A Unified Theory of Program Logics:
An Approach based on the $\pi$-Calculus

Kohei Honda  
Department of Computer Science  
Queen Mary, University of London  
kohei@dcs.qmul.ac.uk

Nobuko Yoshida  
Department of Computing  
Imperial College London  
yoshida@doc.ic.ac.uk

Abstract

Facing a staggering diversity of software behaviours in modern and future computing, we argue for the need of a unified theory of program logics for a wide variety of software behaviours as a foundation of software engineering. We propose Hennessy-Milner logic for typed $\pi$-calculi as one possible such foundation. The $\pi$-calculus enjoys a singular position among computational calculi through its ability to embed sequential and concurrent programs as name passing processes without losing semantic information, and through its connection to other basic theories such as Linear Logic and Game Semantics. The embedding of programs in processes leads to the embedding of logics for programs in the logic for processes, where the observational content of a given program logic is made explicit, analysed and justified on a uniform basis. As a case study, we show embeddings of Hoare logic for sequential programs and a rely-guarantee logic for shared variable concurrency, suggesting that the proposed framework can offer a unifying basis to capture fundamental notions in program logics such as partial/total correctness, sequentiality and different kinds of concurrent computing.

Keywords: The $\pi$-calculus, Types, Hoare logics, Hennessy-Milner logics, Relay-guarantee logic, Logical full abstraction

1. INTRODUCTION

Diversity of software behaviours in modern and future computing environments is staggering. In their external behaviours, they can be stateless or stateful, first-order or higher-order, sequential or concurrent, shared memory or message passing. In their internal structure, they often crucially depend on the use of a variety of concurrent or even distributed components and services. Is there a unifying theory which helps us harness this diversity, acting as a common foundation for building, validating and controlling these complex software systems? Such a uniform foundation will become all the more essential when an application consists of distributed components that are written in different programming languages.

This paper proposes the $\pi$-calculus [45] and its logic as a candidate for such a unifying foundation, building on the well-cultivated use of compositional program logics as a scientific basis of programming [20]. Compositional program logics offer a key underpinning of diverse engineering efforts in software development, by its ability to describe and validate arbitrary semantic properties of programs – one can pinpoint any specific property of interest as an assertion, in order to use the description as a basis of engineering, be it specification, modelling, refinement, verification, testing or program analysis. By creating the structural universe of properties, program logics also lead to a deep understanding of semantics of programs. Compositional program logics as have been known so far are however inherently programming language dependent and do not allow unifying treatment of a variety of features such as sequential and concurrent, first-order and higher-order, functional and stateful computation.

The proposed theory aims to expand the key merits of program logics to a general class of software behaviours, independently from specific programming languages and their logics, while elucidating, analysing and justifying them on a common logical basis. The theory is built on the two kinds of representability known in the literature. The first is the exact embeddability of $\textit{behaviours}$ of diverse programming languages (sequential and concurrent, shared variables and message passing, first-order and higher-order, stateless and stateful) into the $\pi$-calculus by means of $\textit{types}$. The employed type structures for precise representability find their origins in Milner’s sorting [44] and game semantics [3, 4, 12, 30, 34],

BCS International Academic Conference 2008 – Visions of Computer Science 259
the latter offering semantically complete representation (full abstraction) of a wide range of programming languages in the universes of dialogue games, whose connection to the \( \pi \)-calculus was first pointed out in [33]. The second representability is logical embedding of program logics in Hennessy-Milner logics [19], a representative process logic. Such an embedding of a concurrent Hoare logic is first studied in [43]. A recent work [11] introduces a Hennessy-Milner logic for typed \( \pi \)-calculi, based on those for CCS [48, 49] and the untyped \( \pi \)-calculus [6, 14, 46], with an embedding of a program logic for higher-order functions in a process logic [28]. These two kinds of representability offer a unifying foundation for diverse program logics, in the following way: given a concrete program logic, they lead to logical full abstraction, a notion studied by Longley and Plotkin [39], albeit in a different setting: a program satisfies an assertion in the program logic if and only if the program’s representation as a process satisfies the assertion’s embedding in the process logic.\(^1\) Thus validity is reflected and preserved through the embedding, making the process logic a common ground in which each program logic finds its exact mirror image, with its observational content made explicit, analysed and justified on a uniform basis, through the fine grain logical primitives of Hennessy-Milner logic.

As a case study, we treat two well-known assertion methods for imperative computing in our framework: Hoare logic for sequential programs and a rely-guarantee method for shared variable concurrency. After introducing the typed \( \pi \)-calculus (Section 2) and its logic (Section 3), we first treat Hoare logic for partial correctness [16, 20] (Section 4), showing how the “must” modality in Hennessy-Milner logic can precisely embed the notion of partial correctness in Hoare logic, leading to logical full abstraction. The dual embedding of total correctness in the “may” modality leads to another logical full abstraction. We then turn to shared variable concurrency, embedding a version of Jones’ rely-guarantee logic [36], extending a recent work by Coleman and Jones [13] to a logic for shared variable concurrency which can treat a general case of interfering commands, obtaining the third logical full abstraction (Section 5).

**Other Approaches to Unifying Theories.** Abramsky [2] puts forward a general framework to extract program logics from a class of algebraic CPOs. As far as semantics of programs is given in CPOs with a class of operators and they satisfy certain conditions, this method induces program logics. In comparison, both approaches use representation of programs’ semantics in typed mathematical universes. On the other hand, the process-based framework is arguably more flexible and exact in capturing semantics of programs, due to the fine-grain nature of name passing interactions [4, 21, 33, 34, 40, 45] as well as expressiveness of the may-must modalities in Hennessy-Milner logic. Domain-theoretic structures are also essential in the construction and analysis of process-based semantics [4, 8, 34].

Hoare and He [23] study different (observational, algebraic and operational) ways of presenting a theory for imperative programming, and show they are equivalent in the sense that each is derivable from another in a cyclic way. Different presentations serve different engineering purposes and clarify distinct forms mathematical theories of computing can take. The present approach differs from theirs in that we focus on embeddings of different languages and their logics into a specific mathematical structure, name passing processes. The latter unification may be able to strengthen the unification in the former approach. More recently Hoare and O’Hearn [24] introduced a logical method for concurrent processes based an idea of separation and concurrency. The notion of concurrent composition in their logic has interesting features such as demanding geometric separation and including not only process-theoretic parallel composition but other forms of program compositions. These features, including its emphasis on graphical aspects, make their approach close to a specification framework which uses programs and predicates interchangeably [22]. Common to these works [22–24] is a (relatively) synthetic view on program logics, whose interaction with our (relatively) analytic approach would be an interesting topic for further study.

The semantically precise logical embeddings of existing program logics (also see [11] for further results) as outlined in this paper, either using process logics or not, may not be known in the literature. These results are based on the accumulated findings in theories of processes and programs, in particular process algebras and game semantics. There are different approaches to program logics [15, 18, 47, 51]; the applicability of the present framework to these and other forms of program logics would be an interesting subject of study.

---

\(^1\)The study by Longley and Plotkin in [39] considers a single program logic and investigates the validity induced by two kinds of program semantics, operational and denotational. If validity coincides, we say the denotational semantics is logically full abstract. Apart from the difference in the ambient universes (processes vs. CPOs), their framework corresponds to ours by observing that the embedding of a program logic in a process logic in our framework indeed induces an interpretation of the former.
2. PROCESSES AND TYPES

2.1. Processes

We use the asynchronous \(\pi\)-calculus \([29]\) with branching and selection \([8, 52, 54]\). Its communication primitive is asynchronous, whose fine-grain nature is useful for precise representation of various language constructs. The constructs for branching and selection come from \([17, 25, 52]\), and are essential for tractable typing \([27]\). These notions are (albeit in different presentations) in correspondence with interaction structures in game semantics \([8, 27, 54]\) and the polarised version of Linear Logic \([38]\).

