf{let}\ \{ha_1 \otimes cb_1\} = \mathsf{up} \hat{} o_1 \hat{} ca_1 \overset{\triangle}{} f & \mathsf{let}\ \{ha_2 \otimes cb_2\} = \mathsf{up} \hat{} o_2 \hat{} ca_2 \overset{\triangle}{} f \\
\quad \{ta_1 \otimes ca_1 \otimes \triangle f'\} = \mathsf{dnt} \hat{} ha_1 & \quad \{ta_2 \otimes ca_2 \otimes \triangle f''\} = \mathsf{dnt} \hat{} ha_2 \\
\quad \{ha_2 \otimes cb_2\} = \mathsf{up} \hat{} o_2 \hat{} ca_2 \overset{\triangle}{} f' & \quad \{ha_1 \otimes cb_1\} = \mathsf{up} \hat{} o_1 \hat{} ca_1 \overset{\triangle}{} f'' \\
\quad \{ta_2 \otimes ca_2 \otimes \triangle f''\} = \mathsf{dnt} \hat{} ha_2 & \quad \{ta_1 \otimes ca_1 \otimes \triangle f'\} = \mathsf{dnt} \hat{} ha_1 \\
\mathsf{in}\ M'' & \mathsf{in}\ M'
\end{array}
$$

In CLF we would have $f \hat{:} \mathsf{free}$ in the linear context. Observe that in the first expression concurrent equality prevents moving the $\mathsf{dnt}\hat{}ha_1$ computation down, because the binding structure requires the arm to be available (witnessed by $f' \hat{:} \mathsf{free}$) before picking a_2 up. Therefore there is a causal dependence between $\mathsf{dnt}\hat{}ha_1$—which makes the robotic arm available—and $\mathsf{up}\hat{}o_2\hat{}c_2\hat{}f'$—which requires the robotic arm to be available. The intermediate object f' represents the availability of the robotic arm and causes a spurious synchronization between the two operations (i.e., moving a_1, then a_2). Moreover, the restricted permutations of CLF would prevent swapping the otherwise independent computation sequences $\langle \mathsf{up}\hat{}o_1\hat{}c_1\hat{}f, \mathsf{dnt}\hat{}ha_1 \rangle$ and $\langle \mathsf{up}\hat{}o_2\hat{}c_2\hat{}f', \mathsf{dnt}\hat{}ha_2 \rangle$. Even with general permutations, the reordering would change which bound variable of type free is used in M', M'', violating the side condition of the old concurrent equality rule.

In CLF$^{\triangle}$ we would have $f', f'' \hat{:}^{\triangle} \mathsf{free}$ and each occurs in irrelevant position, in M', M'', respectively. Thus the reordering $\sigma = (13)(24)$ respects the binding structure and the expressions are equated by the new concurrent equality.

The choice of making free objects proof irrelevant is motivated by this example of spurious synchronization. The reader may question why holds or clr objects aren't proof irrelevant as well. Indeed, proof irrelevance permits different notions of equality depending on the choice of encoding. Although objects of type holds could be proof irrelevant, they always occur between associated pick-up and put-down steps, so reordering computation steps does not affect their binding. The case for clr objects is less clear-cut because they are unique (at most one clr witness may exist per block) and using proof irrelevance would not affect equality, but since they are not a source of spurious synchronization we consider our encoding suitable. In general, we discourage introducing unnecessary proof irrelevant types because they can obscure the adequacy of encodings.

An alternative encoding could provide primitives for moving a block directly onto another block or the table without the intermediate step of the arm holding the block. Then this example of spurious dependence would not arise, but coarsening the granularity is not always possible nor does it necessarily avoid spurious synchronization.

3.3 Communicating Sequential Processes

The syntax and semantics of a simple imperative language with parallel composition [Hoare, 1978] can be encoded in CLF$^{\triangle}$ using the irrelevant context to represent memory and multiplicative conjunction for parallel composition. The operational semantics of parallel composition is that the execution of the commands is interleaved.

