

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Department of Computer Science and Engineering

Bachelor's Degree in Computer Science

Kairos
a reversible
and concurrent language

Supervisor:
Prof.
Ivan Lanese

Presented by:
Nicolò Giuliani

Session [session] [year]
Academic Year 2026/2027

[dedication]

Abstract

Kairos is a *reversible* and *concurrent* programming language, inspired by Janus [YG07] and extended to express concurrent problems. Every Kairos program can be executed both forwards and backwards, and its inverse is computed from the program text. The language offers explicit parallelism, synchronous communication over typed channels and transactions with compensation.

This thesis gives its structural operational semantics in small-step style, defined as an extension of the semantics of Janus by Lami, Lanese and Stefani [LLS24], and the inversion operator that realises backward execution. On channels we adopt a binary session discipline, checked during execution, which allows delegation, that is, the passing of a channel from one holder to another. The language is implemented by an interpreter, with a debugger that runs the program one step at a time, forwards and backwards. A case study, the reversible Burrows–Wheeler transform with the blocks transformed in parallel, measures its cost and the speedup obtained from concurrency.

Contents

0 Introduction **1**
- 0.1 Notation and conventions . . . 2
- 0.2 Structure of the thesis . . . 2

1 Background **5**

2 Kairos: syntax and operational semantics **9**
- 2.1 Abstract syntax . . . 9
- 2.2 Transition system and store . . . 11
- 2.3 The base semantics . . . 12
- 2.4 Expressions and conditions . . . 14
- 2.5 Semantics of commands ($\rightarrow_c$) . . . 14
- 2.6 Run-time errors . . . 25

3 Reversibility **27**
- 3.1 Summary of the rules . . . 29
- 3.2 `show` and observable output . . . 30
- 3.3 Conformance to Janus . . . 31

4 Programming in Kairos: reversible lossless compression **33**
- 4.1 The starting point . . . 33
- 4.2 What changes when writing it in Kairos . . . 34
- 4.3 The lexicographic comparison . . . 34
- 4.4 Blocks in parallel . . . 35
- 4.5 Results . . . 35
- 4.6 Limitations . . . 37

5 Implementation: interpreter and DAP **39**
- 5.1 The chain, from source to execution . . . 39
- 5.2 A step-by-step example . . . 39
- 5.3 Structures and execution loop . . . 41
- 5.4 Parallelism . . . 42
- 5.5 The DAP debugger . . . 43

6 Related work **45**

7 Conclusions and future work **47**

References **51**

List of Tables

3.1 The reduction rules of Kairos, grouped by construct. 29
3.2 Failure conditions and their outcome: error or block. 30

6.1 Kairos and CRIL compared. 46

Chapter 0

Introduction

A programming language is **reversible** when each of its computations can be retraced backwards until the starting state is recovered exactly. This is not a matter of recording a trace from which the past can be reconstructed: in a reversible language, the information needed to go back is already contained in the current state, because every construct is designed not to destroy it. The destructive assignment $x := e$, which erases the previous value of x, does not belong to a language of this kind; in its place we find invertible updates such as $x \mathrel{+}= e$, whose inverse $x \mathrel{-}= e$ is immediate.

The interest in this way of computing has three independent roots. The first is physical: Landauer's principle links the erasure of a bit to a minimum dissipation of energy, and a computation that does not erase can in principle avoid it. The second is pragmatic: many problems naturally come in pairs of inverse transformations (compression and decompression, encoding and decoding, serialization and parsing), and a reversible language makes it possible to write *a single* program instead of two, ruling out by construction the possibility that the two halves diverge. The third concerns reliability: being able to go back is the basic mechanism of *recovery*, and indeed the *rollback* of a transaction is nothing other than backward execution up to a safe point.

It is precisely the third root that makes the **concurrent** case interesting, and it is here that this work is situated. In a sequential system, backward execution is a simple notion, because there is only one past to retrace. In the presence of several threads the past is no longer linear: two branches proceeding in parallel may have interacted, and undoing a step of one may require first undoing the steps of the other that depend on it. The progenitor of reversible languages, **Janus**[YG07], is purely sequential; reversibility in a concurrent setting has been studied mainly on process calculi[LMSS11;LLM+13], which are models and not programming languages. The territory in between (a structured imperative language, with the syntax and constructs a programmer expects, that is both reversible and concurrent) is sparsely populated: attempts exist, but so far they have defined the concurrent constructs through their translation, without a formal semantics on which to ground the properties expected of a reversible language.

Kairos sits in that territory. The name is the Greek καιρός (*kairós*): not the time that flows, which is χρόνος (*chrónos*), but the opportune moment, the instant at which something must be done. The distinction is pertinent to a reversible language, where time is not an arrow but a coordinate that can be traversed in both directions, and

what matters about a step is not *when* it happens but that it is the right one: the inverse of a program does not rewind its history, it executes another program that leads back to the starting state.

To the core of Janus (invertible assignments, conditionals with a double guard, `from ... until` loops, procedures invoked with `call` and `uncall`, integers and arrays) Kairos adds explicit parallelism `par ... and ... rap`, synchronous communication on channels `ssend`/`srecv` with protocols reconstructed from the program, a stack as a primitive data structure and the transactions `try ... rollback ... yrt`. The design choice that holds everything together is *disjointness*: parallel branches operate on disjoint portions of the store and communicate only through channels. From this single hypothesis follows the confluence of the interleavings that do not touch channels (§2.5), and hence the fact that the residual nondeterminism is not observable on the final store: the concurrency of Kairos serves to express the parallelism of problems, not to introduce races.

The contribution of this document is the formal semantics of the language. We give a structural operational semantics in small-step style, defined as an **extension** of the semantics of Janus by Lami, Lanese and Stefani[LLS24]: we inherit from that base the form of the judgments, the evaluation contexts, the terminal configuration and the treatment of errors, and we add the rules for the constructs that Kairos introduces. We then define the inversion operator $\mathcal{I}(\cdot)$, which is syntactic and computable in linear time, and we show in what sense it realizes backward execution.

0.1 Notation and conventions

Judgments have the form $\langle \sigma, c \rangle \to \langle \sigma', c' \rangle$, with the store on the left as in the base semantics, not the big-step turnstile $\sigma \vdash e \Downarrow v$ customary in the Janus literature. This is not a stylistic variant: interleaving of branches and synchronization on channels (Section 2.5) are expressed much more naturally step by step, because a big-step semantics collapses the entire execution of a command into a single judgment and the points at which the other branch could be interleaved remain implicit. This is the same motivation stated in[LLS24], which adopts the small-step style for Janus precisely in view of a future concurrent extension.

A typographic convention concerns what lies *outside* the transition relation. Error rules, which produce $\bot$, have their name in red. Well-formedness constraints are distinguished according to the moment at which they are checked: in green those checked on the program text, before executing it; in orange those monitored during execution, because they concern state that the program text does not determine.

0.2 Structure of the thesis

In what follows, Chapter 1 recalls Janus, the small-step semantics on which we build, and the work on concurrent reversibility. Next, Chapter 2 is the technical core: abstract syntax, transition system, the boundary between what is inherited from the base and what Kairos adds, and the reduction rules from assignments up to `par` and transactions, with the run-time errors. Then Chapter 3 defines the inversion operator, proves its

properties and summarizes all the rules in a single overview. These are followed by Chapter 4 on the use of the language, Chapter 5 on the implementation, Chapter 6 on the comparison with existing work and Chapter 7 with the conclusions.

Chapter 1

Background

Janus and reversible programming

Janus was defined around 1982 by Lutz and Derby at Caltech[LD82], out of curiosity, and brought back to the attention of the community by Yokoyama and Glück[YG07], who gave it its first formal semantics and an invertible self-interpreter. It is a structured imperative language in which every construct is invertible by construction, following two recurring ideas.

The first is the **invertible update**. In place of destructive assignment Janus has only updates of the form $x \odot= e$ with $\odot \in \{+, -, \hat{}\}$, where x does not occur in e: the old value of x can be recovered from the new one by applying the inverse operation, and the condition on the occurrence of x guarantees that the value of e is the same before and after the step.

The second is the **double guard**. An irreversible conditional `if` e `then` c_1 `else` c_2 is not invertible because, once it has finished, it is no longer known which branch was taken. Janus therefore requires a second expression, the *assertion* at the end:

$$\texttt{if } e_1 \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi } e_2,$$

with the obligation that e_2 hold true at exit if and only if the `then` branch was taken. Backwards, e_2 decides the branch and e_1 acts as the assertion: the roles of the two expressions are swapped. The same scheme governs the loop

$$\texttt{from } e_1 \texttt{ do } c_1 \texttt{ loop } c_2 \texttt{ until } e_2,$$

where e_1 must hold only at the first iteration and e_2 only at the last, so that the backward entry condition is e_2 and the exit condition is e_1. The two bodies are executed according to the trace $c_1 \, [\, c_2 \; c_1 \,]^*$: c_1 carries forward the work of the iteration, c_2 advances the counter between one iteration and the next. Kairos adopts the same form, with the clauses `do` and `loop` optional: an absent body is equivalent to `skip`.

On these foundations Janus is *r-Turing complete*: it computes exactly the computable injective functions, without generating garbage. Its procedures are invoked with `call` forwards and `uncall` backwards, and the inverse of a program is obtained by a purely syntactic transformation.

A small-step semantics for Janus

The classical presentations of Janus, starting from [YG07], use a *big-step* semantics: a judgment $\sigma \vdash c \Downarrow \sigma'$ relates the store before and after the *complete* execution of c. This is a convenient choice for a sequential language, but it collapses the entire computation into a single step, and with it the points at which another execution could be interleaved.

Lami, Lanese and Stefani [LLS24] gave a **small-step** semantics of Janus, with judgments of the form $\langle \sigma, c \rangle \to \langle \sigma', c' \rangle$ between configurations, evaluation contexts for expressions and commands, terminal configuration $\langle \sigma, \texttt{skip} \rangle$ and errors signalled by a distinct outcome $\bot$. One of the stated aims of that work is precisely to prepare the ground for concurrent extensions, on which the properties of interest are more easily defined in small-step form. Kairos starts from there: adopting its conventions means that the presentation choices have already been made and argued upstream, and that the new rules are only those of the new constructs. §2.3 lists exactly the boundary between the two.

An alternative semantics for Janus, in *reversible* style, is that of Lanese and Vidal [LV26], in which the forward rules and the backward ones are given together on the same transition system instead of deriving the latter from a syntactic inverter.

Reversibility and concurrency

Outside imperative languages, reversibility in the presence of concurrency has been studied in depth on *process calculi*. The delicate point is that in a concurrent system there is no single order of execution to reverse: the correct notion is *causal-consistent reversibility*, according to which a step can be undone only after all the steps that causally depend on it have been undone. From this perspective reversibility must also be *controlled*, because going back without limits is of no use: in $\rho\pi$ and in croll-π [LMSS11;LLM+13] going backwards is governed by explicit primitives, and the resulting mechanism (one goes back to a chosen point and from there proceeds differently) is exactly that of transactions with compensation. The transactions `try` ... `rollback` ... `yrt` of Kairos (§2.5) are the programming-language version of the same idea.

Kairos, however, takes a more restrictive, and therefore simpler, route: parallel branches work on *disjoint* portions of the store and interact only through synchronous communication. There is no shared memory, hence there is no causal dependency to track between steps that do not communicate, and causal consistency reduces to the confluence proved in §2.5. The price is that some shared-memory programming schemes are not expressible; the gain is that a program that respects disjointness has no races.

Between process calculi and programming languages there is nevertheless a direct precedent: Janus has already been extended with a parallel composition construct and a message exchange, and the extension was realized with a compiler to C++ [Sam19]. That work defines the constructs through the code into which they are translated, not through reduction rules, and maintains a shared state among the branches; it is therefore the closest starting point to that of this thesis, and Chapter 6 takes it up again for a detailed comparison.

Channels and session types

Communication in Kairos takes place on *synchronous* channels, with protocols reconstructed from the program, under the constraint that each channel has at any instant at most two holders, a *binary session*. This is a restriction that comes from the theory of **session types** [HVK98;HYC08], where the type of a channel describes not only the data exchanged but the *sequence* of exchanges, and the types of the two endpoints must be dual. In Kairos the binary-session constraint serves to guarantee that the partner of every communication is determined: without it an `ssend` could synchronize with several different `srecv`, and the choice would be a nondeterminism observable on the store, in contrast with confluence.