The grammar of processes (ranged over by \(P, Q, \ldots\)) is given by:

\[
P ::= \quad 0 \mid a(x).P \mid \pi(e) \mid a \vdash [l_i(y_i).P_i]_{i \in I} \mid a \ll l(e)
\]

\[
\quad \quad \mid \text{if } e \text{ then } P \quad \quad \mid \text{else } Q \mid P|Q \mid (\forall a)P \mid (\text{rec}X(x).P)(\check{e}) \mid X(\check{e})
\]

Symbols \(a, b, c, \ldots, x, y, z, \ldots\) range over labels for branching/selection. Expressions \((e, e', \ldots)\) are numbers, booleans (tt, ff), the standard arithmetic and boolean operators (such as \(e + e'\)) and names. \(^2\) \(X\) is a process variable. \(I, I, \ldots\) range over labels for branching/selection.

Among processes, an input \(a(x).P\) receives a value from a message \(\pi(e)\); and a branching \(a \vdash [l_i(y_i).P_i]_{i \in I}\) (where \(I\) is finite) waits with \(I\)-labelled branches from a selection \(a \ll l(e)\), which is the dual of branching and which selects the \(l\)-labelled branch. \(P \mid Q\) is a parallel composition and \((\forall a)P\) is a hiding of \(a\). The recursive process \((\text{rec}X(x).P)(\check{e})\) consists of two parts, a recursive definition \((\text{rec}X(x).P)\) and its parameter vector \(\check{e}\). \(\text{rec}X(x).P\) consists of a process variable \(X\) (a binder), its parameters \(x\) (zero or more binders for values), and its body \(P\). \(\text{In}(P)\) denotes the set of free channels appeared in \(P\).

The use of the branching and selection enables not only a direct correspondence with type structures (as in the use of sums and products in typed \(\lambda\)-calculi) but also tractable representation of diverse language constructs. The use of the conditional (which can be encoded by branching) is also for tractable encodings of languages and proof rules.

**Convention.** We use the following abbreviations.

1. \(!a(x).P\) for \(\mu X.a(x).(P|X)\).
2. \(\pi(e)P\) stands for \((\forall c)(\pi(c)|P)\). Further we write \(\pi((e))P\) for \((\forall c)(\pi(c)|P)\). Similarly we write \(a \ll l(e)P\) and \(\pi(a \ll l(e))P\).
3. We often omit \(0\) and an argument or parameter if it is empty, e.g. \(\bar{k}\) and \(\text{rec}X.P\).

The structural congruence \(\equiv\) is standard, in which we include the unfolding rule for recursion \((\text{rec}X(x).P)(\check{e}) \equiv P(\check{e}/x)[\text{rec}X(x).P/X]\) with \(e_i \downarrow v_i\) (\(e \downarrow v\) means \(e\) evaluates to \(v\)). The reduction rules are generated by (with \(e_i \downarrow v_i\)):

\[
\begin{align*}
\;&a(x).P \quad \pi(e) \quad a \vdash [l_i(y_i).P_i]_{i \in I} \quad a \ll l(e) \quad \rightarrow \rightarrow \quad P[\check{v}/x] \quad a \vdash [l_i(y_i).P_i]_{i \in I} \quad a \ll l(e) \quad \rightarrow \rightarrow \quad P_j[\check{v}/x] \quad (j \in I)
\end{align*}
\]

as well as the if-then-else rule, closing under the evaluation contexts and structure rules.

As an example, an encoding of an imperative variable \(\text{Var}(xv)\) based on \([31, 43]\) follows:

\[
\text{Var}(xv) = (\text{rec}X(xv).x \triangleright [\text{read}(e)(X \triangleright xv)|\pi(v)] \& \text{write}(v'k)(X \triangleright xv)|\bar{k}))(xv)
\]  \hspace{1cm} (2.1)

In \(\text{Var}(xv)\), \(x\) is a name of the variable and \(v\) is a stored value. This process waits for invocation with two branches at \(x\), with its left branch for reading and its right branch for writing. The read branch receives the request with a single name \(c\) as a continuation. This continuation is used to return its content \(v\). In the write branch, it receives two names, \(v'\) and \(e\), and uses \(v'\) as its new value (thus changing its state) and acknowledges the receipt of the new value via \(k\). Then the assignment command \(x := y\) with the acknowledgement \(k\) is encoded as:

\[
[[x := y]]_k \triangleq y \triangleright \text{read}(e)c(y')(.(v\triangleright c')(x \triangleright \text{write}(y'c') | c')|\bar{k})
\]  \hspace{1cm} (2.2)

\(^2\)Booleans, integers and operations on them, as well as branching and selection, can be dispensed with by translating them into name passing, without changing semantic properties of the calculus. We use these data types and operations for tractable technical presentation and analysis.
2.2. Types

We use the type structures studied in [8], which is an instance of the family of linear type disciplines [9, 26, 31, 35, 37, 44, 53, 54] and which can embed PCF fully abstractly. The common idea of the linear type discipline is to constrain the direction of information flow (input or output), the kind of information that flows on a channel, and how often a channel may be used: linear (at most once) or replicated (an unlimited number of times, including not at all), originating in Linear Logic [17]. Concretely, a linear typing enforces the following constraints:

1. For each linear name, there is a unique input, and a unique (or at most one) output
2. For each replicated name, there is a unique repeatedly available input and zero or more outputs

To separate deterministic and non-deterministic behaviours, we further introduce the following constraint called sequentiality, coming from Game Semantics [4, 34].

3. Only one single thread (output) can be active at one time, at present and potentially.

The typing with the constraints (1,2,3) is called sequential typing, while one with (1,2) (without 3) is called non-deterministic linear typing. The former is for embedding sequential programs (Section 4), whereas the latter can embed concurrency (Section 5). For example, by (1), $P_1 \overset{def}{=} b.\overline{a} | b.\overline{e}$ is typable because $b$ is associated to two inputs. But $P_2 \overset{def}{=} !b.\overline{a} | b.\overline{e}$ is typable since, while an output at $b$ occurs twice, a replicated input at $b$ appears only once. Finally by (3), $P_3 \overset{def}{=} \overline{a} | !b.(\overline{a} | \overline{e} | \overline{e}) | \overline{a}$ is typable, even though $c$ appears twice.

The conditions informally specified above are formally guaranteed by a simple typing system [8]. We only outline the definitions of types and explain the key ideas of type discipline. Type are given by:

$$ p ::= \bot | \top | \top | \bot \mid \tau ::= \text{nat} \mid \text{bool} \mid (\tau)^p \mid \{l : \tau\}_{i \in I}^p \mid \{\oplus l : \tau\}_{i \in I}^p \mid \top $$

**Action modes** $(p, p', \ldots)$ [8] prescribe different kinds of interaction at each channel. $\bot$ denotes a *linear input*; its dual $\top$ a *linear output*; $\downarrow$ a (replicated) *server*; and its dual $\uparrow$ denotes *client* . $\text{nat}$ and $\text{bool}$ are atomic types. Type $(\tau)^p$ represents a channel carrying types $\tau$ and to be used specified as $p$; for example, $(\tau)^{\bot}$ receives values once, while $(\tau)^{\top}$ receives them repeatedly. Type $\{l : \tau\}_{i \in I}^p$ represents waiting with $n$ options, and inputting values of type $\tau$, with mode $p$; type $\oplus\{l : \tau\}_{i \in I}^p$ is selection. $\bot$ indicates a channel is no longer available for further composition with the outside: for example, if $b.\bot$, then $b.\bot | \overline{b.\overline{a}}$ is a $\bot$-mode and $\overline{b}$ has a $\top$-mode, then $b.\bot | \overline{b}$ has $\bot$-mode at $b$.

The *dual type* of $\tau$, written as $\bar{\tau}$, is defined by exchanging $!$ and $?$, $\top$ and $\bot$, and $\&$ and $\oplus$. The atomic types are self-dual. The key element of the type discipline is a partial commutative and associative operator [8] which controls a parallel composition of two processes is defined as:

(a) $(\tau)^{\bot} \circ (\overline{\tau})^{\top} \equiv \bot$; 
(b) $(\tau)^{\top} \circ (\overline{\tau})^{\bot} \equiv (\tau)^{\bot}$; 
(c) $(\tau)^{\top} \circ (\tau)^{\bot} \equiv (\tau)^{\bot}$.