Syntax

$$\begin{aligned}
&\mathsf{var}, \mathsf{exp}, \mathsf{cmd} : \mathsf{type} \\
&\qquad \mathsf{get} : \mathsf{var} \to \mathsf{exp} \\
&\qquad \mathsf{set} : \mathsf{var} \to \mathsf{exp} \to \mathsf{cmd} \\
&\qquad \mathsf{rst} : \mathsf{var} \to \mathsf{exp} \\
&\qquad \mathsf{par} : \mathsf{cmd} \to \mathsf{cmd} \to \mathsf{cmd}
\end{aligned}$$

State

$$\begin{aligned}
&\mathsf{contents} : \mathsf{var} \to \mathsf{exp} \to \mathsf{type} \\
&\qquad \mathsf{write} : \mathsf{var} \to \mathsf{type}
\end{aligned}$$

Continuations

$$\begin{aligned}
&\mathsf{dest} : \mathsf{type} \\
&\mathsf{return}_\mathrm{e} : \mathsf{exp} \to \mathsf{dest} \to \mathsf{type} \\
&\mathsf{return}_\mathrm{c} : \mathsf{dest} \to \mathsf{type} \\
&\mathsf{rest} : \mathsf{type} \\
&\mathsf{set}_\mathrm{k} : \mathsf{var} \to \mathsf{dest} \to \mathsf{rest} \\
&\mathsf{par}_\mathrm{k} : \mathsf{dest} \to \mathsf{rest} \\
&\mathsf{kont} : \mathsf{rest} \to \mathsf{dest} \to \mathsf{type}
\end{aligned}$$

Evaluation and Execution

$$\begin{aligned}
&\mathsf{eval} : \mathsf{exp} \to \mathsf{dest} \to \mathsf{type} \\
&\mathsf{exec} : \mathsf{cmd} \to \mathsf{dest} \to \mathsf{type} \\
&\mathsf{eget} : \mathsf{eval}\,(\mathsf{get}\,x)\,d \multimap \mathsf{contents}\,x\,v \\
&\qquad \overset{\triangle}{\multimap} \{\mathsf{return}_\mathrm{e}\,v\,d \otimes \triangle\mathsf{contents}\,x\,v\} \\
&\mathsf{xset}_1 : \mathsf{exec}\,(\mathsf{set}\,x\,e)\,d \\
&\qquad \multimap \{\exists d'{:}\mathsf{dest}.\mathsf{eval}\,e\,d' \otimes \mathsf{kont}\,(\mathsf{set}_\mathrm{k}\,x\,d')\,d\} \\
&\mathsf{xset}_2 : \mathsf{return}_\mathrm{e}\,v\,d' \multimap \mathsf{kont}\,(\mathsf{set}_\mathrm{k}\,x\,d')\,d \multimap \mathsf{write}\,x \multimap \mathsf{contents}\,x\,v' \\
&\qquad \overset{\triangle}{\multimap} \{\mathsf{return}_\mathrm{c}\,d \otimes \mathsf{write}\,x \otimes \triangle\mathsf{contents}\,x\,v\} \\
&\mathsf{xrst} : \mathsf{exec}\,(\mathsf{rst}\,x)\,d \multimap \mathsf{write}\,x \multimap \mathsf{contents}\,x\,v \\
&\qquad \overset{\triangle}{\multimap} \{\mathsf{return}_\mathrm{c}\,d \otimes \mathsf{write}\,x \otimes \triangle\mathsf{contents}\,x\ulcorner 0 \urcorner\} \\
&\mathsf{xpar}_1 : \mathsf{exec}\,(\mathsf{par}\,c_1\,c_2)\,d \\
&\qquad \multimap \{\exists d'{:}\mathsf{dest}.\mathsf{exec}\,c_1\,d' \otimes \mathsf{exec}\,c_2\,d' \otimes \mathsf{kont}\,(\mathsf{par}_\mathrm{k}\,d')\,d\} \\
&\mathsf{xpar}_2 : \mathsf{return}_\mathrm{c}\,d' \multimap \mathsf{return}_\mathrm{c}\,d' \multimap \mathsf{kont}\,(\mathsf{par}_\mathrm{k}\,d')\,d \multimap \{\mathsf{return}_\mathrm{c}\,d\}
\end{aligned}$$