The constraint comes in two forms, and the choice between the two is not a detail. In the *static* form a channel belongs to the same two branches for its whole life, and this can be read off the program text. In the *dynamic* form a channel may lie first between two branches and then between two others, provided that the first holder stops using it before the new one begins: this is what in session-type theory is called **delegation**, the mechanism by which, in π-calculus style, a process hands to a third party the channel on which it is talking. The second is strictly more expressive, and Kairos adopts it: the constraint is monitored by observing the execution (§2.5), not by reading the text of the program, because the text is not enough to follow a channel that changes holder.

It is worth stating where the boundary of this work lies. In the semantics the binary session constraint and the disjointness hypothesis are *assumptions* under which confluence holds, not a type system: the rules of §§2.4–2.5 do not prevent writing a program with a race, they merely guarantee nothing about it. The compiler applies a conservative check (no integer or array written in one branch and used in another, even through calls; no shared stack), and the interpreter reports, when it notices, two simultaneous updates of the same integer variable, without this being a guarantee; a complete type system remains to be done. The separation is deliberate and follows current practice, in which the calculus is defined without worrying about races and an overlaid type system rules them out; session types, binary or multiparty, are precisely a family of type systems of this kind. In §2.5 we give the binary discipline for the channels of Kairos, delegation included, and the session types that describe their protocol: types are not declared but reconstructed from the program, and the compiler rejects two endpoints of a channel with provably conflicting protocols. Labelled choice and channels with more than two participants are left out.

Chapter 2

Kairos: syntax and operational semantics

2.1 Abstract syntax

Base sets: integers $m \in \mathbb{Z}$, variables $v \in \mathit{Var}$. Derived categories:

Arithmetic expressions $e \in \mathit{Exp}$**.**

$$e ::= m \mid v \mid a[e] \mid e \oplus e \qquad \oplus \in \{+, -, *, /, \%\}$$

$a[e]$ is the read of the cell of index e of the array a. The multiplicative operators occur only inside expressions, never as an assignment: $*$ and $/$ have no inverse that returns the operand, while on the right of $+=$ they remove nothing, because the inverse of the assignment is $-=$ with the *same* expression, and this depends on variables that the command does not touch. $/$ and $\%$ are integer division and remainder; a zero divisor has no result and produces an error:

$$\frac{}{\langle \sigma,\, m/0 \rangle \to_e \bot} \quad \text{(Div-Err)}$$

with the twin rule for $\%$. This is a failure like the others (§2.6), not a conventional value: a conventional value would make the operation non-injective exactly where it must be injective. In program text, literals are non-negative (a negative value is written $0 - m$), except for the values of `local` and `delocal`; the interpreter represents integers as 64-bit signed two's-complement values. Operations are therefore modulo 2^{64}: a result that overflows wraps around (even $-2^{63}/-1$, which evaluates to -2^{63}), and a literal is read modulo 2^{64}, so that unsigned constants up to $2^{64}-1$ can also be written; a larger literal is a compile-time error.

Conditions $b \in \mathit{Cond}$**.**

$$b ::= e = e \mid e \neq e \mid e \geq e \mid e \leq e \mid e > e \mid e < e \mid b \wedge b \mid b \vee b$$

The connectives $\wedge$ and $\vee$ are those of Janus: a guard can combine several comparisons, as in $(t = 0) \wedge (num > 1)$. In the implementation, comparisons and connectives are

themselves expressions that evaluate to 1 or 0, and a guard is true when it is nonzero: one can therefore also write $x \mathrel{+}= (a < b)$. The connectives do not short-circuit: both operands are always evaluated, and each of them can produce IDX-ERR or DIV-ERR.

Commands $c \in \mathit{Com}$**.**

$c ::= \texttt{skip}$	(empty command)
$\mid \ell \mathrel{+}= e \mid \ell \mathrel{-}= e \mid \ell \mathrel{\hat{}}= e \mid \ell \mathrel{<\!\!=\!\!>} \ell$	(reversible assignments)
$\mid \texttt{local}\ v = m \mid \texttt{delocal}\ v = m$	(scope)
$\mid \texttt{local}\ a[n] = m \mid \texttt{delocal}\ a[n] = m$	(scope, array)
$\mid c\ ;\ c$	(sequence)
$\mid \texttt{if}\ b\ [\texttt{then}\ c]\ [\texttt{else}\ c]\ \texttt{fi}\ b'$	(conditional)
$\mid \texttt{from}\ b_1\ [\texttt{do}\ c_1]$ $[\texttt{loop}\ c_2]\ \texttt{until}\ b_2$	(loop)
$\mid \texttt{call}\ f(\bar{v}) \mid \texttt{uncall}\ f(\bar{v})$	(call / inverse)
$\mid \texttt{push}(v, s) \mid \texttt{pop}(v, s)$	(stack)
$\mid \texttt{ssend}(\langle \bar{v} \rangle, ch) \mid \texttt{srecv}(\langle \bar{v} \rangle, ch)$	(channels)
$\mid \texttt{par}\ c\ \texttt{and}\ c\ [\texttt{and}\ c]^*\ \texttt{rap}$	(parallel)
$\mid \texttt{try}\ b\ c\ [\texttt{rollback}\ c]\ \texttt{yrt}\ b'$	(commit / rollback)

The target of an assignment is a *location* $\ell ::= v \mid a[e]$: an integer variable or an array cell. The two forms behave in the same way, the only difference being that the cell must be identified by evaluating the index before writing.

The concrete syntax admits two abbreviations. A call can be written without the keyword, $f(\bar{v})$, and means $\texttt{call}\ f(\bar{v})$. The value m of `local` and `delocal` can also be a variable already in the store, which is read as in an expression: this is the form $\texttt{delocal}\ i = n$ that closes a counter that has reached the limit n.

A *program* is a list of procedures $\texttt{procedure}\ f(\bar{v})\ \{\, c \,\}$; $\bar{v}$ is the (possibly empty) list of parameters. This is the abstract syntax: here the sequence is written with a semicolon, $c_1\ ;\ c_2$, as in the rules of §2.5, whereas in program text commands are written one after the other, without separators, and the body of a procedure extends up to the next procedure, without braces (by convention, one command per line, indented). The conditional has **two** guards (b on entry, b' on exit) and so does the loop (b_1 on entry, b_2 on exit): this is what makes the language reversible (§3). Square brackets mark the *optional* clauses, as in Janus: an absent branch means `skip`, and in the rules that follow both are written out in full, without loss of generality.

Execution starts from the procedure `main`, without parameters, after all the procedures of the program have been registered: their order in the file does not matter. At the end of execution the interpreter shows the variables of `main`. A well-formed program also satisfies the following constraints, which the compiler checks before execution:

- in an assignment $v \oplus\!= e$ the variable (or array) v does not occur in e, nor in the index of the cell being written (§2.5); $\texttt{delocal}\ v = v$ is not allowed;

- `local` and `delocal` form a block, as in Janus: each `delocal` closes the most recently opened `local` in the same command sequence, no `local` remains open at the end of the sequence, and a `local` does not redeclare a parameter or a local that is still open (§2.5); a `delocal` reports the expected value and has the same type, and for an array the same length, as the `local` it closes;
- declarations without `local` (`int` x, `stack` s, `channel` c) occur only in `main`: they are its initial store, zero or empty, and are what the interpreter shows at the end of execution; in the other procedures every variable is a parameter or a local;
- procedures have distinct names, different from those of the built-in operations (`show`, `push`, `pop`, `ssend`, `srecv`) and from those that the compiler reserves for the procedures it generates (which end in `__inv` or begin with `__try_`), and `main` has no parameters;
- operands have the right type, according to the declaration in scope (the same name can be an integer in one block and a stack in another): an expression reads integers and array cells; an argument of `call`/`uncall` has the type of the corresponding parameter; `push` and `pop` connect an integer variable and a stack; `ssend` and `srecv` use a channel and a non-empty message of variables that are distinct from one another and from the channel (integers, stacks or channels); the value of a `local` or `delocal` is an integer for integers and arrays, `nil` for a stack, `empty` for a channel;
- every variable used by a command is in the store of the activation: it is a parameter, a declared variable or a local that is still open;
- every `call`/`uncall` names a procedure of the program, with as many arguments as it has parameters, and the arguments of a single call are distinct variables: as in Janus, passing is by reference, and with `call` $f(x, x)$ the command $a \mathrel{+}= b$ in the body would become $x \mathrel{+}= x$, which is not invertible;
- in a `par`, two branches that use the same channel do not have provably conflicting protocols (§2.5); that a channel has at most two holders is instead checked at run time (§2.6); a stack is not used by more than one branch; an integer or an array written in one branch is not used in another, even through the procedures called (§2.5);
- in `try`…`rollback`…`yrt` neither the body c nor the rollback body r contains `ssend`/`srecv` or a `par`, either directly or through `call`/`uncall` (§2.5).

2.2 Transition system and store

A transition system is a triple $\langle \Gamma, \top, \to \rangle$ where Γ is the set of states, $\top \subseteq \Gamma$ the terminal states, and $\to \subseteq \Gamma \times \Gamma$ the transition relation. $\to^*$ is its reflexive and transitive closure:

$$\frac{}{y \to^* y} \qquad \frac{y \to^* y' \quad y' \to y''}{y \to^* y''}$$

To give meaning to variables we introduce a **store**. In Kairos variables are of four types: `int` (integers), `int`$[n]$ (arrays: n integers, with n fixed at declaration), `stack`

(sequences of integers) and `channel` (channels: synchronization names between parallel branches). Arrays and stacks are both sequences of integers but are used in opposite ways: an array has fixed length and is read by index without consuming it, whereas a stack grows and shrinks and is touched only at the top. A channel does **not** contain values: it is a rendez-vous point, not a container (§2.5); unlike `stack`, an allocated channel keeps the same identity for its whole life, and $\sigma(ch)$ never appears in the update of a rule. Writing *Chan* for the (opaque) set of channel identities, the store is

$$\sigma : \mathit{Var} \rightharpoonup \mathbb{Z} \cup \mathbb{Z}^n \cup \mathbb{Z}^* \cup \mathit{Chan},$$

partial because the scope changes with `local`/`delocal`. For sequences: ε is the empty one, $n : \ell$ is the *cons* (the head n is the top of the stack). We write $\sigma[v \mapsto w]$ for the update and $\sigma \setminus v$ for the removal of v from the domain. Here ε denotes **only** the empty sequence of a `stack`: it is a value, not a command. The empty command is a distinct symbol, `skip` (§2.1, §2.5); the two notions must not be confused.

2.3 The base semantics

Kairos is not defined from scratch. Its semantics is given as an **extension** of the small-step operational semantics of Janus by Lami, Lanese and Stefani [LLS24]: from there we inherit the form of the judgment, the terminal configuration, the treatment of errors and the apparatus of evaluation contexts; on top of that base we add the constructs that Janus lacks. The design choices already made in that work are therefore not rediscussed here: where this document is silent, the convention of the base applies. In the following we call *the base* that semantics, and we point out explicitly every place where we depart from it.

Judgment. The rules have the form

$$\langle \sigma,\, c \rangle \to \langle \sigma',\, c' \rangle \qquad \text{or} \qquad \langle \sigma,\, c \rangle \to \bot,$$

where the latter denotes a **run-time error** due to the failure of an assertion. The store is on the *left* of the command, as in the base. The base uses a single relation $\to$ on all syntactic categories; here we keep the decoration $\to_e$ / $\to_b$ / $\to_c$ to distinguish the fragment of expressions, that of conditions and that of commands, but it is the same relation read on different categories.

Terminal configuration. In the base, computation terminates in $\langle \sigma,\, \texttt{skip} \rangle$, and for this reason there is *no* rule for `skip`: quoting [LLS24], *"there is no rule for* `skip`*, since* $\langle \sigma, \texttt{skip} \rangle$ *denotes the end of the computation"*. This is the standard convention, and we adopt it: every rule that completes a command concludes in $\langle \sigma,\, \texttt{skip} \rangle$, not in the bare store. The consumption of `skip` in a sequence is left to SeqS (§2.5), and the termination of a parallel branch to Par-L0/Par-R0 (§2.5).

Errors. The error outcome is $\bot$, an element distinct from every configuration. An error is not propagated by one rule for each construct, but by a single contextual rule CtxError (below): this is the choice of the base, and it replaces ad hoc propagation rules.