Similarly for branching/selection. (a) says that once we compose two processes at a session channel, then that channel becomes no longer composable. (b) says a server should be unique, to which an arbitrary number of clients can request interactions.

Typing environments $\Gamma, \Delta, \ldots$ contain two kinds of type assignments, one of the form of $a : \tau$, mapping a name to a channel type; and the other of the form $X : \tau$, mapping a process variable to a vector of channel types. The typing judgement has the shape $\Gamma \vdash P$. The typing rules are identical with [8]. The key rule is for parallel composition which uses $\circ$. Suppose $\Gamma_i \vdash P_i$ with $i = 1, 2$. We first check $\Gamma_1(a) \circ \Gamma_2(a)$ is defined for each $a$, then we derive $\Gamma_1 \circ \Gamma_2 \vdash P_1 | P_2$. For example, $P_1$ and $P_2$ are typable by (a) and (b), respectively. The sequentiality condition is guaranteed by prohibiting two active outputs from being parallel-composed.

As an example, $x$ in the variable agent $\overline{\text{Var}}(xv)$ in (2.1) with $v$ of type $\tau$ is typed as:

$$ \text{var}(\tau) \overset{def}{=} \{\text{read} : (\tau)^{\bot} \& \text{write} : \tau(\bot)\}^{\top} \quad (2.3) $$

On the other hand, typing $x$ from the user’s viewpoint, $x$ in the agent in (2.2) has its dual type $\{\text{read} : (\tau)^{\bot} \oplus \text{write} : \tau(\bot)\}^{\top}$, which is composable by $\circ$ with $\text{var}(\tau)$.

The linear typing with recursive agents without sequentiality can express the non-determinism, see [31].
3. LOGIC FOR TYPED PROCESSES

3.1. A Logical Language

Our logical language is Hennessy-Milner logic with equality, value/name passing modality and fixed point formulae [6, 14], augmented with three new operators. The grammar of assertions \( (A, B, C, \ldots) \) follows.

\[
A ::= e_1 = e_2 \mid A \lor B \mid \neg A \mid \forall x^\tau A \mid \Box A \mid (\ell)A \mid (\mu X(\bar{x}).A)(\bar{e}) \mid X(\bar{e})
\]

Above \( \ell \) ranges over \( a\) (\( \bar{b} \)) \( \bar{e} \), \( \pi (\bar{b}) \bar{e} \) and \( a \ll (\bar{b}) \bar{e} \) where \( \bar{b} \) gives new name binding on expressions \( \bar{e} \). We define \( A \lor B, A \supset B, \exists x^\tau A \), \( [\ell]A \), \( [\beth]A \), and \( (\nu X(\bar{x}).A)(\bar{e}) \), by duality.

\( [\ell]A \) says that the process has a semantically immediate, or strong, \( \ell \) action, satisfying \( A \) as the result (technically it denotes the strong modality [14] taken up to the weak bisimilarity). \( \Box A \) says that after zero or more silent actions, the process satisfies \( A \) (dually in \( [\beth]A \), after whatever zero or more silent actions the process may have, the process still satisfies \( A \)). We write \( (\ell)A \) for \( \Box (\ell)A \), saying that a weak \( \ell \)-transition leads to \( A \) (thus denoting the standard weak modality). Dually \( [\ell]A \) says that any weak \( \ell \)-transition leads to \( A \). The combination of strong and weak modalities is important for descriptive power, proof systems and axioms. The minimal/maximal fixed points use parameters following [6, 14], which are essential for describing, among others, \textit{state-changing loops}, as found in the variable agent in (2.1). \( X(\bar{e}) \) should never occur negatively [14] in a formula.

We now illustrate three new operators. The roles of these operators are, on the one hand, to allow a precise logical description of semantics of processes (\( \times, \triangleright \)); and, on the other, to enable a flexible rely-guarantee-based [36] logical description of process behaviour (\( \triangleright \) in combination with \( \times, \triangleright \)). First, \( A \triangleright B \) (read as “\( A \) \textit{par} \( B \)”) is understood as \( A, B \) in [49]. Informally, a process \( P \) satisfies \( A \triangleright B \) when \( P \) has the same observable behaviour as \( Q|R \), such that \( Q \) satisfies \( A \) and \( R \) satisfies \( B \). Secondly, \( A \triangleright B \) (read as “\( A \) \textit{then} \( B \)”) is a typed version of the consequence relation studied in [48], and allows specifications and reasoning under hypothetical composition of another process, enabling the rely-guarantee-based reasoning for general process (hence program) behaviours [36]. Concretely a process \( P \) satisfies \( A \triangleright B \) if, for each appropriately typed \( Q \) satisfying \( A \), the composition \( P|Q \) satisfies \( B \). Both constrain the typing of \( A \) and \( B \): For example, for \( \Gamma \vdash P \) with \( \Gamma(k) = ()^\top \) to satisfy \( B \triangleright C \), \( k \) can be typed as \( ()^\top \) in \( B \), and, if so, \( k \) is typed \( \bot \) in \( C \) (note \( \tau \triangleright \tau = \bot \) for linear \( \tau \)). Finally \( \forall x^\tau A \) is the quantifier for name hiding, \( A \) process, say \( P \), satisfies \( \forall x^\tau A \) if there is a fresh name \( a \) of type \( \tau \) and \( P' \) such that \( (\nu a)P' \approx P \) and \( P' \) satisfies \( A \), where \( \approx \) is the weak bisimilarity. Having this operator in the logic is important for calculation of \( \times \).

We often omit type annotations for quantifiers. \( T \) denotes \( 1 \times 1 \), \( F \) its negation. The interpretation of assertions follows [14], except the three new operators: for details see [11]. The mixture of the strong and weak modalities gives us the following modality:

\[
[\ell]A = [\beth]([\ell]A) \quad (3.1)
\]

The modal formula \( [\ell]A \) (read: “surely \( \ell \) then \( A \)”), which mixes two modalities, says that now or after any silent actions of the process, it can do a strong \( \ell \)-action, and then satisfies \( A \). We call this modality \textit{mixed modality}. Intuitively one may consider this modality as a logical description of the semantics of a prefixed process (either a message or an input prefix).

We give a simple example of the assertion for the encoding of an imperative variable given in (2.1). Under the typing in (2.3), we can assert for the process \( \text{Var}(x,v) \) as:

\[
\text{VarSpec}(x,v) \overset{\text{def}}{=} (\mu X(\bar{y}).(\forall c. )(\llbracket x \rightarrow \text{read} \rrbracket)(\llbracket y \rrbracket) \triangleright X(\bar{y})) \land \\
(\forall w. )(\llbracket x \rightarrow \text{write} \rrbracket)(\llbracket w \rrbracket) \triangleright X(\bar{w}))\end{array})(\bar{y})
\]

In \( \text{VarSpec}(x,v) \), \( x \) is the name of the variable and \( v \) is an initial value. The specification says that the imperative variable can either be read, in which case it sends a natural number through a linear channel it receives as well as regenerating itself with the same state as before; or be written, in which case it receives a natural number and sends back an acknowledgement through a linear channel it receives as well as regenerating itself with the newly received value as its state.
3.2. Proof Rules

We write $\Gamma ; E \vdash P \triangleright A$ for provability, where $\Gamma$ types $P$ and $A$ (except auxiliary variables in $A$) and $E$ contains assignments of the form $X : (\bar{x})A$, mapping a process variable to a parametrised formula ($\bar{x}$ are binders). We often leave write $\Gamma$ implicit writing $E \vdash P \triangleright A$. There are three proof systems, one for May modality, one for Must, and one for their combination. They soundly and completely characterise the May/Must preorder and bisimilarity, respectively [11].