Figure 10: CLF$^{\triangle}$ signature of the imperative parallel language

Syntax. We only consider the following fragment of the language:

$$\begin{aligned}
&x && \text{Variables} \\
&e ::= \cdots \mid x && \text{Expressions} \\
&c ::= \cdots \mid x := e \mid \mathtt{reset}\,x \mid c_1 \parallel c_2 && \text{Commands}
\end{aligned}$$

Integer expressions e include variables (x) for implicit dereference, and commands c include assignment ($x := e$), clearing the contents of a variable ($\mathtt{reset}\,x$), and parallel composition ($c_1 \parallel c_2$). We assume the granularity of actions are individual variable reads and writes. We use x, y, z for variables in both the imperative language and its encoding. The encoding of the syntax and semantics as a CLF$^{\triangle}$ signature is given in Figure 10.

Signature for Syntax. The signature for syntax and state is:

$$\begin{aligned}
\mathsf{var}, \mathsf{exp}, \mathsf{cmd} &: \mathsf{type} \\
\mathsf{get} &: \mathsf{var} \to \mathsf{exp} \\
\mathsf{set} &: \mathsf{var} \to \mathsf{exp} \to \mathsf{cmd} \\
\mathsf{rst} &: \mathsf{var} \to \mathsf{exp} \\
\mathsf{par} &: \mathsf{cmd} \to \mathsf{cmd} \to \mathsf{cmd} \\
\\
\mathsf{contents} &: \mathsf{var} \to \mathsf{exp} \to \mathsf{type} \\
\mathsf{write} &: \mathsf{var} \to \mathsf{type}
\end{aligned}$$

The types var, exp, cmd describe the syntactic categories of the language. The constructors get, set, rst, and par represent the corresponding expressions and commands. The type family contents represents the memory cell associated with each variable as an irrelevant variable so as to permit the reordering of concurrent reads, but we require any update to use a linear token associated with each variable (write) to ensure synchronization of updates to the same variable.

Encoding. The bijective encoding $\ulcorner \cdot \urcorner$ translates the various syntactic categories to well-typed CLF$^{\triangle}$ objects of the corresponding type. In particular, the translation translates variables $\ulcorner x \urcorner$:var:

$$\ulcorner x \urcorner = x,$$

expressions $\ulcorner e \urcorner$:exp:

$$\ulcorner x \urcorner = \mathsf{get}\, \ulcorner x \urcorner,$$

and commands $\ulcorner c \urcorner$:cmd

$$\begin{aligned}
\ulcorner x := e \urcorner &= \mathsf{set}\, \ulcorner x \urcorner \\
\ulcorner c_1 \parallel c_2 \urcorner &= \mathsf{par}\, \ulcorner c_1 \urcorner \ulcorner c_2 \urcorner \\
\ulcorner \texttt{reset}\, x \urcorner &= \mathsf{rst}\, \ulcorner x \urcorner
\end{aligned}$$

Signature for Semantics. Following Pfenning [2004], we represent concurrent computation using a substructural operational semantics and linear destination-passing style. We introduce the auxiliary types and constructors for destinations, frames, and continuations. Intermediate computations are associated with a destination of type dest, evaluation of an expression returns the value to the destination ($\mathsf{return}_{\mathrm{e}}$) and execution of commands returns to the destination ($\mathsf{return}_{\mathrm{c}}$) after performing the desired side-effect:

$$\begin{aligned}
\mathsf{dest} &: \mathsf{type} \\
\mathsf{return}_{\mathrm{e}} &: \mathsf{exp} \to \mathsf{dest} \to \mathsf{type} \\
\mathsf{return}_{\mathrm{c}} &: \mathsf{dest} \to \mathsf{type}
\end{aligned}$$