Evaluation contexts. To give a compact definition of the semantics, the base lifts the reduction of sub-terms through *evaluation contexts*: a context is a command in which a sub-term is replaced by a hole $\bullet$. We distinguish **expression** contexts $C_e[\bullet]$, whose hole is to be filled with an expression or a condition, and **command** contexts $C_s[\bullet]$, whose hole is to be filled with a command; $C_e[e]$ and $C_s[c]$ denote the command obtained by replacing $\bullet$ with e and with c respectively. Adapted to Kairos, the contexts are

$$
\begin{aligned}
C_e[\bullet] ::= {} & v \mathrel{+}= \bullet \mid v \mathrel{-}= \bullet \mid v \mathrel{\hat{}}= \bullet && \text{(§2.5)}\\
\mid {} & \texttt{if } \bullet \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi } b_2 && \text{(§2.5)}\\
\mid {} & \texttt{if } \mathit{tt} \texttt{ then skip else } c_2 \texttt{ fi } \bullet \mid \texttt{if } \mathit{ff} \texttt{ then } c_1 \texttt{ else skip fi } \bullet \\
\mid {} & ((\bullet, b_1),\, c_1,\, b_2,\, c_2) \mid (b_1,\, (\texttt{skip}, c_1),\, (\bullet, b_2),\, c_2) && \text{(§2.5)}\\
\mid {} & ((\bullet, b_1),\, (\texttt{skip}, c_1),\, b_2,\, (\texttt{skip}, c_2)) \\
C_s[\bullet] ::= {} & \bullet ;\, c && \text{(§2.5)}\\
\mid {} & \texttt{if } \mathit{tt} \texttt{ then } \bullet \texttt{ else } c_2 \texttt{ fi } b_2 \mid \texttt{if } \mathit{ff} \texttt{ then } c_1 \texttt{ else } \bullet \texttt{ fi } b_2 && \text{(§2.5)}\\
\mid {} & (b_1,\, (\bullet, c_1),\, (b_2, b_2),\, c_2) \mid ((b_1, b_1),\, (\texttt{skip}, c_1),\, b_2,\, (\bullet, c_2)) && \text{(§2.5)}\\
\mid {} & \texttt{par } \bullet \texttt{ and } c \texttt{ rap} \mid \texttt{par } c \texttt{ and } \bullet \texttt{ rap} && \text{(§2.5, specific to Kairos)}
\end{aligned}
$$

The clauses with quadruples belong to the loop and are explained in §2.5; those of `par` are Kairos's only addition, and are discussed in §2.5. There is no clause for `try`: its rules (§2.5) reduce the body with $\to_c^*$ in a premise, not in place, so the body is not a context hole. The three contextual rules are those of the base:

$$\frac{\langle \sigma,\, e\rangle \to_e \langle \sigma,\, e'\rangle}{\langle \sigma,\, C_e[e]\rangle \to_c \langle \sigma,\, C_e[e']\rangle}\ (\textsc{CtxExp}) \qquad \frac{\langle \sigma,\, c\rangle \to_c \langle \sigma',\, c'\rangle}{\langle \sigma,\, C_s[c]\rangle \to_c \langle \sigma',\, C_s[c']\rangle}\ (\textsc{CtxStmt})$$

$$\frac{\langle \sigma,\, c\rangle \to_c \bot}{\langle \sigma,\, C_s[c]\rangle \to_c \bot}\ (\textsc{CtxError})$$

CtxExp and CtxStmt lift a successful step, CtxError lifts an error. The same rules hold for a condition in place of an expression, replacing $\to_e$ with $\to_b$: in the base, relational operators are binary operators like the others and there is no separate category (§2.4). By applying them repeatedly one traverses a nesting of contexts of arbitrary depth.

These three rules make all the congruence and error-propagation rules written one per construct superfluous: Seq1 (§2.5) is CtxStmt instantiated on $C_s[\bullet] = \bullet ; c$, Par-L/Par-R (§2.5) are CtxStmt instantiated on the two positions of the `par`, and CtxError makes every error-propagation rule dedicated to a single construct superfluous (§2.6). We keep the context instances written out in full where they aid readability, pointing out each time that they are instances and not independent rules.

What Kairos adds

What follows is the **contribution of this thesis**: the constructs and constraints that the base lacks and that are defined here for the first time on top of it. The base is explicit about its own limits: *"In this paper, we do not consider parameters or local variables"*[LLS24]. Kairos therefore extends that semantics on seven fronts:

1. **Local variables**: `local` / `delocal` (§2.5), absent from the base.
2. **Procedure parameters**: `call` $f(\bar{v})$ / `uncall` $f(\bar{v})$ (§2.5); the base has only `call` *id* without parameters, and passes values as a side effect on the global store.

3. **Stack**: `push` / `pop` with the *zero-cleared* convention (§2.5); the base has arrays $x[e]$, not stacks.
4. **Channels**: `ssend` / `srecv` with **labelled transitions** (§2.5).
5. **Parallel composition**: `par`...`and`...`rap`, with interleaving and PAR-COMM synchronization (§2.5).
6. **Transactions**: `try`...`rollback`...`yrt` (§2.5).
7. **Binary sessions** on channels as a dynamic property, delegation included (§2.5), with the properties that follow from it: confluence (§2.5) and reversibility extended to the `par` case (§3).

The base is sequential, but it adopts the small-step style *precisely with a view to* a concurrent extension: points 4–7 are the extension that this choice made possible.

2.4 Expressions and conditions

We adopt **without modification** the small-step semantics of the expressions of the base ([LLS24], Fig. 5): the judgment $\langle \sigma, e\rangle \to_e \langle \sigma, e'\rangle$ never touches the store and reduces constants (CONS), variables (VARS) and binary operators (BOP1–BOP3) with a left innermost strategy. As in the base, the syntax of expressions and conditions is *extended with values as leaves*, so that a partially evaluated term remains a term: this is what allows the contexts $C_e[\bullet]$ (§2.3) to lift its reduction.

Conditions are not a separate category in the base: the relational operators $(=, \neq, \geq, \leq, >, <)$ are already among its binary operators $\odot$, and therefore require no rules of their own. For readability we keep the decoration $\to_b$ for the reduction of a condition, whose values are *tt* (true) and *ff* (false), where the base uses the predicates $\mathsf{is_true}?(v)$ and $\mathsf{is_false}?(v)$ on an arbitrary value.

Since $\to_e$ and $\to_b$ are deterministic and do not modify the store, we have $\langle \sigma, e\rangle \to_e^* \langle \sigma, m\rangle$ with m unique and $\langle \sigma, b\rangle \to_b^* \langle \sigma, \beta\rangle$ with $\beta \in \{tt, ff\}$ unique; we write $eval(e, \sigma) = m$ and $eval_b(b, \sigma) = \beta$. In Kairos `push` and `pop` remain **outside** expressions (§2.5): expressions have no side effects, and their evaluation therefore remains exactly that of the base.

2.5 Semantics of commands ($\to_c$)

$\Gamma_c = \{\langle \sigma, c\rangle\} \cup \{\bot\}$, with terminal configurations $\top_c = \{\langle \sigma, \texttt{skip}\rangle\} \cup \{\bot\}$: as in the base semantics (§2.3), a *terminal* configuration is $\langle \sigma, \texttt{skip}\rangle$: there is no rule for `skip`, because $\langle \sigma, \texttt{skip}\rangle$ *is* the end of the computation.

Sequence

`skip` is the empty command and, on its own, is the terminal configuration: it has no rules. It must not be confused with ε (§2.2), which is the empty sequence of a `stack`, a *value*, not a command. In a sequence, `skip` is consumed by the rule SEQS of the base

semantics:

$$\frac{\langle \sigma,\, c_1\rangle \to_c \langle \sigma',\, c_1'\rangle}{\langle \sigma,\, c_1 \,;\, c_2\rangle \to_c \langle \sigma',\, c_1' \,;\, c_2\rangle}\ (\textsc{Seq1}) \qquad \frac{}{\langle \sigma,\, \texttt{skip} \,;\, c_2\rangle \to_c \langle \sigma,\, c_2\rangle}\ (\textsc{SeqS})$$

$$\frac{\langle \sigma,\, c_1\rangle \xrightarrow{\alpha} \langle \sigma,\, c_1'\rangle}{\langle \sigma,\, c_1 \,;\, c_2\rangle \xrightarrow{\alpha} \langle \sigma,\, c_1' \,;\, c_2\rangle}\ (\textsc{Seq-Lbl})$$

Seq1 is CtxStmt (§2.3) instantiated on the context $C_s[\bullet] = \bullet\,;\, c$: we spell it out only for readability. SeqS, instead, is not an instance of a context: it is the rule of the base semantics that consumes a `skip` that has already been reached and advances the sequence to the next command (§2.3). Seq-Lbl lifts a **labelled** step (§2.5) of c_1 into the sequence, for a generic label α: it is needed because CtxStmt lifts only unlabelled $\to_c$ steps, and it is what allows an `ssend`/`srecv` followed by other code (the normal idiom, not just a lone channel operation in the body of a procedure) to still offer itself as a partner in Par-Comm (§2.5). No error rule is needed for the sequence: an $\bot$ produced by c_1 propagates upward thanks to CtxError on the same context $\bullet\,;\, c$ (§2.6).

Reversible assignments

Constraint (checked statically): the updated variable does not appear on the right-hand side ($v \notin e$), so that the operation is injective.

$$\frac{\langle \sigma,\, e\rangle \to_e \langle \sigma,\, e'\rangle}{\langle \sigma,\, v \mathrel{+}= e\rangle \to_c \langle \sigma,\, v \mathrel{+}= e'\rangle}\ (\textsc{Add1})$$

$$\frac{}{\langle \sigma,\, v \mathrel{+}= m\rangle \to_c \langle \sigma[v \mapsto \sigma(v) + m],\, \texttt{skip}\rangle}\ (\textsc{Add2})$$

$$\frac{\langle \sigma,\, e\rangle \to_e \langle \sigma,\, e'\rangle}{\langle \sigma,\, v \mathrel{-}= e\rangle \to_c \langle \sigma,\, v \mathrel{-}= e'\rangle}\ (\textsc{Sub1})$$

$$\frac{}{\langle \sigma,\, v \mathrel{-}= m\rangle \to_c \langle \sigma[v \mapsto \sigma(v) - m],\, \texttt{skip}\rangle}\ (\textsc{Sub2})$$

$$\frac{\langle \sigma,\, e\rangle \to_e \langle \sigma,\, e'\rangle}{\langle \sigma,\, v \mathrel{\hat{}}= e\rangle \to_c \langle \sigma,\, v \mathrel{\hat{}}= e'\rangle}\ (\textsc{Xor1})$$

$$\frac{}{\langle \sigma,\, v \mathrel{\hat{}}= m\rangle \to_c \langle \sigma[v \mapsto \sigma(v) \oplus m],\, \texttt{skip}\rangle}\ (\textsc{Xor2})$$

$$\frac{}{\langle \sigma,\, v_1 \mathrel{<\!\!=\!\!>} v_2\rangle \to_c \langle \sigma[v_1 \mapsto \sigma(v_2)][v_2 \mapsto \sigma(v_1)],\, \texttt{skip}\rangle}\ (\textsc{Swap})$$

Array cells. When the target is $a[e_i]$ the rules are the same, with one extra step in front: the index is reduced to an integer before the write takes place. We write $\sigma(a)_k$ for the k-th component and $\sigma[a\langle k\rangle \mapsto m]$ for the update of cell k alone, that is, $\sigma[a \mapsto \sigma(a)[k \mapsto m]]$. We give the rules for $+\!=$; $-\!=$ and $\hat{}=$ are identical up to the operator.

$$\frac{\langle \sigma,\, e_i\rangle \to_e \langle \sigma,\, e_i'\rangle}{\langle \sigma,\, a[e_i] \mathrel{+}= e\rangle \to_c \langle \sigma,\, a[e_i'] \mathrel{+}= e\rangle}\ (\textsc{IdxL})$$

$$\frac{0 \le k < |\sigma(a)|}{\langle \sigma,\, a[k] \mathrel{+}= m\rangle \to_c \langle \sigma[a\langle k\rangle \mapsto \sigma(a)_k + m],\, \texttt{skip}\rangle}\ (\textsc{AddA})$$

$$\frac{k < 0 \ \vee \ k \geq |\sigma(a)|}{\langle \sigma,\, a[k] \mathrel{+}= m \rangle \to_c \bot} \quad (\textsc{Idx-Err})$$

An out-of-bounds index is an error and not a silent access: in a reversible language, writing outside the array would break inversion without leaving a trace, and the error would show up elsewhere. The same holds when reading: in an expression $a[e]$ the index e is evaluated first, then the cell $\sigma(a)_k$ is read, and an out-of-bounds index yields Idx-Err.