In Figure 1 we list the proof rules of the May modality. There is a single rule for each typing rule except $\text{Conseq}$. The prefix rules are standard. The rules for the must and mixed modalities (sound under the same semantics) essentially replace the may modality in the prefix rules with must and mixed modalities respectively, using the same rules for parallel composition and hiding as well as the same consequence rule. The key to these prefix rules is the representation of the “lack of the remaining possible actions” through the must modality. Because each process is equipped with a typing (which bounds possible actions the process can have), this information is finitely representable (cf. [6]). For example the rule for the inaction becomes:

$$
\Gamma ; E \vdash \top \triangleright \text{noact}(\Gamma)
$$

where we emphasised the typing $\Gamma$ for precision and the predicate $\text{noact}(\Gamma)$ says the process has (and will have) no action of types $\Gamma$. For example we have $\text{noact}(u: (\bar{x})^2) \equiv [[u]]F$. The prefixing rule for the Must modality similarly uses the inaction predicate, saying that the all and only immediate action is the prefix action itself. For example the must modality for the input prefix is inferred as:

$$
\Gamma, x : \text{nat}; E \vdash P \triangleright A \\
\Gamma, x : \text{nat}; E \vdash u(x).P \triangleright [[u(x)].\Gamma][A]
$$

where $[[u(x)].\Gamma][A]$ is expanded into: $\forall x : \text{nat}. [[u(x)]A \land \text{noact}(\Gamma)]$: here the first conjunct says that if any (well-typed) action at $u$ takes place then $A$ will hold; in this case the use of $[[u]]$ cannot be replaced by $[[u]]$ since we may not have $A \supset [[u]]A$. The second conjunct describes the inaction. For recursion, we use the following rule, which is useful when reasoning about partial correctness.

$$
E, X : (\bar{x})A \vdash P \triangleright A \quad A \text{ admissible} \\
E \vdash (\text{rec}X(\bar{x}),P)(\bar{e}) \triangleright [[A[\bar{e}/\bar{x}]]
$$

This rule allows cancellation of admissible formulae following [28], where admissibility is defined via syntactic unfoldings (the bottom case is 0 or, depending on its, divergent counterpart; and if a chain of finite unfoldings satisfy it, the limit should also satisfy it). Finally the proof rules for mixed modality replaces $[[\ell]]A$ in each prefix rule for the must modality with $\ell A$ introduced in (3.1) in the previous page, which semantically subsumes both the may and must modality prefix rules, as well as adding another rule for recursion. All of these proof rules are sound under the same semantics: thus, for example, the induction-based recursion rule $\text{Rec-ind}$ is valid when reasoning with must modalities. For details, see [11].
4. LOGIC FOR SEQUENTIAL IMPERATIVE PROGRAMS

4.1. Partial Correctness

**Hoare Logic.** Based on Floyd’s assertion method [16], Hoare puts forward the idea that programs should be well-structured and that they may as well be given a logical basis by compositional proof rules [20]. We consider the standard while programs:

\[
C ::= x := e \mid C_1; C_2 \mid \text{if } b \text{ then } C_1 \text{ else } C_2 \mid \text{while } b \text{ do } C \mid \text{skip}
\]

where \(e, e', \ldots\) denote first-order expressions of type boolean \(\text{bool}\) or natural numbers \(\text{nat}\), of which \(b, b', \ldots\) denote boolean expressions, evaluating either to \(\text{tt}\) or \(\text{ff}\). We only consider variables of type \(\text{nat}\). We use a simple typing from [43]: letting \(\Gamma\) be a finite map from program variables to \(\text{nat}\), we write \(\Gamma \vdash C\) if \(\text{dom}(\Gamma)\) covers all variables in \(C\) and variables and expressions in \(C\) are well-typed.

Assertions \((A, A', B, B', \ldots)\) of Hoare logic are those of number theory:

\[
A ::= e_1 = e_2 \mid A \land B \mid \forall x.A \mid \neg A \mid T
\]

The Hoare triple for partial correctness is written \(\{A\} C \{B\}\), where in \(A\) and \(B\) we may use program variables (which may occur in \(C\)) in addition to auxiliary variables (which should not occur in \(C\)). Semantically, the triple means that if we run \(C\) from a state satisfying \(A\) and if the computation terminates, then the resulting state satisfies \(B\). Here we are not concerned with programs which diverge, hence a diverging program satisfies arbitrary assertions. For example let

\[
diverge \overset{\text{def}}{=} \text{while } \text{tt} \text{ do } \text{skip},
\]

i.e. \(\text{diverge}\) is a diverging program. Then we have:

\[
\{T\} \text{diverge}\{F\}
\]

(4.1)

**hence \(\{A\} \text{diverge}\{B\}\) for any \(A\) and \(B\). There is a sound and complete proof system for Hoare logic [20]. Our purpose is to have a simple and exact embedding of this logic in the logic of processes.**

**Embedding: Programs.** The first step towards the logical full abstraction is embedding programs as processes. Imperative programs work by reading from and writing to memory cells. This is made explicit in process encoding: when evaluating an expression, a program interacts with memory cells to read their values.

\[
eval(e, y, P) \overset{\text{def}}{=} \begin{cases} 
\overline{e} \triangleleft \text{read}(c)(y).P & (e \text{ variable}) \\
\overline{P[e/y]} & (e \text{ constant}) \\
\eval(e_1, y_1, \eval(e_2, y_2, P[y_1 + y_2/y])) & (e = e_1 + e_2)
\end{cases}
\]

The process \(\eval(e, y, P)\) “evaluates” \(e\) and uses it as \(y\) in \(P\). The first line shows the case when \(e\) is a variable, in which case the process does a read interaction; the second when \(e\) is a constant; and the third an operator, taking the addition as an example. A program \(C\) is encoded as \(\llbracket C \rrbracket_k\) where \(k\) is a linear output signal for termination (“done” in [43]):

\[
\llbracket x := e \rrbracket_k \overset{\text{def}}{=} \eval(e, y, (\forall c)(\exists \text{write}(yc)(c, \overline{x})))
\]

\[
\llbracket C_1; C_2 \rrbracket_k \overset{\text{def}}{=} (\forall h)(\llbracket C_1 \rrbracket_k[h, \llbracket C_2 \rrbracket_k])
\]

\[
\llbracket \text{if } b \text{ then } C_1 \text{ else } C_2 \rrbracket_k \overset{\text{def}}{=} \eval(b, y, (\forall \text{if } y \text{ then } C_1 \text{ else } C_2))
\]

\[
\llbracket \text{while } b \text{ do } C \rrbracket_k \overset{\text{def}}{=} \mu X. \eval(b, y, (\forall \text{if } y \text{ then } \llbracket C \rrbracket_k[h, X] \text{ else } \llbracket C \rrbracket_k))
\]

\[
\llbracket \text{skip} \rrbracket_k \overset{\text{def}}{=} \overline{x}
\]

The encoding is well-typed in the following sense. Assume \(\Gamma \vdash C\). Write \(\Gamma^\circ\) which translates each type assignment in \(\Gamma\) say \(x : \text{nat}\) to \(x : \text{var}(\text{nat})\). Then we have \(\Gamma^\circ, k : (\cdot) \vdash \llbracket C \rrbracket_k\), which says the encoded program may repeatedly read and write at \(\Gamma\) and, in addition, may output at \(k\) once.

As an example of the encoding, \(\llbracket y := x + x \rrbracket_k\) becomes:

\[
\overline{\text{read}(c)(x_1).\overline{\text{read}(c)(x_2).((\forall c)(\exists \text{write}(x_1 + x_2c)(c, \overline{x}))}
\]

(4.2)
This process faithfully captures the abstract behaviour of the assignment command. Note this behaviour differs from the encoding of $[[y := 2 \times x]]_k$ which is given as:

$$\tau \triangleleft \text{read}(c)(x'), (v c)(\tau \triangleleft \text{write}(2 \times x') \mid c \vec{x})$$

which interacts only once at $x$. When computing is sequential, this difference does not lead to different effects. When computing involves interference, however, this difference becomes significant, as we shall see in the next section.