A suspended computation frame of type rest is necessary for assignment (set_k) while the expression is evaluated and before the variable is updated, and for parallel composition (par_k) while both commands are executed and before they both return:

$$\begin{aligned}\mathsf{rest} &: \mathsf{type}\\ \mathsf{set}_\mathrm{k} &: \mathsf{var} \to \mathsf{dest} \to \mathsf{rest}\\ \mathsf{par}_\mathrm{k} &: \mathsf{dest} \to \mathsf{rest}\end{aligned}$$

Finally, kont is the type of continuations that recombines a frame with the result.

$$\mathsf{kont} : \mathsf{rest} \to \mathsf{dest} \to \mathsf{type}$$

The rules for evaluation of expressions and execution of commands are classified by the type constructors:

$$\begin{aligned}\mathsf{eval} &: \mathsf{exp} \to \mathsf{dest} \to \mathsf{type}\\ \mathsf{exec} &: \mathsf{cmd} \to \mathsf{dest} \to \mathsf{type}\end{aligned}$$

Evaluating a variable expression (get) accesses the appropriate memory cell, returns the value, and leaves memory unchanged:

$$\begin{aligned}\mathsf{eget} : \mathsf{eval}\,(\mathsf{get}\,x)\,d &\multimap \mathsf{contents}\,x\,v\\ &\overset{\triangle}{\multimap} \{\mathsf{return}_\mathrm{e}\,v\,d \otimes \triangle\mathsf{contents}\,x\,v\}\end{aligned}$$

Executing variable assignment (set) first creates a fresh destination d', spawns a subcomputation to evaluate the expression and a continuation with the appropriate frame:

$$\begin{aligned}\mathsf{xset}_1 : \mathsf{exec}\,(\mathsf{set}\,x\,e)\,d&\\ \multimap \{\exists d'{:}\mathsf{dest}.&\,\mathsf{eval}\,e\,d' \otimes \mathsf{kont}\,(\mathsf{set}_\mathrm{k}\,x\,d')\,d\}\end{aligned}$$

When the subcomputation returns the result ($\mathsf{return}_\mathrm{e}$), the assignment resumes the frame (set_k) by consuming the memory cell and write token ($\mathsf{write}\,x$), and updates the memory cell and returns control ($\mathsf{return}_\mathrm{c}$):

$$\begin{aligned}\mathsf{xset}_2 : \mathsf{return}_\mathrm{e}\,v\,d' \multimap \mathsf{kont}\,(\mathsf{set}_\mathrm{k}\,x\,d')\,d &\multimap \mathsf{write}\,x \multimap \mathsf{contents}\,x\,v'\\ &\overset{\triangle}{\multimap} \{\mathsf{return}_\mathrm{c}\,d \otimes \mathsf{write}\,x \otimes \triangle\mathsf{contents}\,x\,v\}\end{aligned}$$

Resetting a variable also requires the memory cell and write token, and immediately sets the contents to zero:

$$\begin{aligned}\mathsf{xrst} : \mathsf{exec}\,(\mathsf{rst}\,x)\,d &\multimap \mathsf{write}\,x \multimap \mathsf{contents}\,x\,v\\ &\overset{\triangle}{\multimap} \{\mathsf{return}_\mathrm{c}\,d \otimes \mathsf{write}\,x \otimes \triangle\mathsf{contents}\,x\,\ulcorner 0 \urcorner\}\end{aligned}$$

Parallel execution first first creates a fresh destination d' and spawns concurrent computations for the constituent commands:

$$\begin{aligned}\mathsf{xpar}_1 : \mathsf{exec}\,(\mathsf{par}\,c_1\,c_2)\,d&\\ \multimap \{\exists d'{:}\mathsf{dest}.&\,\mathsf{exec}\,c_1\,d' \otimes \mathsf{exec}\,c_2\,d' \otimes \mathsf{kont}\,(\mathsf{par}_\mathrm{k}\,d')\,d\}\end{aligned}$$