The static constraint extends accordingly: in $a[e_i] \mathrel{+}= e$ the name a may not appear in e. This is stronger than necessary (it would suffice that it not be the *same* cell), but indices are not statically decidable, and it is the same choice as in Janus. The swap $a[e_1] \mathrel{<}\!\!\Rightarrow\!\!\mathrel{>} a[e_2]$ remains allowed, provided that neither e_1 nor e_2 reads the array a: if an index depended on the swapped cells, the second swap would act on different cells and would not undo the first. With indices that do not read a the swap is an involution, and this is what makes an in-place sort writable.

Scope: `local` / `delocal`

`local` allocates v with value m; `delocal` *checks* that v still holds m and then removes it (the assertion makes the operation invertible).

$$\frac{v \notin \mathrm{dom}(\sigma)}{\langle \sigma,\, \texttt{local}\ v = m \rangle \to_c \langle \sigma[v \mapsto m],\, \texttt{skip} \rangle} \quad (\textsc{Local})$$

$$\frac{\sigma(v) = m}{\langle \sigma,\, \texttt{delocal}\ v = m \rangle \to_c \langle \sigma \setminus v,\, \texttt{skip} \rangle} \quad (\textsc{Deloc})$$

$$\frac{v \in \mathrm{dom}(\sigma)}{\langle \sigma,\, \texttt{local}\ v = m \rangle \to_c \bot} \quad (\textsc{Local-Err})$$

$$\frac{v \notin \mathrm{dom}(\sigma) \ \vee \ \sigma(v) \neq m}{\langle \sigma,\, \texttt{delocal}\ v = m \rangle \to_c \bot} \quad (\textsc{Deloc-Err})$$

For an array the declared value applies to *every* cell: opening it fills it with m, closing it checks that all cells still hold m. A single cell out of place is information that has not been given back, and the closing fails.

$$\frac{a \notin \mathrm{dom}(\sigma)}{\langle \sigma,\, \texttt{local}\ a[n] = m \rangle \to_c \langle \sigma[a \mapsto m^n],\, \texttt{skip} \rangle} \quad (\textsc{LocalA})$$

$$\frac{\sigma(a) = m^n}{\langle \sigma,\, \texttt{delocal}\ a[n] = m \rangle \to_c \langle \sigma \setminus a,\, \texttt{skip} \rangle} \quad (\textsc{DelocA})$$

where m^n is the sequence of n copies of m. The two error cases are those of Local-Err and Deloc-Err, with the addition of a mismatch on the length: closing as $a[n']$ an array opened as $a[n]$, with $n' \neq n$, is not the inverse of the opening. One subtlety specific to Deloc-Err: if $v \notin \mathrm{dom}(\sigma)$, $\sigma(v)$ is not defined and cannot be compared with m: the premise explicitly covers this case as well, not just $\sigma(v) \neq m$.

Stacks and channels are opened and closed in the same way, `local stack` s = `nil` and `local channel` c = `empty`: closing checks that they are empty. The store σ in these rules is that of a procedure *activation*, parameters included: each call has its own, so Local-Err concerns a name already present in the same activation, and the compiler detects it before execution (a `local` that redeclares a parameter or a still-open local).

Conditional (double guard)

The exit guard b' must be consistent with the branch: *true* after `then`, *false* after `else`. We adopt the four rules of the base semantics ([LLS24], Fig. 8), which do **not** use any auxiliary command `assert`: the exit guard is evaluated *in context*, in place, and the comparison between the value of the test and that of the assertion decides between a successful step and an error.

The rules therefore apply to an already reduced conditional: CtxExp (§2.3) has brought the test to a value, CtxStmt has reduced the selected branch to `skip`, and CtxExp has finally evaluated the assertion. The clauses of $C_e[\bullet]$ and $C_s[\bullet]$ for the conditional are exactly the positions used here.

$$\frac{}{\langle \sigma,\ \texttt{if}\ tt\ \texttt{then skip else}\ c_e\ \texttt{fi}\ tt\rangle \to_c \langle \sigma,\ \texttt{skip}\rangle}\ (\textsc{IfTrueEnd})$$

$$\frac{}{\langle \sigma,\ \texttt{if}\ ff\ \texttt{then}\ c_t\ \texttt{else skip fi}\ ff\rangle \to_c \langle \sigma,\ \texttt{skip}\rangle}\ (\textsc{IfFalseEnd})$$

$$\frac{}{\langle \sigma,\ \texttt{if}\ tt\ \texttt{then skip else}\ c_e\ \texttt{fi}\ ff\rangle \to_c \bot}\ (\textsc{IfError1})$$

$$\frac{}{\langle \sigma,\ \texttt{if}\ ff\ \texttt{then}\ c_t\ \texttt{else skip fi}\ tt\rangle \to_c \bot}\ (\textsc{IfError2})$$

The first two rules give the successful step when the test and the assertion have the *same* truth value (both tt, case IfTrueEnd; both ff, case IfFalseEnd); the other two signal the error when the two values *differ*. The exit assertion thus does not require any auxiliary command to be executed at the end of the branch: it is evaluated in a context position, like the entry test.[1]

Loop `from ... do ... loop ... until`

We adopt the **two-body** loop of the base semantics, with its seven rules ([LLS24], Fig. 9): b_1 is the entry assertion (true only on the first entry), c_1 the *do*-body, c_2 the *loop*-body executed *between* one iteration and the next, b_2 the exit test. The execution trace is $c_1\,[c_2\,c_1]^*$: c_1 runs k times and c_2 runs $k-1$ times.

As in the base semantics, execution uses an auxiliary run-time term, the **quadruple** (X_1, X_2, X_3, X_4), where each component is either the original term *at rest*, or a *pair* (current, original) that marks the component as being evaluated and keeps the original alongside it, so that it can be restarted at the next iteration:

$$X_1 ::= b_1 \mid (\beta, b_1) \quad X_2 ::= c_1 \mid (c, c_1) \quad X_3 ::= b_2 \mid (\beta, b_2) \quad X_4 ::= c_2 \mid (c, c_2)$$

with β a reduct of a condition. The distinction between the bare form and the pair form is what makes the rules **mutually exclusive**: e.g. LoopExp1T applies when X_2 and X_4 are bare (first entry), LoopError2 when they are pairs whose current component is `skip` (end of an iteration). The reductions inside the quadruple are lifted by the

[1] In Fig. 8 of [LLS24] the rule IfFalseEnd is printed with premise $\mathsf{is_true?}(v_2)$, which would make it identical to IfError2. This is evidently a typo: the text accompanying the figure states explicitly that the first two rules cover the case in which test and assertion are *"either both true* (IfTrueEnd *case*) *or both false* (IfFalseEnd *case*)*"*. Here we report the corrected version, with a false assertion.

contexts $C_e[\bullet]$ and $C_s[\bullet]$ of §2.3, whose clauses for the loop are exactly the positions used here.

$$\frac{}{\langle \sigma,\ \texttt{from}\ b_1\ \texttt{do}\ c_1\ \texttt{loop}\ c_2\ \texttt{until}\ b_2\rangle \to_c \langle \sigma,\ ((b_1,b_1),\ c_1,\ b_2,\ c_2)\rangle} \quad (\textsc{LoopMainS})$$

$$\frac{}{\langle \sigma,\ ((tt,b_1),\ c_1,\ b_2,\ c_2)\rangle \to_c \langle \sigma,\ (b_1,\ (c_1,c_1),\ (b_2,b_2),\ c_2)\rangle} \quad (\textsc{LoopExp1T})$$

$$\frac{}{\langle \sigma,\ (b_1,\ (\texttt{skip},c_1),\ (tt,b_2),\ c_2)\rangle \to_c \langle \sigma,\ \texttt{skip}\rangle} \quad (\textsc{LoopEndS})$$

$$\frac{}{\langle \sigma,\ (b_1,\ (\texttt{skip},c_1),\ (ff,b_2),\ c_2)\rangle \to_c \langle \sigma,\ ((b_1,b_1),\ (\texttt{skip},c_1),\ b_2,\ (c_2,c_2))\rangle} \quad (\textsc{LoopExp2F})$$

$$\frac{}{\langle \sigma,\ ((ff,b_1),\ (\texttt{skip},c_1),\ b_2,\ (\texttt{skip},c_2))\rangle \to_c \langle \sigma,\ (b_1,\ (c_1,c_1),\ (b_2,b_2),\ c_2)\rangle} \quad (\textsc{LoopExp1F})$$

$$\frac{}{\langle \sigma,\ ((ff,b_1),\ c_1,\ b_2,\ c_2)\rangle \to_c \bot} \quad (\textsc{LoopError1})$$

$$\frac{}{\langle \sigma,\ ((tt,b_1),\ (\texttt{skip},c_1),\ b_2,\ (\texttt{skip},c_2))\rangle \to_c \bot} \quad (\textsc{LoopError2})$$

The flow is therefore as follows: LoopMainS enters the loop and arms the evaluation of b_1; CtxExp brings it to a value, and LoopExp1T (which requires b_1 to be *true*) starts c_1 and arms b_2. The execution of c_1 takes place via CtxStmt; when c_1 is reduced to `skip`, CtxExp evaluates b_2: if it is true, LoopEndS terminates the loop in $\langle \sigma,\ \texttt{skip}\rangle$; if it is false, LoopExp2F starts the *loop*-body c_2 and re-arms b_1. Once c_2 has terminated, b_1 is evaluated again and must now be *false*: LoopExp1F restarts the iteration. The two error rules cover the two cases in which the assertion b_1 has the wrong value: LoopError1 when b_1 is false *on entry*, LoopError2 when b_1 becomes *true* again in the middle of the loop. These are the only two error rules of the loop: every other failure in the body propagates upward through CtxError.

The Janus loop. Kairos adopts the Janus loop *as is*, in the form and with the rules given by the base semantics[LLS24], which in turn is faithful to the original by Yokoyama and Glück[YG07]. The two clauses are optional: `from` b_1 `do` c `until` b_2 is the case $c_2 = \texttt{skip}$, and `from` b_1 `loop` c `until` b_2 the case $c_1 = \texttt{skip}$.

Call and inverse

Let `procedure` $f(\bar{p})\ \{\, c_f \,\}$. The call substitutes the formal parameters $\bar{p}$ with the actual ones $\bar{v}$ (the $\to_c$ operates on the cells of the caller). `uncall` executes the *inverted* body $\mathcal{I}(c_f)$ (operator $\mathcal{I}(\cdot)$, §3).

$$\frac{}{\langle \sigma,\ \texttt{call}\ f(\bar{v})\rangle \to_c \langle \sigma,\ c_f[\bar{v}/\bar{p}]\rangle} \quad (\textsc{Call})$$

$$\frac{}{\langle \sigma,\ \texttt{uncall}\ f(\bar{v})\rangle \to_c \langle \sigma,\ \mathcal{I}(c_f)[\bar{v}/\bar{p}]\rangle} \quad (\textsc{Uncall})$$

Stack

$\texttt{push}(v, s)$ moves the value of v onto the top of s and **clears** v to zero; $\texttt{pop}(v, s)$ does the dual (it removes the top and puts it in v), with the *zero-cleared* premise: the target must already be 0.

$$\frac{\sigma(v) = n}{\langle \sigma,\, \texttt{push}(v,s)\rangle \to_c \langle \sigma[v \mapsto 0][s \mapsto n : \sigma(s)],\, \texttt{skip}\rangle} \quad (\textsc{Push})$$

$$\frac{\sigma(v) = 0 \quad \sigma(s) = n : \ell}{\langle \sigma,\, \texttt{pop}(v,s)\rangle \to_c \langle \sigma[v \mapsto n][s \mapsto \ell],\, \texttt{skip}\rangle} \quad (\textsc{Pop})$$

$$\frac{\sigma(v) = 0 \quad \sigma(s) = \varepsilon}{\langle \sigma,\, \texttt{pop}(v,s)\rangle \to_c \bot} \quad (\textsc{Pop-Err})$$

$$\frac{\sigma(v) \neq 0}{\langle \sigma,\, \texttt{pop}(v,s)\rangle \to_c \bot} \quad (\textsc{Pop-Err2})$$