**Embedding: Validity.** Under the process representation of imperative computing, the Hoare triple $\{A\} C \{B\}$ can now be given the following interactional reading:

If $[[C]]_k$ interacts with memory cells whose initial values satisfy $A$, and if it outputs through channel $k$, then we shall find the memory cells with values satisfying $B$.

Write $A(\vec{x})$ for $A$ parametrised by the vector of variables $\vec{x} = x_1, \ldots, x_n$, then we set, letting $\vec{v} = v_1, \ldots, v_n$ to be the vector of fresh and pairwise distinct variables:

$$[[A(\vec{x})]] \overset{\text{def}}{=} \exists \vec{v}. (\bigwedge_{1 \leq i \leq n} \text{VarSpec}(x_i, v_i) \land A(\vec{v}))$$

which gives a full specification of the behaviour of memory cells, with initial values satisfying $A$. The encoding of a Hoare triple for partial correctness follows. Below $\Gamma \vdash \{A(\vec{x})\} C \{B(\vec{x})\}$ indicates $\Gamma \vdash C$ and $\text{dom}(\Gamma) = \{\vec{x}\}$, in addition to $\{A(\vec{x})\} C \{B(\vec{x})\}$.

**Definition 1 (Hoare Triple in Process Logic)** We define the encoding of a Hoare triple as:

$$[[\Gamma \vdash \{A(\vec{x})\} C \{B(\vec{x})\}, k]] \overset{\text{def}}{=} \Gamma^* \cdot k : () \vdash [[C]]_k \triangleright [[A(\vec{x})]] \triangleright [[\vec{x}]] [[B(\vec{x})]].$$

under the sequential typing system in Section 2.2.

Henceforth we often omit the typing of the encoding. The translation directly follows the interactional reading of a Hoare triple discussed above. The use of the Must modality $[[\vec{x}]]$ is essential to capture partial correctness. Recall the Hoare triple (4.1) for the diverging program $\text{diverge}$. Its encoding becomes, fixing $\vec{x}$ as program variables:

$$[[\text{diverge}]]_k \triangleright (\exists \vec{v}. \text{VarSpec}(\vec{x}, \vec{v})) \triangleright [[\vec{x}]] F$$

which asserts that whatever the initial values of the variables composed with the program may be, we shall never observe the linear output $\vec{x}$.

Another focal point of Definition 1 is that the encoding considers the properties of $[[C]]_k$ only under hypothetical composition with the process encoding of imperative variables. In the presence of the sequentiality constraint on typed processes (the third condition in Section 2.2), the first thing we can ever observe from this composition, by way of its typed labelled transition, is a termination of the process, signified by an output via $k$. Thus the pre/post conditions of the original Hoare triple are directly transported into the encoded typed formula. Further, since we can only observe the resulting effect, and never intermediate interactions between the process and the variables, intensional differences among programs, such as the one between (4.2) and (4.3) discussed above, are absorbed by the environment and hence are invisible, making the encoded judgements precisely mirror the original semantics. This mirroring relationship can be distilled with the following formal result.

**Theorem 1 (Logical Full Abstraction for Hoare Triples)** $\{A\} C \{B\}$ is valid in Hoare logic if and only if $[[\{A\} C \{B\}, k]]$ is valid under the sequential typing (cf. Section 2.2).

The proof depends on two properties: one is computational adequacy [41] of the encoding, showing $(C, \sigma)$ terminates with $\sigma'$ iff $[[C]]_k[[\sigma]]$ reduces to $[[\vec{x}]] [[\sigma']]$, where $[[\sigma]]$ is the process encoding of $\sigma$ using Var$(xv)$. The other is logical full abstraction of $[[A(\vec{x})]]$ in the sense that $P$ satisfies this property if and only if it is bisimilar to the encoding of an environment satisfying $A(\vec{x})$.
Embedding: Proof Rules. It is notable that all proof rules of Hoare Logic have a direct syntactic justification by the Must modality fragment of the proof rules of the typed Hennessy-Milner logic discussed in Section 3. In each case, we use basic axioms for $\diamond$ and $\triangleright$, which are essentially a logical version of the expansion laws [42]. We only outline the case for the partial correctness of the proof rule for $\text{while}$.

$$\frac{[A \land b] C [A]}{[A] \text{ while } b \text{ do } C [A \land \lnot b]} \quad (4.5)$$

Assume by induction we have:

$$[[C]]_h \triangleright [[A \land b]] \triangleright [[\bar{k}]] [[A]]. \quad (4.6)$$

Now the encoding of $\text{while } b \text{ do } C$ has the form $\text{rec } X_P$. Using (4.6), we can infer for this $P$, that:

$$X : [[A]] \triangleright [[\bar{k}]] [[A \land \lnot b]] \vdash P \triangleright [[A]] \triangleright [[\bar{k}]] [[A \land \lnot b]]. \quad (4.7)$$

Noting $[[A]] \triangleright [[\bar{k}]] [[A \land \lnot b]]$ is admissible (clearly the diverging process satisfies it; now suppose each point in the chain of finite unfoldings satisfies the formula: first if all points diverge then surely the limit diverges; second if some point converges with the result satisfying the post-state then the limit also does the same, noting that by sequentiality the process is quiescent after outputing via $k$; hence done), we can then apply $\text{Rec-adm}$ discussed in Section 3 to cancel $[[A]] \triangleright [[\bar{k}]] [[A \land \lnot b]]$ assigned to $X$ in (4.7), obtaining:

$$[[\text{while } b \text{ do } C]]_k \triangleright [[A]] \triangleright [[\bar{k}]] [[A \land \lnot b]] \quad (4.8)$$

as required. Other rules of Hoare Logic are similarly justified. We conclude:

**Theorem 2 (Embedding of Hoare Rules)** $\{A\} C B$ is provable in Hoare logic if and only if $\{[[A]] C [B], k\}$ is provable under the Must modality rules in the process logic, augmented with a finite number of axioms on assertions in the logic.

4.2. Total Correctness

Hoare logic for total correctness asserts that, for the same imperative computation as its partial counterpart, the computation terminates in addition to satisfying the stipulated pre and post conditions. Writing $[A] C [B]$ for Hoare triple for total correctness, the embedding in the process logic is identical except for having a small, but basic, difference.

$$[[C]]_k \triangleright [[A(\bar{i})]] \triangleright \langle\bar{k}\rangle [[B(\bar{i})]]. \quad (4.9)$$

The only difference of (4.9) from Definition 1 is that it uses the May modality rather than the Must modality for an output at $k$, indicating, because of sequentiality, that an output is necessarily emitted. The same logical full abstraction holds. The proof rules are justified through the May fragment of process logic: in particular, the standard termination sensitive proof rule for $\text{while}$ based on well-founded induction [16] is validated using $\text{Rec-ind}$. Below let $i$ be fresh and $i$ and $e$ be of the same type, which is an arbitrary well-founded set with $\leq$ denoting its order (for example it can be the pairs of natural numbers under the standard lexicographic ordering).

$$[[C]]_h \triangleright [[A \land b \land e = i]] \triangleright \langle\bar{k}\rangle [[A \land e \leq i]] \quad (4.10)$$

where we assume, following the standard proof rule for total correctness, that $A \land e = \perp \supset \lnot b$ holds, with $\perp$ being the least element in $\leq$ (e.g. zero in the natural numbers under the standard ordering). From (4.10), we can infer, through compositional reasoning, for the same $P$ as in (4.7):

$$X : \forall j \leq i, \exists k ([[A \land e = j]] \triangleright \langle\bar{k}\rangle [[A \land e = \perp]]) \vdash P \triangleright [[A \land e = i]] \triangleright \langle\bar{k}\rangle [[A \land e = \perp]]. \quad (4.11)$$

Now we can apply $\text{Rec-ind}$ in Figure 1, and then use $A \land e = \perp \supset \lnot b$, and as a result cancel $e = i$, reaching $[[\text{while } b \text{ do } C]]_k \triangleright [[A]] \triangleright \langle\bar{k}\rangle [[A \land \lnot b]]$, as required. Similarly we can justify the remaining rules.