The parallel execution joins on the subcomputations when both have completed, and returns control:

$$\mathsf{xpar}_2 : \mathsf{return}_\mathrm{c}\, d' \multimap \mathsf{return}_\mathrm{c}\, d' \multimap \mathsf{kont}\,(\mathsf{par}_\mathrm{k}\, d')\, d \multimap \{\mathsf{return}_\mathrm{c}\, d\}$$

In these examples we use mnemonics that reflect the type, and in particular we assume each variable v is assigned an initial value n and a write token, whence the irrelevant context contains a corresponding memory cell $c_v^n \stackrel{\triangle}{:} \mathsf{contents}\, v \ulcorner n \urcorner$ and the linear context contains a write token $w_v \stackrel{\wedge}{:} \mathsf{write}\, v$. The intended use of the encoding is that the value resulting from evaluating an expression or the memory state after executing a command can be obtained with the logic programs:

$$\begin{aligned}
&\mathsf{eval}\, e\, d \multimap \{\mathsf{return}_\mathrm{e}\, v\, d \otimes \top\} \\
&\mathsf{exec}\, c\, d \multimap \{\mathsf{return}_\mathrm{c}\, d \otimes \triangle\mathsf{contents}\, x\, v \otimes \top\}
\end{aligned}$$

where e is an expression, c is a command, d is a concrete destination, and v is a logic variable. In the case of expressions the resulting value of the expression is bound to v, while in the case of commands the final memory state can be inspected and in particular variable x has value v. Via the operational semantics of proof search, a goal of the form $A \multimap \{S\}$ is proved by adding the antecedent A to the linear context, nondeterministically applying clauses of the logic program to consume and produce resources from the context, and finally attempting to prove the succedent S from the resulting context. Moreover, the result of successful search is realized by a proof term that represents the trace of the concurrent computation that the initial state of the antecedent into the final state of the succedent. In the expression case, when the clause $\mathsf{eval}\, e\, d$ is added to the context and the proof constructors are successfully applied, the computation completes with $\mathsf{return}_\mathrm{e} \ulcorner n \urcorner\, d$ in the context for some n, so the goal $\mathsf{return}_\mathrm{e}\, v\, d$ can be solved by unifying v with $\ulcorner n \urcorner$ while the other linear and irrelevant hypotheses are consumed by $\top$. Similarly in the command case, when the clause $\mathsf{exec}\, c\, d$ is added to the context and the proof constructors are applied, the final state reflects the side effects of variable updates, so the goal $\mathsf{return}_\mathrm{c}\, d$ is satisfied because the execution is complete, $\triangle\mathsf{contents}\, x\, v$ is solved by unifying with the memory cell $\mathsf{contents}\, x \ulcorner n \urcorner$ in the irrelevant context, and again the rest of the state is discarded with $\top$.

Concurrently reading the shared variable x in the command $y := x \parallel z := x$ has CLF$^\triangle$ encoding $\mathsf{par}\,(\mathsf{set}\, y\,(\mathsf{get}\, x))\,(\mathsf{set}\, z\,(\mathsf{get}\, x))$. Executing the command (xpar_1) with a destination d creates an intermediate destination d' and leads to the parallel execution of the assignments. The execution of each assignment (xset_1) will in turn create intermediate destinations d_y, d_z and computations $g_{xy} \stackrel{\wedge}{:} \mathsf{eval}\,(\mathsf{get}\, x)\, d_y$, $g_{xz} \stackrel{\wedge}{:} \mathsf{eval}\,(\mathsf{get}\, x)\, d_z$. In CLF these concurrent reads could not be reordered because reading from memory cell c_x^- causes a spurious synchronization, but in CLF$^\triangle$ they can be swapped because proof irrelevance eliminates contention on the read-only variable and the command has a single execution sequence modulo reordering.