Channels: `ssend` / `srecv`

A channel ch does **not** hold values: it is a synchronization name, a rendez-vous point between the two branches that currently hold it (well-formedness constraint, below). `ssend` and `srecv` do not reduce in isolation: each produces a **labelled** step, which combines with the one of the partner branch in rule Par-Comm (§2.5) into a single $\to_c$ step of the enclosing `par`. The labels are

$$\alpha ::= ch!\langle \bar{v}, \bar{n}\rangle \mid ch?\langle \bar{w}\rangle$$

($ch!\langle \bar{v}, \bar{n}\rangle$: send offer, variables $\bar{v}$ and current values $\bar{n}$; $ch?\langle \bar{w}\rangle$: receive offer, target variables $\bar{w}$). Labelled steps do not modify the store on their own; all the movement happens in Par-Comm:

$$\frac{\sigma(v_1) = n_1 \quad \cdots \quad \sigma(v_k) = n_k}{\langle \sigma,\, \texttt{ssend}(\langle \bar{v}\rangle, ch)\rangle \xrightarrow{ch!\langle \bar{v}, \bar{n}\rangle} \langle \sigma,\, \texttt{skip}\rangle} \quad (\textsc{Ssend-Lbl})$$

$$\frac{\sigma(w_1) = 0 \quad \cdots \quad \sigma(w_k) = 0}{\langle \sigma,\, \texttt{srecv}(\langle \bar{w}\rangle, ch)\rangle \xrightarrow{ch?\langle \bar{w}\rangle} \langle \sigma,\, \texttt{skip}\rangle} \quad (\textsc{Srecv-Lbl})$$

$$\frac{\exists i.\ \sigma(w_i) \neq 0}{\langle \sigma,\, \texttt{srecv}(\langle \bar{w}\rangle, ch)\rangle \to_c \bot} \quad (\textsc{Srecv-Err})$$

Srecv-Err is not a labelled step: it is a premise on state **local** to the receiving branch ($\sigma(\bar{w}) \neq \bar{0}$), so it produces $\bot$ directly, without waiting for or depending on any partner, exactly like Pop-Err2. Ssend-Lbl has no corresponding error: reading $\sigma(\bar{v})$ has no preconditions, and it may take any value.

The rules are written for integer payloads; the implementation also admits stacks and channels, following the same scheme. A stack travels as a whole: the sender empties it and the receiver receives it, and the premise of Srecv-Lbl becomes "the destination stack is empty" (otherwise Srecv-Err), so that the received content does not get mixed with the previous one. A channel travels as a reference: the receiver becomes its holder (delegation, below).

Blocking is **emergent**, not the failure of a shared premise: if no concurrent branch offers the dual label, no rule of the whole system applies to that configuration: there is no SSEND/SRECV rule "in isolation" from which a premise could be missing; PAR-COMM (§2.5) simply finds no match. The synchronization itself *is* the shared premise.

Why no queue is needed. A channel has at most two holders (below), so at any instant there is at most one `ssend` and one `srecv` waiting, which PAR-COMM consumes together: there is never a message in transit to be kept. In the interpreter the sender deposits the message in the channel and waits for the receiver, which takes it in full: an `srecv` that receives fewer values than were sent is an error, and no value remains in the channel for the next exchange. Above all, a queue would break invertibility: `ssend` and `srecv` would act on opposite ends, and the inverse of an enqueue at the tail would not be a receive from the head. With the rendez-vous the duality $\mathcal{I}(\texttt{ssend}) = \texttt{srecv}$ (§3) holds exactly, by the same *zero-cleared* convention as `push`/`pop`, which remain an inverse pair precisely because they both touch the top.

Two guarantees, and two moments at which to verify them. Two distinct things are needed on a channel: *who* may use it, and *in what order* the two holders use it. In session type theory a single static type system provides both. Here the two are kept separate, and not for expository convenience: the first concerns a holdership that *changes during execution*, because a channel can be passed on, and no analysis of the text can follow it; the second is instead a property of the program text, and is checked before running it. Hence the two colors: holdership is guarded at run time, the protocol verified at compile time.

Binary session, as a dynamic property. A channel has, at every instant, **at most two** holders: it is a *binary session*. No fixed role (sender or receiver) is required: the two holders may swap roles over time, because with only two holders it is always clear who the partner of a given offer $ch!/ch?$ is. This is what makes the backward pairing unique (§3). In the literature on process calculi it is the linearity condition guaranteed by binary *session types* [HVK98], of which there exists a generalization to more than two participants (*multiparty session types* [HYC08]).

Holdership is **execution state**, not a property of the program text, and it may change during execution. A branch becomes a holder when it uses the channel or when it receives it as the payload of another channel, and it stops being one when it sends it to a third party or when the enclosing `par` joins its branches: holdership lasts as long as the `par`, not as long as a single branch. Two `par` in sequence on the same channel still have new holders. From this single rule follow, without special cases, the two ways in which a channel is passed on:

- *name passing*, in the style of the π-calculus: a branch creates a private channel, sends it and keeps using it to receive the reply. It gives up holdership by sending it and regains it at the next use, so the holders end up being sender and receiver.
- *delegation*: the channel is between A and B, A passes it to C and stops using it; the holders become B and C. If A reused it, it would be a third holder, and it is stopped. The classical condition of delegation, *whoever gives up the channel stops*

using it before the new holder begins, is not here a separately declared obligation: it is a consequence of counting at most two holders at a time.

The difference with respect to the *static* form of the constraint, in which a channel belongs to the same two branches for its whole life, is not one of convenience. By reading the program text one can count how many branches *name* a channel, but one cannot follow a channel that changes holder, because the new holder receives it under another name, perhaps in another procedure; static analysis is forced either to reject delegation or to exempt it from the check. When observing the execution the difficulty does not arise: the identity of a channel is the channel itself, not the name of the variable that contains it.

The constraint is not part of the transition relation: it is a condition on the execution, whose violation produces $\bot$. There are two ways to violate it, and they correspond to the two errors of §2.6: a third branch that touches a channel already engaged, and *deadlock*, that is, a branch that waits for a partner that can no longer arrive.

Session types. The constraint just given says *how many* branches touch a channel, not *how* they use it: two branches that both execute an `ssend` satisfy it, and remain blocked forever. A **session type** describes the sequence of exchanges that a branch performs on a channel:

$$S ::= \texttt{end} \mid !k.S \mid ?k.S \mid \mu X.S \mid X$$

where k is the number of payload variables exchanged in one step. **Duality** swaps the two directions and leaves the rest unchanged:

$$\overline{\texttt{end}} = \texttt{end} \qquad \overline{!k.S} = ?k.\overline{S} \qquad \overline{?k.S} = !k.\overline{S} \qquad \overline{\mu X.S} = \mu X.\overline{S} \qquad \overline{X} = X$$

The type of a branch is reconstructed from the program: $!k$ for an `ssend` of k values, $?k$ for an `srecv`, concatenation for sequencing, $\mu X.S.X$ for a loop whose body has type S, and the type of the body for a procedure call, with the channel substituted for the formal parameter. Two branches use a channel correctly when their types are dual, and duality is **invariant under inversion**: inverting a `par` swaps `ssend` and `srecv` in both branches (§3), that is, it dualizes both types, and two dual types remain dual.

This check neither replaces nor duplicates the dynamic one: it detects in advance the conflicts already visible in the text, such as two branches that offer the same direction or exchange different numbers of values, which would otherwise show up only as deadlock once execution has started. The fragment is minimal and should be taken for what it is: there is no *labelled choice*, so the type of a channel is determined only if the two branches of an `if` that communicates perform the same actions on it, and there are no *multiparty* sessions. Where the fragment does not suffice to describe the use of a channel (different branches of an `if`, a loop with a second body, an `uncall`, recursion), the type remains undetermined and nothing is reported: only provable conflicts are detected, never rejecting a correct program because the description is insufficient.

Parallel composition: `par ... and ... rap`

The branches proceed by nondeterministic *interleaving* (two-branch case; with more `and` it generalizes). A step advances either the left or the right branch:

$$\frac{\langle \sigma,\, c_1\rangle \to_c \langle \sigma',\, c_1'\rangle}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma',\, \texttt{par}\ c_1'\ \texttt{and}\ c_2\ \texttt{rap}\rangle} \quad \text{(Par-L)}$$

$$\frac{\langle \sigma,\, c_2\rangle \to_c \langle \sigma',\, c_2'\rangle}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma',\, \texttt{par}\ c_1\ \texttt{and}\ c_2'\ \texttt{rap}\rangle} \quad \text{(Par-R)}$$

When a branch terminates (i.e., it has become `skip`), it is removed from the block:

$$\frac{}{\langle \sigma,\, \texttt{par skip and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma,\, c_2\rangle} \quad \text{(Par-L0)}$$

$$\frac{}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and skip rap}\rangle \to_c \langle \sigma,\, c_1\rangle} \quad \text{(Par-R0)}$$

(When only one branch is left, `par` c `rap` behaves like c.) Par-L and Par-R are CtxStmt (§2.3) instantiated on the two clauses $C_s[\bullet] =$ `par` $\bullet$ `and` c `rap` and `par` c `and` $\bullet$ `rap`: we spell them out because this is where the interleaving is visible, but they are not independent rules. Par-L0/Par-R0 are, because they concern the termination of a branch. If a branch fails, the whole block fails (an error cannot be localized to a single branch once the branches share the `par` block), but this too does not require rules of its own: the failure of a branch propagates to the whole block by CtxError on the same two clauses (§2.6).

Par-L/Par-R (and the variants -L0/-R0) lift an **ordinary** (unlabelled) $\to_c$ step of a single branch. An `ssend`/`srecv` never produces a $\to_c$ step by itself (§2.5), so these rules never lift it directly; but if the branch that contains it is itself a nested `par`, the offer must be able to travel up to the level where its partner is. Two twin rules act as intermediaries, lifting a **labelled** step instead of an ordinary one:

$$\frac{\langle \sigma,\, c_1\rangle \xrightarrow{\alpha} \langle \sigma,\, c_1'\rangle}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \xrightarrow{\alpha} \langle \sigma,\, \texttt{par}\ c_1'\ \texttt{and}\ c_2\ \texttt{rap}\rangle} \quad \text{(Par-L-Lbl)}$$

$$\frac{\langle \sigma,\, c_2\rangle \xrightarrow{\alpha} \langle \sigma,\, c_2'\rangle}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \xrightarrow{\alpha} \langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2'\ \texttt{rap}\rangle} \quad \text{(Par-R-Lbl)}$$

With these, a channel offer generated by Ssend-Lbl/ Srecv-Lbl at the bottom of a nested `par` travels up, one level of nesting at a time, to the first level where it finds a twin branch with the dual offer on the same channel, exactly the standard CCS style. The synchronization proper remains a single rule, applied at the level where the two offers are twins:

$$\frac{\langle \sigma,\, c_1\rangle \xrightarrow{ch!\langle \bar{v},\bar{n}\rangle} \langle \sigma,\, c_1'\rangle \qquad \langle \sigma,\, c_2\rangle \xrightarrow{ch?\langle \bar{w}\rangle} \langle \sigma,\, c_2'\rangle}{\langle \sigma,\, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma[\bar{v} \mapsto \bar{0}][\bar{w} \mapsto \bar{n}],\, \texttt{par}\ c_1'\ \texttt{and}\ c_2'\ \texttt{rap}\rangle} \quad \text{(Par-Comm)}$$

(symmetric in c_1, c_2: it also applies by swapping which of the two offers $ch!$ and which $ch?$; with more than two `and` it applies to the pair of branches that offers the dual labels). The channel ch appears only to match the two labels: it does **not** appear in the store update, which touches only $\bar{v}$ (zeroed) and $\bar{w}$ (filled with $\bar{n}$); this is the joint step described in §2.5. Par-Comm produces an ordinary (unlabelled) $\to_c$ step: once synchronized, the result must not travel up any further; it is *this* step, not the raw offer, that Par-L/ Par-R lift to the still more outer levels.

Par-L-Lbl/Par-R-Lbl do not open a wrong routing. An offer that travels up through several nested `par` could in principle meet a partner other than the intended one. This does not happen because, by the binary session constraint (§2.5), at any instant at most two branches, at any depth, hold a channel: the other offer on the same channel that the label meets is necessarily the right one.

With `par` the relation $\to_c$ is **no** longer deterministic: the interleaving order is not fixed. However, the branches **do not share memory**: each operates on a *disjoint* portion of the store (its own `int`/`stack` variables). The **only** form of communication/synchronization between branches goes through channels, via Par-Comm. Thanks to this disjointness, the different interleavings that do not touch channels produce the same final store (confluence).