**Theorem 3 (Logical Full Abstraction for Total Correctness)** $\{A\} C [B]$ is valid iff $\{[[A]] C [B], k\}$ is valid under the sequential typing in Section 2.2. Moreover $\{A\} C [B]$ is provable iff $\{[[A]] C [B], k\}$ is provable using the May modality proof rules in Figure 1.
5. A LOGICAL ANALYSIS OF SHARED VARIABLE CONCURRENCY

The logic treated in the previous section captures behaviours of programs based on the relation between initial and final states. When we introduce concurrency, this framework is no longer valid, due to interference at shared variables. A well-known method which offers modular reasoning for shared variable concurrency is the rely-guarantee framework introduced by Cliff Jones [36]. In the following we present a fully abstract embedding of a rely-guarantee logic for shared variable concurrency in the typed process logic. The rely-guarantee logic is built on a recent work by Coleman and Jones [13], combined with an analysis through process representation.4

5.1. Fine-Grain Imperative Concurrency

Syntax. The grammar of concurrent programs extends that of the sequential imperative language in Section 4:

\[ C ::= \ldots | C_1||C_2 | \text{now } x := e \text{ in } C \]

The first command introduces the parallelism, while the second the local variable declaration. This local variable declaration is not essential for the theory we shall develop, though it is convenient in many places. From a practical viewpoint, it is a natural and recommended practice to use local variables as much as possible when we write concurrent programs.

While we shall later discuss the $\pi$-calculus semantics of this extended language, it may be instructive to note that there is no change in the encoding of read, write, loop, conditional, and all other commands in processes: the only thing we do is we allow parallel composition of two (encoded) processes. This is precisely what the operational semantics we discuss next.

Dynamics. In the extended language, we allow fine-grain concurrency following [13], where individual reads, writes and evaluations can be interfered at each step, which is the standard dynamics in practical systems, both hardware and software. As illustration of the meaning of “fine-grain”, suppose $x$ and $y$ are originally 0 and we run the following simple command $y := x + x$. Now suppose the command $x := x + 1$ also runs in parallel in the present fine-grain operational semantics, i.e. we have a configuration

\[ y := x + x | x := x + 1. \quad (5.1) \]

Then the first and second reads of $x$ in $x + x$ may be different: more precisely, the results can be one of the following tuples, $(0,0)$, $(0,1)$, $(1,1)$, because the assignment in $x := x + 1$ can interfere at the beginning, in the middle, or after both of the two reads are done. Thus we cannot guarantee, for example, $y$ is an even number in the post-condition of $(5.1)$. The formal operational semantics is standard [13], starting from the rule which infers $(x := e, \sigma) \rightarrow (x := e', \sigma)$ from $(e, \sigma) \rightarrow (e', \sigma)$, allowing interference during the evaluation of $e$.

Challenge of Fine-Grain Concurrency. As this example suggests, fine-grain concurrency makes the understanding and reasoning on programs hard: indeed, this is why we use various atomicity operations such as lock and CAS. But, as indicated by the wide usage of spin lock, it is also a basis of building high-level abstractions for atomicity: treating fine-grain interference is an essential part of a logical foundation of practical shared variable concurrency.

To our knowledge, a recent work by Coleman and Jones [13] is the first to offer a rely-guarantee logic for such fine-grain concurrency. Their method allows a program, in its guard of the while command, to read from an interferable variable, enabling the treatment of non-trivial examples. On the other hand, they do not allow interference in assignment: in $x := e$, the variables in $e$ cannot be written by other threads (some mutual exclusion algorithms need such an assignment [7, §3]). The following shows a simple example which cannot be treated by the logic in [13].

4There are other recent works which use the rely-guarantee framework for fine-grain concurrency than the work by [13], cf. [50, 51]. They address different aspects of reasoning about fine-grain concurrency such as the use of locks. However [13] seems the only work so far that presents a framework addressing the reasoning about read of variables with fine-grain interference, presenting a proof method which can treat non-trivial examples.
Example 1 (Assignment of Interferable Variable) Consider the following three programs:

\[ C \overset{\text{def}}{=} x := 2; z := 1 \]

\[ C_1 \overset{\text{def}}{=} c := z; \text{if } c \geq 0 \text{ then } (a := x; b := y; \text{if } a = 0 \land b = 0 \text{ then diverge}) \]

\[ C_2 \overset{\text{def}}{=} a := x; b := y; c := z; \text{if } c \geq 0 \text{ then } (\text{if } a = 0 \land b = 0 \text{ then diverge}) \]

where \( \text{if } b \text{ then } C \) stands for \( \text{if } b \text{ then } C \text{ else skip} \). Above \( z \) is used as a flag to say \( x \) or \( y \) becomes non-zero. Assuming \( x, y \) and \( z \) are initially all zero, we can operationally show

\[ (*) \quad C \vdash C_1 \text{ always terminates, while } C \vdash C_2 \text{ may diverge.} \]

The only difference between \( C_1 \) and \( C_2 \) is the position of the phrase \( a := x; b := y \). Thus the timing of reading matters: the flag \( z \) is meaningful only when \( x \) and \( y \) are read after \( z \) becomes one.

The assignment of the form \( x := y \) with \( y \) interferable used above is significant not only by difficulty in reasoning but also because of its generality, in the sense that it can simulate other interferable commands ("\( x := e \)", "if \( e \) then \( C_1 \text{ else } C_2 \)" and "while \( e \) do \( C \)", each with multiple interferable variables in \( e \)), in combination with their non-interferable counterparts. Thus we may ask:

Can we refine the framework by Coleman and Jones to enable reasoning about interferable assignments?

In the following we present one possible answer to this question, reaching the compositional validation of the operational observation \((*)\) above. The logic is suggested by Milner’s logical analysis of shared variable concurrency using CCS \([43, \S 10]\) as well as the encoding into the typed process logic. The process logic also leads us to a logical framework for this language which is semantically complete in a certain technical sense, as we shall discuss at the end of the section.

5.2. A Rely-Guarantee Logic (1): Judgement and Formulae

Following Milner \([43, \S 10]\), our logic uses programs under a simple type discipline for read-write capability. The typing \( \Gamma ; \Delta \vdash C \) ensures that a concurrent program \( C \) uses the variables in the domain of \( \Gamma \) for reading and writing (without interference from others) and read those from \( \Delta \) (which may be interfered). As before, \( \Gamma \) and \( \Delta \) are finite maps from variables to their type which is always \( \text{nat} \), such that their domains are disjoint. \( \text{dom}(\Gamma) \) are called internal variables and \( \Delta \) external (or interferable) variables. The typing rules, which we omit, allow at most one thread to own the read/write capability for each variable.

Assertions are again those of number theory, written \( A, B, . . . \) as well as \( R, G, . . . \). The judgement is written:

\[ \Gamma ; \Delta \vdash [A, R] C [G, B] \quad (5.2) \]

where the program \( C \) should be typed under \( \Gamma ; \Delta \). Formulae \( A, R, G \) and \( B \) above are respectively called pre-condition, rely-condition, guarantee-condition and post-condition.

The pre/post-conditions \( A \) and \( B \) are about both internal and external variables so their program variables are in \( \Gamma \cup \Delta \); the rely-condition \( R \) is about external (interferable) variables, indicating their possible transition, using hooked/unhooked variables from \( \Delta \); for example \( R \) may say \( \forall y \leq y \) which asserts \( y \) is stable or increases over time. Similarly the guarantee-condition \( G \) uses hooked/unhooked program variables in \( \Gamma \).

Notation 1 (Parametrised Formulae) We often fix the domains of \( \Gamma \) and \( \Delta \) in \((5.2)\), say \( \vec{x} \) and \( \vec{y} \), and parametrise its formulae by these program variables as \( A(\vec{x}; \vec{y}) \), \( R(\vec{x}, \vec{y}) \) \( G(\vec{x}, \vec{y}) \) and \( B(\vec{x}; \vec{y}) \), where \( \vec{x} \) hooks variables point-wise (e.g. if \( \vec{x} = x_1 x_2 x_3 \), then \( \vec{x} = x_1 x_2 x_3 \)).