We use write objects to enforce uniform write synchronization. The execution of the command $x := 0 \parallel x := 1$ concurrently updates x with different values and has an intermediate step involving destinations d_0, d_1 and computations $s_0 \mathbin{\hat{:}} \mathsf{set_k}\, x\, d_0$, $s_1 \mathbin{\hat{:}} \mathsf{set_k}\, x\, d_1$. These steps cannot be commuted as they lead to distinct states $\triangle\mathsf{contents}\, x \ulcorner 0 \urcorner \neq_{\mathrm{S}} \triangle\mathsf{contents}\, x \ulcorner 1 \urcorner$ which cannot be equated at the type level. However, the command $x := 0 \parallel x := 0$ does agree at the type level and without the write token we would have two indistinguishable execution sequences. By requiring $\mathsf{xset_2}$ to use the $\mathsf{write}\, x$ token, we achieve write synchronization even when the updates are identical.

If the encoding did not require assignments to hold the write token, we would lose uniform synchronization and distinct silent writes would be confused. Without using the write token, the execution term $\mathsf{xset_2}$ would be replaced by:

$$\begin{aligned}\mathsf{xset}_2' : \mathsf{return_e}\, v\, d' \multimap \mathsf{kont}\,(\mathsf{set_k}\, x\, d')\, d \multimap \mathsf{contents}\, x\, v' \\ \overset{\triangle}{\multimap} \{\mathsf{return_c}\, d \otimes \triangle\mathsf{contents}\, x\, v\}\end{aligned}$$

The concurrent update $x := 0 \parallel x := 0$ would involve the computations $s_0 \mathbin{\hat{:}} \mathsf{set_k}\, x\, d_0$, $s_0' \mathbin{\hat{:}} \mathsf{set_k}\, x\, d_0'$. Since s_0 and s_0' can commute without violating the side conditions of the concurrent equality rule, the command would only have one trace. However, the concurrent update $x := 0 \parallel x := 1$ would still have two traces because the final states remain distinguishable.

Token synchronization can also be used to enforce synchronization between different operators. In order to guarantee synchronization between variable assignment and reset, both $\mathsf{xset_2}$ and rst use the write token. The concurrent assignment and reset of x in the command $x := 0 \parallel \texttt{reset}\, x$ has two distinct traces depending on whether the assignment ($\mathsf{xset_2}$) or the reset (xrst) completes first, even though both subcommands lead to the same state $\triangle\mathsf{contents}\, x \ulcorner 0 \urcorner$. In the absence of token synchronization, with the initial value $n_x = 0$ of x the subcommands would be indistinguishable and the command would have only one trace. However, with a different initial value $n_x \neq 0$ the order of the subcommands would be distinguished because the type of the initial contents $c_x^{n_x} \overset{\triangle}{:} \mathsf{contents}\, x \ulcorner n_x \urcorner$ differs from the intermediate state $c_x^0 \overset{\triangle}{:} \mathsf{contents}\, x \ulcorner 0 \urcorner$: if the reset executes first the trace includes $\mathsf{xrst}'\hat{}\_\hat{}^{\triangle} c_x^{n_x}$ followed by $\mathsf{xset}_2'\hat{}\_\hat{}\_\hat{}^{\triangle} c_x^0$, but if the assignment executes first the trace includes $\mathsf{xset}_2'\hat{}\_\hat{}\_\hat{}^{\triangle} c_x^{n_x}$ followed by $\mathsf{xrst}'\hat{}\_\hat{}^{\triangle} c_x^0$, so the two xrst' terms are incomparable because they don't have the same type and the two traces are not equal up to permutation. Here we use xset_2' and xrst' as the constructors that don't use the synchronization token, and replace unimportant terms with an underscore (_).