Why confluence holds. Let $\langle \sigma, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma_1, \texttt{par}\ c_1'\ \texttt{and}\ c_2\ \texttt{rap}\rangle$ (via Par-L) and $\langle \sigma, \texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma_2, \texttt{par}\ c_1\ \texttt{and}\ c_2'\ \texttt{rap}\rangle$ (via Par-R) be the two steps enabled in σ, neither of them an `ssend`/`srecv` (no channel access). Since c_1 reads and writes only its own `int`/`stack` variables and c_2 only its own, and these two sets are disjoint, the update of σ performed by the step of c_1 and the one performed by the step of c_2 act on disjoint portions of the domain: they are *independent* and commute. Applying one and then the other (in either order) therefore yields the same σ_{12}, with $\langle \sigma_1, \texttt{par}\ c_1'\ \texttt{and}\ c_2\ \texttt{rap}\rangle \to_c \langle \sigma_{12}, \texttt{par}\ c_1'\ \texttt{and}\ c_2'\ \texttt{rap}\rangle$ and symmetrically from σ_2: a local confluence diamond. Extending by induction on the length of the derivation (every permutation of independent steps is a sequence of adjacent swaps of this kind) we obtain that any two interleavings that do not touch channels lead to the same final store. The clause is necessary: a Par-Comm step involves **both** branches at once (§2.5), not a step of c_1 alone lifted by Par-L independently of a step of c_2 alone; there is therefore no separate step-of-c_1 and step-of-c_2 to commute, there is only one that consumes both in a single stroke; it does not fall under the case treated here. (§3 treats the argument, analogous but not the same, for the forced backward pairing on a channel.)

Commit / rollback: `try` ... `rollback` ... `yrt`

Executes the body c forward, then evaluates the commit guard b on the resulting store: if b is *true* the body stays applied (commit); if it is *false* the body is **undone** by executing the inverse $\mathcal{I}(c)$ (§3), which restores exactly the initial store, and then the rollback body r is executed. The final condition b' plays the role of the exit assertion of `if` ... `fi`: it must be true after a commit and false after a rollback, and it is what allows the inverse of the `try` to know which of the two outcomes occurred. If it does not tell the two outcomes apart, execution stops with an error, as for the conditional.

In standard terminology the construct is a **transaction**: with an empty rollback body ($r = \texttt{skip}$) it is an *ACID* transaction (it is either applied entirely or leaves no trace), whereas with a non-empty rollback body it is a **transaction with compensation**, where r is precisely the compensating action executed after the attempt has been undone. Rollback in a concurrent setting is studied in [LMSS11;LLM+13]. We give the rules here in big-step style.

$$\frac{\langle \sigma,\, c\rangle \to_c^* \langle \sigma',\, \texttt{skip}\rangle \quad \langle \sigma',\, b\rangle \to_b \langle \sigma',\, tt\rangle \quad \langle \sigma',\, b'\rangle \to_b \langle \sigma',\, tt\rangle}{\langle \sigma,\, \texttt{try}\ b\ c\ \texttt{rollback}\ r\ \texttt{yrt}\ b'\rangle \to_c \langle \sigma',\, \texttt{skip}\rangle} \quad \text{(Try-Commit)}$$

$$\frac{\begin{array}{c}\langle \sigma,\, c\rangle \to_c^* \langle \sigma',\, \texttt{skip}\rangle \quad \langle \sigma',\, b\rangle \to_b \langle \sigma',\, ff\rangle \\ \langle \sigma',\, \mathcal{I}(c)\rangle \to_c^* \langle \sigma,\, \texttt{skip}\rangle \quad \langle \sigma,\, r\rangle \to_c^* \langle \sigma'',\, \texttt{skip}\rangle \\ \langle \sigma'',\, b'\rangle \to_b \langle \sigma'',\, ff\rangle\end{array}}{\langle \sigma,\, \texttt{try}\ b\ c\ \texttt{rollback}\ r\ \texttt{yrt}\ b'\rangle \to_c \langle \sigma'',\, \texttt{skip}\rangle} \quad \text{(Try-Rollback)}$$

$$\frac{\langle \sigma,\, c\rangle \to_c^* \bot}{\langle \sigma,\, \texttt{try}\ b\ c\ \texttt{rollback}\ r\ \texttt{yrt}\ b'\rangle \to_c \bot} \quad \text{(Try-Err)}$$

$$\frac{\begin{array}{c}\langle \sigma,\, c\rangle \to_c^* \langle \sigma',\, \texttt{skip}\rangle \\ \langle \sigma',\, b\rangle \to_b \langle \sigma',\, tt\rangle \quad \langle \sigma',\, b'\rangle \to_b \langle \sigma',\, ff\rangle\end{array}}{\langle \sigma,\, \texttt{try}\ b\ c\ \texttt{rollback}\ r\ \texttt{yrt}\ b'\rangle \to_c \bot} \quad \text{(TryError1)}$$

$$\frac{\begin{array}{c}\langle \sigma,\, c\rangle \to_c^* \langle \sigma',\, \texttt{skip}\rangle \quad \langle \sigma',\, b\rangle \to_b \langle \sigma',\, ff\rangle \\ \langle \sigma',\, \mathcal{I}(c)\rangle \to_c^* \langle \sigma,\, \texttt{skip}\rangle \quad \langle \sigma,\, r\rangle \to_c^* \langle \sigma'',\, \texttt{skip}\rangle \\ \langle \sigma'',\, b'\rangle \to_b \langle \sigma'',\, tt\rangle\end{array}}{\langle \sigma,\, \texttt{try}\ b\ c\ \texttt{rollback}\ r\ \texttt{yrt}\ b'\rangle \to_c \bot} \quad \text{(TryError2)}$$

The premise $\langle \sigma',\, \mathcal{I}(c)\rangle \to_c^* \sigma$ is guaranteed by the reversibility property (§3): since $\langle \sigma,\, c\rangle \to_c^* \sigma'$, the inverse of the body restores the *pre-body* store σ. Without a `rollback` clause we set $r = \texttt{skip}$ (undo only). This is **native backtracking**: restoring the state is the exact inverse of the attempt, with no copies or snapshots. Try-Err lets an error of the body escape from the `try` unchanged: an $\bot$ is a program bug, not a failure of the guard b, and therefore does not trigger the rollback. TryError1 and TryError2 are the errors of the assertion b', the same as IfError1 and IfError2 of the conditional into which the construct is rewritten.

Well-formedness constraint: no channels in the body. The premises $\langle \sigma,\, c\rangle \to_c^* \sigma'$ and $\langle \sigma',\, \mathcal{I}(c)\rangle \to_c^* \sigma$ use the **unlabelled** relation: however, an `ssend`/`srecv` inside c produces only a labelled step (§2.5), and the only rule that consumes it, Par-Comm (§2.5), lives at the level of a `par`, above c, not inside it. If c (or r) contains an `ssend`/`srecv`, either directly or reached through `call`/`uncall`, $\langle \sigma,\, c\rangle \to_c^* \sigma'$ is not derivable: the `try` remains blocked, it does not fail with $\bot$. A well-formed program therefore avoids `ssend`/`srecv` in the body and in the rollback of every `try`, directly or through `call`/`uncall` (statically checked). This is not an arbitrary limitation: undoing a receive that has already happened would require making the sender branch go back as well, which the `try` does not control. This is the well-known difficulty of rollback in a concurrent setting[LMSS11;LLM+13].

Implementation (syntactic sugar): the construct is rewritten with `if` and `call`/`uncall` on synthetic procedures f_c and f_r with bodies c and r (parameters = free variables):

$$\texttt{call}\ f_c(\bar{v})\,;\ \texttt{if}\ b\ \texttt{then skip else uncall}\ f_c(\bar{v})\,;\ \texttt{call}\ f_r(\bar{w})\ \texttt{fi}\ b'.$$

The entry guard of the `if` is the commit condition b, evaluated after the body; the exit assertion is b'.

2.6 Run-time errors

The error outcome is $\bot$, inherited from the base together with the rule that raises it: $\Gamma'_c = \Gamma_c \cup \{\bot\}$, $\top'_c = \top_c \cup \{\bot\}$. **Propagation** is entirely the responsibility of the base: CtxError (§2.3) lets an $\bot$ bubble up from any context position $C_s[\bullet]$: through a sequence, the branches of a conditional, the bodies of a loop and the branches of a `par`. No construct therefore needs its own propagation rule. One exception remains, Try-Err (§2.5): the body of a `try` is not a context hole, because its rules reduce it with $\rightarrow_c^*$ in a premise and must keep it intact in order to execute its inverse $\mathcal{I}(c)$ in case of rollback. There an explicit error rule is still needed.

What remains specific to Kairos is **which** constructs can fail *locally*. These are the constructs that Kairos adds to the base (§2.3), and each has its own rule that produces $\bot$ instead of remaining blocked, because no other branch could ever satisfy that constraint, since it concerns state that is **local** to the branch (§2.5): Local-Err ($v \in \mathrm{dom}(\sigma)$) and Deloc-Err ($v \notin \mathrm{dom}(\sigma)$ or $\sigma(v) \neq m$) for the scope (§2.5); Pop-Err (empty stack) and Pop-Err2 ($\sigma(v) \neq 0$) for the stack (§2.5); Srecv-Err ($\sigma(\bar{w}) \neq \bar{0}$) for receiving on a channel (§2.5). The errors of the *inherited* constructs, that is, the inconsistent assertion in the conditional (IfError1/IfError2) and in the loop (LoopError1/LoopError2), come instead from the base and are not discussed here.

$\bot$ versus blocking. This distinction is specific to Kairos, because it is concurrency that makes it possible. Synchronization blocking on channels (§2.5) is not the failure of a premise: it is the absence of a ready partner branch to combine with in Par-Comm (§2.5): no rule applies until the partner is ready, but a parallel branch may still offer itself at a later time, hence **blocking**, not $\bot$. The criterion is therefore: if the violated condition concerns state local to the branch, no evolution of the environment can ever make it true and the outcome is $\bot$; if it concerns the availability of a partner, the outcome is blocking. This is why Srecv-Err produces $\bot$ and not blocking: its premise $\sigma(\bar{w}) \neq \bar{0}$ concerns the target variables of the receiver, not the channel.

Violations of the binary session. The constraint of §2.5 is a condition on execution, and it can be violated in two ways, both with outcome $\bot$. The first is a **third holder**: a branch touches a channel that has already two holders in the current `par`, even if the branch of one of them has already terminated. With three branches of the same `par` on the same channel the partner of an offer is no longer determined, and the interpreter stops immediately, whatever the order in which the branches advance, instead of succeeding or failing depending on which one terminates first. The second is **deadlock**: no branch of the program can advance any more, because each has finished, is stopped waiting on a channel, or is waiting for the branches of a nested `par`. The simplest case is an `ssend` or an `srecv` outside any `par`: there is only one branch, and the partner cannot exist. A single block remains a block, as said above, because a partner could still offer itself; but when no branch is left that could offer itself, the wait is no longer provisional and becomes $\bot$. This is the difference between not having a partner *yet* and being unable to have one.

The execution function

$$exec : Com \times Store \rightharpoonup Store \cup \{\bot\}$$

therefore remains partial (a loop may diverge, a channel may block) but explicitly signals violations of reversibility.