The notation above makes it explicit that a rely-guarantee formula is to be regarded as a binary relation on state. Regarding these formulae as relations in this way, we demand that \( R \) (resp. \( G \)) in \((5.2)\) is partially reflexive \([15]\) (its domain and codomain coincide and it contains the identity on the domain) and transitive (in the standard sense). Practically these conditions are often automatically met in rely/guarantee formulae by construction, as the following example shows.
Example 2 (Invariance and Rely/Guarantee) Returning to Example 1, we consider the condition relied by $C_1$ and guaranteed by $C$ in $C\models C_1$. Define:

$$I(xyz) \overset{\text{def}}{=} z \geq 0 \supset (x \geq 0 \lor y \geq 0) \quad (5.3)$$

$$J(\tilde{y}yz, xyz) \overset{\text{def}}{=} I(\tilde{y}yz) \land I(xyz) \land x \geq \tilde{x} \land y \geq \tilde{y} \land z \geq \tilde{z} \quad (5.4)$$

$I(\tilde{y}yz)$ and $I(xyz)$ are about invariance, saying: if $z$ is non-zero then $x$ or $y$ must also be non-zero. The remaining inequality is about progress, saying $x, y, z$ monotonically increase if ever. Immediately $J$ is partially reflexive and transitive.

The decomposition of a rely/guarantee condition into the invariant portion and the progress portion is important and we write $\text{inv}(R)(\tilde{x})$ for the invariant portion of $R$ (formally given as $\exists \tilde{y}. R(\tilde{x}, \tilde{y})$). For example, for the predicate $J$ above, $\text{inv}(J)(xyz)$ is $I(xyz)$ itself. We stipulate that $A$ is stronger than $\text{inv}(R)$ and $\text{inv}(G)$ in the sequent $\{A, R \} C \models \{G, B\}$.

5.3 A Rely-Guarantee Logic (2): Reading Order

Reading Variables in Fine-Grain Concurrency. In the standard rely-guarantee reasoning, we consider that, when a program reads a state, it can and must always read the snapshot of a store. This is not generally possible in the standard semantics of shared variable programs, i.e. in fine-grain concurrency. For example, consider the following state change, setting $C \overset{\text{def}}{=} x := x + 1; y := y + 2$.

$$(C, \{x \mapsto 0, y \mapsto 0, z \mapsto 0\}) \implies (y := y + 2, \{x \mapsto 1, y \mapsto 0\}) \implies (\text{skip, } \{x \mapsto 1, y \mapsto 2\})$$

Now suppose we compose $C$ with command $z := x + y$. In the standard (non-fine-grain) rely-guarantee logic, the command $z := x + y$ would surely read, for $x$ and $y$, one of the three states given above. However in the fine-grain concurrency, when it reads $x$ and then $y$, their values may as well be 0 and 2 respectively, since the value of $y$ can change between reading two variables. Thus, in fine-grain semantics, a command need to read a vector of variables over multiple states. 5 We call the result of a reading over multiple states, cross-state vector. Formally, let $\tilde{\sigma} = \sigma_1, \sigma_n$ be a sequence of states (called state sequence) with the identical domain, say $x_1, x_m = \tilde{x}$. Let $\sigma$ have the same domain. Then $\sigma$ and the map $\psi$ from $\tilde{x}$ to $\{1, \ldots, n\}$, called reading index or index, is a cross-state vector over $\tilde{\sigma}$ if for each $x_i$ in $\tilde{x}$, we have $\sigma(x_i) = \sigma_{\psi_i}(x_i)$. We call each $\sigma_i$, a slice. We often say $\sigma$ is a cross-state vector, leaving $\psi$ implicit. $\sigma$ precedes $\sigma'$ (over the same state sequence) if the first reads from, for each variable, the same slice or strictly preceding one. Note that, given a state sequence $\tilde{\sigma}$, a cross-state vector may read two or more variables in a single state.

Reading Order. The key idea in generalising the method by Coleman and Jones to allow the compositional reasoning of a general assignment from the interferable variable (of the form ‘$x := y$’, with $y$ interferable), is the introduction of the order by which a program should read variables, which we call reading order. For illustration of this idea, consider $C \models C_1$ and $C \models C_2$ in Example 1: the former always converges while the latter may diverge. The key difference between $C_1$ and $C_2$ is that $C_2$ reads $z$ first, while $C_1$ does not. Logically, this means $C$ guarantees $J$ in Example 2 as far as $z$ is read first: and $C_1$ can rely on the property since it does read $z$ first, in contrast to $C_2$. The use of such a partial order is omnipresent in many algorithms for realising mutual exclusion by locks and lock-free algorithms [7] where a shared variable is often used as a flag to signal a state change. From the viewpoint of rely-guarantee reasoning, the reading order narrows down possible cross-state vectors, thus facilitating the logical reasoning — a program can rely on and guarantee a stronger property when the reading order is stipulated. We capture this notion by enriching the rely (resp. guarantee) condition with a fixed partial order. Formally we incorporate the reading order to our logic by enriching the judgement $\Gamma; A \vdash [A, R]C[G, B]$ with two partial orders, one for the domain of $\Gamma$, and the other for the domain of $A$. We write $x \leadsto y$ when $x, y \in \text{dom}(\Gamma)$ are ordered by the partial order on $\text{dom}(\Gamma)$ and moreover they are distinct. We often omit $\Gamma$ and simply write $x \leadsto y$ (read: “$x$ should be read before $y$”). Informally, $x \leadsto y$ means that a program running concurrently should read $x$ from the same state as $y$ or from some preceding state, for it to read values satisfying the stipulated condition.

5While there is an algorithm by which we can take a snapshot of the state of multiple variables through additional steps [5], which in effect allows each command to look at only atomic snapshots, this is relatively expensive operations whose cost many practical shared variable programs may not afford except when absolutely necessary.
Consistency. If $\sigma$ is a cross-state vector over $\bar{\sigma}$ with index $\psi$, and $\leadsto$ is the reading order on $\text{dom}(\sigma)$, then we say $\sigma$ conforms to $\leadsto$ if whenever $x \leadsto y$ we have $\psi(x) \leq \psi(y)$. We further say $\sigma$ is an $A$-legal sequence if any consecutive slices in $\sigma$ are related by $A$. Then we say $A$ is consistent under $\leadsto$ if, for each subset $\bar{x}$ of $\text{dom}(\Gamma)$, whenever a cross-state vector $\sigma$ precedes another $\sigma'$ over a $(\exists \bar{x}.A)$-legal sequence $\bar{\sigma} = \sigma_1, \sigma_n$, we have that $\sigma$ and $\sigma'$ are related by $\exists \bar{x}.A$ again (we consider all restrictions of $A$ onto the subsets of variables since we may want each part of the condition to be meaningful independently).

Example 3 (reading order, conformance, consistency) Recall the following state sequence from Example 1:
\[
\{ x \mapsto 0, y \mapsto 0, z \mapsto 0 \} \{ x \mapsto 2, y \mapsto 0, z \mapsto 0 \} \{ x \mapsto 2, y \mapsto 0, z \mapsto 1 \}
\]
and recalling $z$ is used as a flag to indicate either $x$ or $y$ has turned non-zero, we may stipulate the reading order $z \leadsto x$ and $z \leadsto y$. And the state $\{ x \mapsto 0, y \mapsto 0, z \mapsto 1 \}$ does not conform to this reading order (even though it is indeed a cross-state vector over the state sequence above). However $\{ x \mapsto 0, y \mapsto 0, z \mapsto 1 \}$ is a (non-slice) cross-state vector conforming to the reading order above. Note also all slices conform to the order, by definition. Finally, taking $J$ from Example 1, whenever one reads $x, y, z$ following $z \leadsto x$ and $z \leadsto y$ in any sequence of states pairwise related by $J$, the values thus read always satisfies $\text{inv}(J)(xyz)$.