The concurrent read and write of x in $y := x \parallel x := 0$ reveals a subtlety in our design. The computation either first performs the read ($\mathsf{eget}\hat{}\, g_{xy}\, \hat{}^{\triangle} c_x^{n_x}$) followed by the write ($\mathsf{xset_2}\hat{}\, r\hat{}\, k\hat{}\, w_y\, \hat{}^{\triangle} c_x^{n_x}$), or first performs the write ($\mathsf{xset_2}\hat{}\, r\hat{}\, k\hat{}\, w_y\, \hat{}^{\triangle} c_x^{n_x}$) followed by the read ($\mathsf{eget}\hat{}\, g_{xy}\, \hat{}^{\triangle} c_x^0$). Since the $\mathsf{eget}\hat{}\, g_{xy}\, \hat{}^{\triangle} c_x^m$ object has type $\{\mathsf{return_e}\, m\, d \otimes \triangle\mathsf{contents}\, x\, m\}$ when the value of x is m, the $\mathsf{return_e}\, m\, d$ object exposes the value of x concealed by the proof irrelevant $\triangle\mathsf{contents}\, x \ulcorner m \urcorner$ object. Therefore the two execution traces are equated only if the initial value of x is $n_x = 0$. In this work, we defer on whether such silent writes should be allowed

to commute with reads.

4 Future and Related Work

Watkins et al. [2004] developed CLF as an improved extension of Linear LF [Cervesato and Pfenning, 2002] with intrinsic support for concurrency. We closely follow the canonical forms presentation of CLF and augment the type theory with a proof irrelevance modality and a richer definitional equality.

The second author has considered an unrestricted proof irrelevance modality [Pfenning, 2001] wherein proof irrelevant variables may be used an arbitrary number of times whereas our primitive notion of proof irrelevance is linear and has a first-class modal operator. We conjecture that unrestricted proof irrelevance and linear proof irrelevance are orthogonal because neither can be encoded with the other. Unrestricted irrelevance cannot directly encode linear irrelevance because the intuitionistic case lacks primitive support for resource management. Conversely, representing unrestricted irrelevance with linear irrelevance would require a combination of the unrestricted ! and linear irrelevant $\triangle$ modality, but due to the syntactic restriction of those modalities only applying to asynchronous types, it would be necessary to use the monad to compose the modalities. However, attempting to encode unrestricted irrelevance using the monad (e.g., $\triangle\{!A\}$ or $!\{\triangle A\}$) is less general than a pure unrestricted irrelevance modality because the monad weakens truth to lax truth.

The LolliMon logic programming language of López et al. [2005] uses a type theory based on CLF that combines concurrent and saturating computation. They give an operational semantics that combines committed choice and forward chaining in the monadic fragment with backtracking and backward chaining outside the monad. It will be interesting to consider how the operational semantics interacts with proof irrelevance and how to efficiently use concurrent equality to prune the space of computations.

Proof nets [Girard, 1987] constitute a geometrical approach to identifying notionally equivalent computations in linear logic. The method has been applied successfully to fragments of linear logic, although the general case remains an active research topic. For example, Hughes and van Glabbeek [2003] have recently proposed a new notion of proof net for unit-free MALL. However, we do not see how proof nets could capture proof irrelevance in our sense, and the monadic approach with commuting conversions and proof irrelevance of $\text{CLF}^{\triangle}$ appears to be a promising alternative.

Models for true concurrency date back to Mazurkiewicz traces [Mazurkiewicz, 1977] and Pratt's pomsets [Pratt, 1984]. True concurrency is achieved in $\text{CLF}^{\triangle}$ by considering execution traces modulo reordering of independent steps and ignoring proof irrelevant terms. We see our work as a generalization and perhaps more syntactic, type-theoretic form of these prior approaches.