Chapter 3

Reversibility

Backward execution is given by the *inversion* operator $\mathcal{I}(\cdot)$, defined by induction on the structure of the command:

$$\begin{aligned}
\mathcal{I}(\texttt{skip}) &= \texttt{skip} & \mathcal{I}(c_1\,;\,c_2) &= \mathcal{I}(c_2)\,;\,\mathcal{I}(c_1) \quad \text{(reversed order)}\\
\mathcal{I}(v \mathrel{+}= e) &= v \mathrel{-}= e & \mathcal{I}(v \mathrel{-}= e) &= v \mathrel{+}= e\\
\mathcal{I}(v \mathrel{\hat{}}= e) &= v \mathrel{\hat{}}= e & \mathcal{I}(v_1 \mathrel{<\!\!=\!\!>} v_2) &= v_1 \mathrel{<\!\!=\!\!>} v_2\\
\mathcal{I}(\texttt{local}\ v = m) &= \texttt{delocal}\ v = m & \mathcal{I}(\texttt{delocal}\ v = m) &= \texttt{local}\ v = m\\
\mathcal{I}(\texttt{push}(v,s)) &= \texttt{pop}(v,s) & \mathcal{I}(\texttt{pop}(v,s)) &= \texttt{push}(v,s)\\
\mathcal{I}(\texttt{call}\ f(\bar{v})) &= \texttt{uncall}\ f(\bar{v}) & \mathcal{I}(\texttt{uncall}\ f(\bar{v})) &= \texttt{call}\ f(\bar{v})\\
\mathcal{I}(\texttt{ssend}(\langle\bar{v}\rangle, ch)) &= \texttt{srecv}(\langle\bar{v}\rangle, ch)\\
\mathcal{I}(\texttt{srecv}(\langle\bar{v}\rangle, ch)) &= \texttt{ssend}(\langle\bar{v}\rangle, ch)
\end{aligned}$$

$$\mathcal{I}(\texttt{if}\ b\ \texttt{then}\ c_t\ \texttt{else}\ c_e\ \texttt{fi}\ b') = \texttt{if}\ b'\ \texttt{then}\ \mathcal{I}(c_t)\ \texttt{else}\ \mathcal{I}(c_e)\ \texttt{fi}\ b$$

$$\mathcal{I}(\texttt{from}\ b_1\ \texttt{do}\ c_1\ \texttt{loop}\ c_2\ \texttt{until}\ b_2) = \texttt{from}\ b_2\ \texttt{do}\ \mathcal{I}(c_1)\ \texttt{loop}\ \mathcal{I}(c_2)\ \texttt{until}\ b_1$$

$$\mathcal{I}(\texttt{par}\ c_1\ \texttt{and}\ \ldots\ \texttt{and}\ c_n\ \texttt{rap}) = \texttt{par}\ \mathcal{I}(c_1)\ \texttt{and}\ \ldots\ \texttt{and}\ \mathcal{I}(c_n)\ \texttt{rap}$$

The key rewrites are the **swapping of the guards**: $b \leftrightarrow b'$ in the conditional, $b_1 \leftrightarrow b_2$ in the loop. This is what allows inverse execution to choose the right branch / know when to stop without storing the history.

The `par` is inverted branch by branch, and `ssend` and `srecv` are the inverse of one another without any auxiliary commands: reversing the order of the commands in each branch reverses the sequence of rendez-vous, and the two ends of each one remain facing each other (the proof is below). The argument relies on an assumption: that the holders of every channel are fixed by the program text. If a channel travels *inside* a message, as in $\texttt{ssend}(\langle x, ch_2\rangle, ch_1)$, whoever receives it uses it under another name, and the name alone no longer says which channel it designates. The implementation keeps the information where it is needed: when a variable of channel type is closed with `delocal`, the interpreter remembers its channel, and the `local` that reopens it backward gives back exactly that one. Thus, when the order is reversed, the receiver goes back to using ch_2 and the two ends of every rendez-vous remain facing each other even with delegation.

Reversibility property. For every well-formed command c and states σ, σ' (without $\bot$):

$$\langle \sigma,\, c\rangle \to_c^* \langle \sigma',\, \texttt{skip}\rangle \qquad \Longleftrightarrow \qquad \langle \sigma',\, \mathcal{I}(c)\rangle \to_c^* \langle \sigma,\, \texttt{skip}\rangle.$$

That is, $\mathcal{I}(c)$ computes the inverse relation of c on the store. In particular, excluding `par`, $\to_c$ is deterministic and c denotes an *injective* partial function; consequently $\langle \sigma,\, c\,;\, \mathcal{I}(c)\rangle \to_c^* \sigma$ (identity on the store).

Determinism follows from the rules themselves: for each command the premises of its rules, both success and error ones, are mutually exclusive and together cover every state, so in every configuration exactly one of them applies. For `pop`, for example, the cases $\sigma(v) \neq 0$ (Pop-Err2), $\sigma(v) = 0 \wedge \sigma(s) = \varepsilon$ (Pop-Err) and $\sigma(v) = 0 \wedge \sigma(s) = n : \ell$ (Pop) do not overlap and leave no gaps. The only exception is Par-Comm, which may not find its partner immediately: this is a wait, not a choice between two rules.

Proof (sketch). By structural induction on c.

- **Assignments.** $v \mathrel{+}= e$ and $v \mathrel{-}= e$ cancel out because $v \notin e$, so e has the same value in both directions; $\hat{}=$ and $\Leftrightarrow$ are involutions.
- **Stack and channels.** `push` and `pop` cancel out by the *zero-cleared* convention: Pop requires the target to be 0 and Push leaves it at 0. The same holds for `ssend` and `srecv`: the sender is zeroed as in Push, the receiver starts from 0 as in Pop.
- **Conditional and loop.** The double guard ensures that backward the same branch is taken and the same iterations are repeated.
- **Parallel.** Every step of a derivation of `par` c_1 `and` c_2 `rap` advances a single branch or, with Par-Comm, both together. A local step is inverted by the induction hypothesis, independently of the other branch, because the branches touch disjoint variables. A Par-Comm step is inverted with the same Par-Comm with the roles swapped: the state it produces satisfies exactly the premises of the opposite rendez-vous, which leads back to σ.

It remains to see why backward execution always pairs the right rendez-vous. This is the binary session constraint: a channel has only two holders, so every `srecv` has only one possible `ssend`, both forward and backward, and there is no third branch with which to go wrong. It is required, however, that the holders be those written in the text: if a channel travels inside a message, it acquires them at run time, and this is the case excluded above.

In all cases the execution history need not be recorded: it is avoided by the double guard for `if`/`from`, by the disjointness of variables for the branches of `par`, and by the binary session for channels.

The `try ... rollback` construct (§2.5) is not a primitive reversible command: it is a *control structure* that *uses* inversion (it executes $\mathcal{I}(c)$ to undo the failed attempt). Being sugar for `if` + `call`/`uncall`, its inverse inherits that of the conditional with calls.

3.1 Summary of the rules

Command	Effect on the store	Inverse
$v \mathrel{+}= e$	$\sigma(v) \mathrel{+}= \mathit{eval}(e)$	$v \mathrel{-}= e$
$v \mathrel{-}= e$	$\sigma(v) \mathrel{-}= \mathit{eval}(e)$	$v \mathrel{+}= e$
$v \mathrel{\hat{}}= e$	$\sigma(v) \mathrel{\oplus}= \mathit{eval}(e)$	itself
$v_1 \mathrel{<\!\!=\!\!>} v_2$	swaps v_1, v_2	itself
`local` $v = m$	allocates $v = m$	`delocal` $v = m$
`delocal` $v = m$	checks $v = m$, removes	`local` $v = m$
`if` b `...fi` b'	branch on b, asserts b'	`if` b' `...fi` b
`from` b_1 `do` c_1 `loop` c_2 `until` b_2	loop, trace $c_1[c_2\, c_1]^*$	`from` b_2 `do` $\mathcal{I}(c_1)$ `loop` $\mathcal{I}(c_2)$ `until` b_1
`call` f	executes c_f	`uncall` f
`push`(v, s)	top of s, zeroes v	`pop`(v, s)
`pop`(v, s)	checks $v = 0$, pops top into v	`push`(v, s)
`ssend`$(\langle \bar{v} \rangle, ch)$	(with partner ready) zeroes $\bar{v}$	`srecv`$(\langle \bar{v} \rangle, ch)$
`srecv`$(\langle \bar{w} \rangle, ch)$	(with partner ready) checks $\bar{w} = \bar{0}$, receives	`ssend`$(\langle \bar{w} \rangle, ch)$
`par` c_1 `and` c_2 `rap`	interleaving of c_1, c_2 (nondet.)	`par` $\mathcal{I}(c_1)$ `and` $\mathcal{I}(c_2)$ `rap`
`try` b c `rollback` r `yrt` b'	commit if b, otherwise $\mathcal{I}(c)$ then r; b' exit assertion	sugar (§2.5)

Table 3.1: The reduction rules of Kairos, grouped by construct.

The `ssend`/`srecv` rows never reduce on their own: the indicated effect takes place only jointly, in the joint synchronization Par-Comm (§2.5, §2.5).

Every premise failure produces $\bot$ (local state) or a block (shared state, channel):

Failure condition	Rule	Outcome
$v \in \mathrm{dom}(\sigma)$ (`local`)	LOCAL-ERR	$\bot$
$v \notin \mathrm{dom}(\sigma)$ or $\sigma(v) \neq m$ (`delocal`)	DELOC-ERR	$\bot$
test true, assertion false (`if`)	IFERROR1	$\bot$
test false, assertion true (`if`)	IFERROR2	$\bot$
$\sigma(v) = 0 \wedge \sigma(s) = \varepsilon$ (`pop`)	POP-ERR	$\bot$
$\sigma(v) \neq 0$ (`pop`)	POP-ERR2	$\bot$
$\sigma(\bar{w}) \neq \bar{0}$ (`srecv`)	SRECV-ERR	$\bot$
no partner branch ready (`ssend`/`srecv`)	–	block
b_1 false on entry (`from`)	LOOPERROR1	$\bot$
b_1 true mid-loop	LOOPERROR2	$\bot$
index out of bounds ($a[k]$)	IDX-ERR	$\bot$
zero divisor (e/e', $e \,\%\, e'$)	DIV-ERR	$\bot$

Table 3.2: Failure conditions and their outcome: error or block.

Arrays. Inversion does not distinguish a variable from a cell: the target is a location, and the location stays the same.

$$\mathcal{I}(a[e_i] \mathrel{+}= e) = a[e_i] \mathrel{-}= e \qquad \mathcal{I}(a[e_1] \mathrel{<\!\!=\!\!>} a[e_2]) = a[e_1] \mathrel{<\!\!=\!\!>} a[e_2]$$

$$\mathcal{I}(\texttt{local}\ a[n] = m) = \texttt{delocal}\ a[n] = m \qquad \mathcal{I}(\texttt{delocal}\ a[n] = m) = \texttt{local}\ a[n] = m$$

The index e_i is not inverted because it is not a command: it is an expression, evaluated in the same store in both directions. The condition that makes inversion correct does hold, however: between the forward command and its inverse the variables appearing in e_i must not change, and this is the same condition, already required for the right-hand expression, that the static constraint of §2.5 guarantees.

3.2 `show` and observable output

Kairos provides a construct `show`(v) that prints the value of a variable to an external stream. It is **not** part of the semantic core of the reversible machine and for this reason was omitted from the preceding rules: it serves as a tool of *external observation* (inspection of values during execution). The associated transition is simply $\langle \sigma, \texttt{show}(v) \rangle \to_c \langle \sigma, \texttt{skip} \rangle$: the store is not touched, so `show` is transparent with respect to all the properties stated in the previous sections, which speak only of the store. The form `show`(v) prints v with its name (integers, stacks, arrays and channels); the form `show`(v, `char`) prints only the character of the low byte of an integer, without a newline.

The question remains, however, of what happens to the **observable output** when a procedure is executed backward. The answer is not that the inverse of `show` is the identity: it is that `show` is an *involution*,

$$\mathcal{I}(\texttt{show}(v)) = \texttt{show}(v),$$

syntactically self-inverse like ^= and <=>, with the difference that for those the involution is also semantic, whereas `show`; `show` prints twice (§3). An `uncall` therefore does not suppress prints: it re-executes them.

In the sequential fragment (without `par`), where $\to_c$ is deterministic (§3), inversion reverses the order of the commands ($\mathcal{I}(c_1\,;\,c_2) = \mathcal{I}(c_2)\,;\,\mathcal{I}(c_1)$), so the `show`s are encountered in opposite order: if forward the procedure prints $n_1, n_2, \ldots, n_k$, backward it prints $n_k, \ldots, n_2, n_1$. This is an *observable symmetry*, the external trace that makes reversibility visible.

With `par` the symmetry is only local. Inversion acts branch by branch,

$$\mathcal{I}(\texttt{par}\ c_1\ \texttt{and}\ c_2\ \texttt{rap}) = \texttt{par}\ \mathcal{I}(c_1)\ \texttt{and}\ \mathcal{I}(c_2)\ \texttt{rap} \quad (\S3),$$

so the sequence of prints *of each branch* is reversed; but the order in which the prints of different branches are interleaved is not determined, because the interleaving is not (§2.5). Already `par show`(a) `and show`(b) `rap` may emit a, b both forward and backward. The global output is not therefore the exact reversal of the forward one: only the projection onto each single branch is.

3.3 Conformance to Janus

That Kairos contains Janus is a verifiable claim, and it is best verified on programs we did not write ourselves. The original report of the language[LD82] contains three: the integer root, a *bubble sort* with permutation and the factorization of an integer. They date from 1982, and their notation is not today's one (the comment is `;`, the swap `:`, equality `=`, inequality `#`, the remainder `\`, conjunction `&`, and the variables are global instead of parameters), but the structure is the same, and translating them is a substitution of symbols, not a rewrite. Only the procedures that read data from the terminal are left out, because Kairos has no read instruction.