5.4. A Rely-Guarantee Logic (3): Proof Rules for Interferable Assignment

We can now introduce the proof rule for interferable assignment. Let $\bar{x}$ and $\bar{y}$ be the domains of $\Gamma$ and $\Delta$ respectively, $\text{ID}(\bar{x})$ stands for $\bar{x} = \bar{x}$, while $\downarrow (\bar{x}, \bar{y})$ denotes the set of names in $\bar{y}$ which are strictly below (which should be read strictly before $y$) with respect to $\leadsto$.

\[
\Gamma \vdash x : \text{nat} \quad \Delta \vdash y : \text{nat} \\
[l\text{-Assign}] \quad \exists (\bar{x}, \bar{y}).(A(\bar{x}; \bar{y}) \land \text{ID}(\bar{x}) \land R(\bar{x}, \bar{y})) \supset (G'(\bar{x}, \bar{y}) \land B(\bar{x}, \bar{y}))[\downarrow x / y] \\
[A,R] \quad x := y \quad [G,B]
\]

Above the existential quantifier $\exists (\bar{x}, \bar{y})$ in the premise disregards those variables that should be read strictly before $y$: for them the constraint in $R$ is not valid. As an example usage of this rule:

Example 4 (Reasoning for Interferable Assignment) Using $I$ and $J$ from Example 2 and the order on $x, y, z$ given above, a crucial part of $C_1$ from Example 1 can be inferred as follows.

\[
a, b, c : \text{nat} ; x,y,z : \text{nat} \vdash [l(\bar{x}, \bar{y}, \bar{z})], J \quad c := z \quad [T, I(xyc)] \quad (5.5)
\]

Now following $c := z$ above, $a$ and $b$ are assigned the values of $x$ and $y$ respectively (in $C_1$), reaching $I(abc)$, so that if $c = 1$ then we know either $a$ or $b$ is non-zero, concluding $C[1]C_1$ always converges. Note this inference is impossible for $C_2$ since it does not read $z$ first.

We omit the remaining proof rules which are standard, see [1]. The resulting reasoning framework can concisely reason about such non-trivial examples as the parallel predicate search treated in [13]. We now move to the process-logical justification of this logic, which is in fact one of its origins.

5.5. Process Encoding and Logical Full Abstraction

Encoding of Programs and Formulae. For the encoding, we use the non-deterministic linear typing, taking off the sequentiality constraint from the conditions in §2.2. The encoding for the extended grammar is given below:

\[
[[C_1 || C_2]]_k \overset{\text{def}}{=} (v h_1 h_2)[[[C_1]]_k][[[C_2]]_k][h_1, h_2, \bar{x}] \\
[[\text{new } x := e \in C]]_k \overset{\text{def}}{=} (v x)(\text{Var}(x e))[[[C]]_k]
\]

Other clauses remain the same. This defines $[[C]]_k$.

For the embedding of the rely-guarantee judgement, we first stipulate the mapping for a rely-guarantee formula

\[
g[[G(\bar{x}, \bar{x})]]_r, \quad (5.6)
\]
in which we use the ordering associated with \( \Gamma \). The construction is simple. For brevity we present the case when the vector \( \bar{x} \) is given as \( x_1, x_2 \). Assume first there is the order \( x_1 \rightarrow x_2 \). Then \( g[[G(\bar{x}, x_1, x_2)]]_{\Gamma} \) is given as

\[
\exists h. (\text{Read}(x_1, x_2) (i_1 i_2) \text{Read}(x_1, x_2) (j_1 j_2) [[h]] (i_1 i_2 j_1 j_2) T \supset \exists x_1 x_2 x_1, x_2 G(x_1 x_2, x_1 x_2)) \tag{5.7}
\]

where we set

\[
\text{Read}(x_1, x_2) (i_1 i_2) A \overset{\text{def}}{=} (v c) \langle x_1 \text{read}c \rangle \forall i_1. \langle x_1 \text{read}c \rangle \emptyset v_2. \langle v_2 \text{read}c \rangle v_1. (\forall i_2 v_2. \langle k' i_2 \rangle A) \tag{5.8}
\]

If on the other hand there is no order between \( x_1 \) and \( x_2 \), we use the same formula for each of the all possible interleaving of the reads at \( x_1 \) and \( x_2 \) and take their conjunction.

The assertion \( g[[G(\bar{x}, \bar{x})]]_{\Gamma} \) characterises a behaviour which, when composed with a terminating “probe” of two consecutive reads of the vector in the stipulated order, it will ensure the condition given by \( G \). Note this assertion does not describe the behaviour of the process itself: rather it characterises the result of the observations by a stipulated probe (which reads the vector twice consecutively). In fact no intensional details of behaviour of programs should be stipulated, since the key scope of the reasoning method of our rely-guarantee logic is to reason entirely based on state transformations induced by programs, rather than their direct intensional behaviour.

**Encoding of Judgement and Logical Full Abstraction.** Then given a well-typed, order-enriched formula \( \Gamma; \Delta \vdash [A, R] C [G, E] \), we define its encoding \( [[\Gamma; \Delta \vdash [A, R] C [G, E], k]] \) as follows, under the refined non-deterministic linear typing.

\[
[[C]]_k \rightarrow ([[A(\bar{x})]] \circ [[G(\bar{x}, \bar{x})]]_{\Delta}) \rightarrow [[B]] (g[[G(\bar{x}, \bar{x})]]_{\Gamma} \circ [[B(\bar{x})]]) \tag{5.9}
\]

The encoded formula (5.9) as a whole says that if we run \( [[C]]_k \) in parallel with (1) imperative variables whose values satisfy \( A \) and (2) any terminating process which ensures \( R \) at \( \bar{y} \) during computation, then \( G \) is ensured at \( \bar{x} \) during computation and there will surely be an output at \( k \). Here we are using the modal action \( [[\text{R}]] \), introduced in (3.1), which entails, even in the presence of non-deterministic computation paths, that this action surely takes place. By \( [[B]] \), we also know after this action we reach a stable state satisfying \( B \). Arguing by direct operational correspondence, we can show that the embedding is logically fully abstract in validity and proof rules. The embedding also justifies the proof rules for the rely-guarantee logic.

Based on fine-grain concurrency and a firm observational basis by the use of typed interactions, we can extend this framework to treat other language constructs for both sequential and concurrent computation. Such constructs include various atomic operations, structured message passing, objects and higher-order functions (for which we can use the embedding of the logics studied in [10, 32, 55]), unifying their treatment on a common mathematical foundation.

**Further Topics.** It is notable that the encoding of concurrent imperative programs in itself gives a semantically complete logic for the imperative concurrent programming language, in the sense that any observable properties of programs can be specified and reasoned. For this purpose we work with a formula of the form \( [[A(\bar{x})]] \supset B \) for \( [[C]]_k \), where \( [[A(\bar{x})]] \) is as before and \( B \) is a general formula from the process logic. In some sense, this is the “canonical logic” for this programming language, in which we can specify and infer any properties of interest through axioms in the process logic. Note this general logic now dispenses with, and subsumes, the presented reasoning framework based on a fixed reading order.

Since such a logic is represented (at least initially) as a logic for processes rather than a logic for programs, it may not directly offer a most convenient method for specifying and reasoning about programs: what we have shown in this section is, in effect, how we may obtain a relatively tractable reasoning framework by restricting the general logic to a class of well-behaved programs. But a significant fact is that we now have a general and rigorous technical basis for reasoning about a variety of programming language features, ranging from higher-order procedures to complex data types to diverse atomicity primitives to communication primitives, as far as they can be encoded into the \( \pi \)-calculus. The challenge is to deepen and broaden this framework, in both theories and applicability, so that we can make the best of its potential, drawing on the long accumulated studies on the semantics and logics of computing and in close dialogue with practice.
Acknowledgements. We thank a reviewer for useful comments and Martin Berger for ongoing collaborations on logics of computation. This work is partially supported by EPSRC grants GR/T03208, EP/F002114 and EP/F003757 and IST2005-015905 MOBIUS.

REFERENCES