The next step of this work will involve an implementation of a typechecker for the $\text{CLF}^{\triangle}$ type theory. Concurrent equality can potentially be described as graph isomorphism of the dependence graph of let-bindings where objects are

vertices and each bound variable induces an directed edge to the object where the variable is consumed. The $\top$ connective introduces some difficulty because it acts as a sink for all unused resources. Trivial equality for proof irrelevant terms suggests that graph isomorphism should be restricted to the result of the $(\cdot)^*$ translation. Furthermore, it may be possible to use partial-order methods to avoid backtracking over all possible permutations of a sequence of bindings.

5 Conclusion

We have presented the logical framework CLF$^{\triangle}$, an extension of the CLF type theory with a linear proof irrelevance modality, which internalizes linear proof irrelevant hypothetical judgments, and a generalized definition of concurrent equality, which captures a richer form of true concurrency than that of CLF. The computational interpretation of linear irrelevant terms is that they are resources whose computational origin is unimportant and only their existence matters.

Proof irrelevant terms of the same type are always considered equal, which can be used to equate computations that use distinct but isomorphic objects. Concurrent equality is extended to admit the arbitrary reordering of concurrent computations modulo proof irrelevant terms. Therefore definitional equality can be used to avoid spurious synchronization due to false causal dependencies. Incorporating proof irrelevance into CLF achieves a more expressive framework and preserves the meta-theoretic properties, including decidability and complexity.

We illustrated the effectiveness of these extensions by contrasting the CLF and CLF$^{\triangle}$ encodings of Petri nets, an AI planning domain, and an imperative parallel language. In these examples, named objects of the same type become truly isomorphic by employing the proof irrelevance modality, spurious synchronization can be eliminated by making the resource under contention proof irrelevant, and token synchronization provides a general method for controlling the degree of concurrency between different actions that use the same resource.

References

Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992.

Iliano Cervesato and Frank Pfenning. A Linear Logical Framework. *Inf. Comput.*, 179(1):19–75, 2002.

Jean-Yves Girard. Linear Logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8): 666–677, 1978.

Dominic J. D. Hughes and Rob J. van Glabbeek. Proof Nets for Unit-free Multiplicative-Additive Linear Logic (Extended abstract). In *Proceedings of the 18th IEEE Symposium on Logic in Computer Science (LICS'03)*, pages 1–10, 2003.

Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic Concurrent Linear Logic Programming. In A. Felty, editor, *Proceedings of the 7th International Symposium on Principles and Practice of Declarative Programming (PPDP'05)*, pages 35–46, Lisbon, Portugal, ACM Press 2005.

Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. Technical Report Technical Report DAIMI PB 78, Aarhus University, 1977.

Antoni W. Mazurkiewicz. True versus artificial concurrency. In *PSTV*, pages 53–68, 1995.

Eugenio Moggi. Computational Lambda-Calculus and Monads. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science (LICS'89)*, pages 14–23, 1989.

C. A. Petri. Fundamentals of a Theory of Asynchronous Information Flow. In *Proceedings of 2nd Information Processing Congress (IFIP'62)*, pages 386–390, 1962.

Frank Pfenning. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In J. Halpern, editor, *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS'01)*, pages 221–230, Boston, Massachusetts, jun 2001. IEEE Computer Society Press.

Frank Pfenning. Substructural Operational Semantics and Linear Destination-Passing Style. In Wei-Ngan Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS'04)*, page 196, Taipei, Taiwan, November 2004. Springer-Verlag LNCS 3302.

Vaughan R. Pratt. The Pomset Model of Parallel Processes: Unifying the Temporal and the Spatial. In *Proceedings CMU/SERC Workshop on Analysis of Concurrency*, pages 180–196. Springer-Verlag Lecture Notes in Computer Science LNCS 196, 1984.

Kevin Watkins. CLF: A logical framework for concurrent systems. Thesis proposal, May 2003.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A Concurrent Logical Framework I: Judgments and Properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A Concurrent Logical Framework: The Propositional Fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, pages 355–377. Springer-Verlag LNCS 3085, 2004. Revised selected papers from the *Third International Workshop on Types for Proofs and Programs*, Torino, Italy, April 2003.

Glynn Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.