The integer root runs in both directions. It computes $rt = \lfloor\sqrt{num}\rfloor$ and leaves in *num* the remainder $num - rt^2$; the `uncall` brings *rt* back to zero and *num* to its initial value. On its own it exercises a good part of what a Janus program needs: multiplication, division and remainder inside guards (§2.1), loops with only one of the two clauses `do` and `loop`, and an auxiliary procedure invoked with `call` and `uncall` within the same body.

The *bubble sort* is not invertible. This is the interesting case, because the flaw lies in the 1982 listing and not in the translation. Its conditional is

```
if list[j] > list[j+1]
then list[j] <=> list[j+1]
     perm[j] <=> perm[j+1]
fi perm[j] > perm[j+1]
```

and the exit guard must distinguish the two branches: true after `then`, false after `else` (§2.5). It does not. With $list = [4, 2, 5, 1, 3]$ and *perm* the identity, at the third outer iteration we have $list = [1, 2, 4, 3, 5]$ and $perm = [3, 1, 0, 4, 2]$: the comparison $list[3] > list[4]$ is $3 > 5$, false, so the empty branch is taken, but $perm[3] > perm[4]$

is $4 > 2$, true. The conditional has no inverse at that point, because *perm* records where each element comes from and not whether the swap has just taken place, and the two things do not coincide. This is the use case for the assertion: without it, the program would carry on and the error would appear much later, as a wrong value when a variable is closed. The file contains the whole listing: the procedure that builds the identity permutation, a valid Janus program that fills an array with $0, \ldots, n-1$, runs and prints *perm*; then *sort* stops with the assertion error exactly at the point described.

The factorization runs in both directions. It consumes *num* and leaves its prime factors, in increasing order, in an array: 840 becomes $2, 2, 2, 3, 5, 7$ and *num* goes to zero; the `uncall` recomposes 840 from the factors and zeroes the table again. It is the most complete of the three: two nested loops, an entry guard with conjunction, the swap between a cell and a variable to set aside the last prime, and the `uncall` of a procedure inside a loop to bring the trial divisor back to zero.

Chapter 4

Programming in Kairos: reversible lossless compression

4.1 The starting point

Lossless compression is the natural test bed for a reversible language: compressing and decompressing are by construction the inverse of each other, so a single reversible program implements both and there is no risk that the two halves diverge.

The work we start from is that of Lyngby et al.[LNGY24], who implemented in Janus reversible versions of two algorithms of the zip standard: LZW dictionary compression and the Burrows–Wheeler transform. Their goal was not reversibility alone but the stronger property of being *clean*, that is, of leaving no garbage: a reversible algorithm cannot erase, so auxiliary structures must be zeroed rather than discarded. The technique they use is a pair of procedures: the first produces the output together with the auxiliary data, the second recomputes the same auxiliary data starting from the output, and calling the first while undoing the second leaves only the output, with no global history.

The results they reach are clear-cut. Their LZW runs in $\Theta(n)$ like the best irreversible version, and is therefore asymptotically optimal. Their BWT, with naive sorting of the rotations, costs $O(n^3)$. They conclude that the techniques of reversible programming hold up on real problems, and that improvements remain *possible and necessary*, pointing in particular at the sorting of the rotations.

Their work is, however, entirely **sequential**, and in describing the BWT they observe that in practice the input is divided into blocks, each transformed separately. This is the observation we start from: blocks transformed separately are independent blocks, and independent blocks are branches of a `par` operating on disjoint portions of the store, that is, the hypothesis under which confluence holds (Section 2.5). In Janus that road cannot be taken, because the language has no concurrency. In Kairos it can, and this is the contribution of this chapter.

4.2 What changes when writing it in Kairos

Compression algorithms work by index, and access by index is direct: `block[j]` reads cell j without consuming it and without touching the others, as in the reference work.

```
procedure char_at(int block[], int n, int off, int k, int c)
    local int j = 0
        j += off
        j += k
        if j >= n then
            j -= n
        fi j < off
        c += block[j]
        if j < off then
            j += n
        fi j >= n
        j -= k
        j -= off
    delocal int j = 0
```

The k-th character of the rotation starting at *off* is at position $(\mathit{off} + k) \bmod n$. With the `%` operator it would be enough to write $c \mathrel{+}= \mathit{block}[(\mathit{off} + k) \bmod n]$, which is reversible because the index depends only on variables that the command does not modify. The listing instead keeps the explicit correction, because it shows how to make reversible a step that in itself loses information: j is a local variable that must be brought back to zero, so the correction $j \mathrel{-}= n$ must be undoable, and to undo it we need to know whether it took place. The exit guard $j < \mathit{off}$ tells us this (after the correction j has dropped below *off*, without the correction it has not), and the second `if`, which is the inverse of the first, uses it to put j back in place.

4.3 The lexicographic comparison

This is the point that the reference work identifies as the main difficulty, and it can be seen first-hand. Comparing two rotations means finding the first character in which they differ, but **one cannot exit early**: the exit guard of an `if` would not distinguish "I entered and have not yet found it" from "I did not enter".

The approach that works uses a witness. As long as the two rotations coincide, the counter `pref` is exactly k. The first `if` does not touch it, the second increments it when the two characters are equal: after the body it is k or $k + 1$, in both cases $\geq k$, whereas a branch that did not enter has remained below k.

```
if pref == k then // we are still in the common prefix
    if c1 > c2 then
        gt += 1
    fi gt == 1
fi pref >= k // valid exit guard
if pref == k then
    if c1 == c2 then
        pref += 1
    fi pref > k
fi pref >= k
```

An alternative formulation, accumulating the positional value in base B of the difference between the rotations, also sorts correctly and is pure accumulation, but it requires multiplying by B at every step: the value grows as B^n, and with 64-bit integers it overflows already for blocks of a few tens of characters.

4.4 Blocks in parallel

Each block is a branch, with its own structures; the only point of contact is the channel on which the branch delivers its result. The block, the rotations and the transformed column are arrays, and remain in the branch that owns them. The payload of an `ssend` can be a whole stack but not a vector, so at the end the column is copied into a stack and this travels in one go together with the index.

```
procedure worker(int block[], int offs[], int out[], stack scratch,
                 stack column, channel c, int n)
    local int idx = 0
        call sort(offs, block, scratch, n)
        call extract(offs, block, out, n, idx)
        // copy of the column into the stack that travels on the channel
        ssend(<idx, column>, c)
    delocal int idx = 0

par
    call worker(b0, offs0, out0, sc0, st0, c0, n)
and
    call worker(b1, offs1, out1, sc1, st1, c1, n)
and
    call collector(c0, c1, col0, col1, indices)
rap
```

The arrays of different branches are disjoint, so the static check on the branches of the `par` accepts them: each one is written by a single branch. The channel, which does not carry vectors, is the only point at which data change owner.

4.5 Results

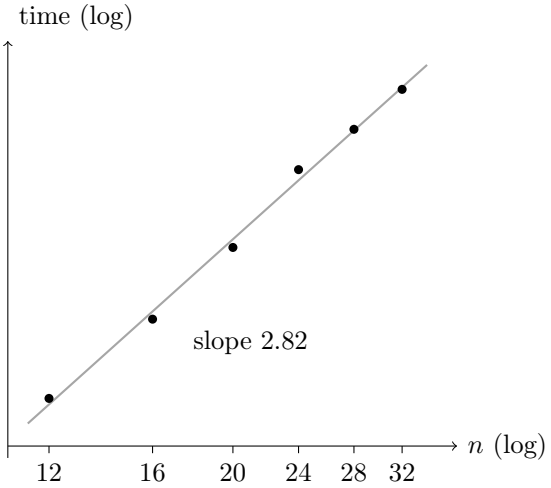
Correctness. On the test vector of the reference work, the string `banana` with a reduced alphabet ($a = 1$, $b = 2$, $n = 3$):

block	`{2, 1, 3, 1, 3, 1}`	
sorted rotations	`{5, 3, 1, 0, 4, 2}`	
last column	`{3, 3, 2, 1, 1, 1}`	i.e. `nnbaaa`
index	`3`	
reconstruction from (column, index)	`{2, 1, 3, 1, 3, 1}`	i.e. `banana`

It matches their example. The reconstruction starts from the column and the index *alone*, so it is a genuine decompression and not a mere `uncall`: it uses the classical inverse transform, which builds the vector of the preceding rows and traverses it starting from the index, recovering `banana`.

Cost. The execution time of a block, as its length n varies (for each n, the minimum of three runs):

n	time (s)
12	0.38
16	0.76
20	1.42
24	2.79
28	3.97
32	5.63



The axes are base-2 logarithmic and time is in seconds (the ordinate is $\log_2$ of the time in units of 0.25 s), so the alignment of the points says that the cost is a power law and the slope gives its exponent. The regression line of $\log t$ on $\log n$ has slope 2.82: the cost is n^3, the same as the reference work, and access by index adds nothing to the order of magnitude. The value just below 3 comes from the fixed start-up time of the interpreter, which weighs on small blocks. This is what one expects from the shape of the program: *bubble sort* performs $O(n^2)$ comparisons between rotations, and each comparison scans $O(n)$ characters, each read in constant time.

Concurrency. The same computation in two variants that differ in a single point, the procedure that orchestrates the blocks. In the first each block sits in a `par` of its own, together with only the reception of its own result: the worker and its receiver proceed together, but the blocks follow one another, and there is no parallelism *between* blocks. In the second all the workers are branches of a single `par`, with the collector as the last branch.

Sequential

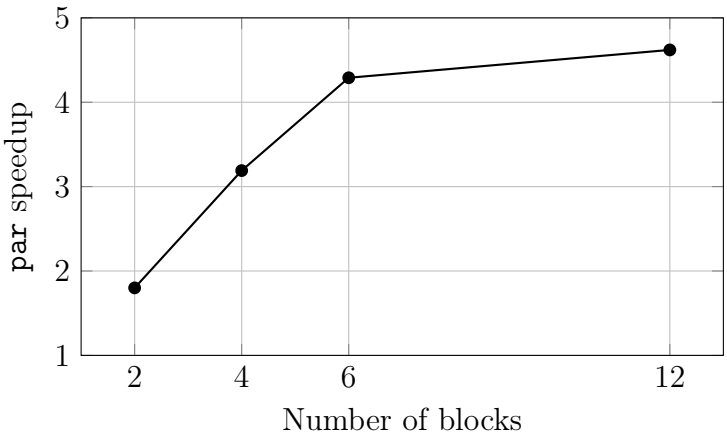
```
par
    call worker(b1, ..., c1, n)
and
    srecv(<i1, col1>, c1)
rap
par
    call worker(b2, ..., c2, n)
and
    srecv(<i2, col2>, c2)
rap
```

Parallel

```
par
    call worker(b1, ..., c1, n)
and
    call worker(b2, ..., c2, n)
and
    call collector(c1, c2, col1, col2,
        indices)
rap
```

With blocks of $n = 20$ symbols, the speedup of the parallelized version over the sequential one is the one in the table (median of three runs, ten for six blocks, on a machine with 6 physical cores and 12 hardware threads):

blocks	`par` speedup
2	1.80×
4	3.19×
6	4.29×
12	4.62×



The speedup grows almost linearly up to four blocks, where it reaches 3.2×, and then rises more slowly up to 4.6× with twelve. This means that the parallelism between blocks is real: adding blocks does not slow down the execution, and the total time grows much less than the sum of the times. The trend is not linear because the hardware does not scale indefinitely, and because there remains the part that does not parallelize, that is, the start-up of the interpreter and the collector, which receives from the channels one at a time.

4.6 Limitations

Three, stated openly. The alphabet of the tests is reduced to three symbols: nothing in the code depends on it, but nothing exercises it either. The blocks are short, imposed by the n^3 cost of the naive sorting of the rotations, which is the point on which the reference work itself states that improvements remain necessary. And the *clean* property in the strict sense, that is, zeroing the input as well and not only the auxiliary structures, would require the pair of procedures applied to the whole transform: here the auxiliaries are zeroed by undoing the procedures that produced them, while the input remains.

Chapter 5

Implementation: interpreter and DAP

5.1 The chain, from source to execution

The implementation is split into two halves with a sharp boundary. The first, written in Python, takes the source down to a textual bytecode; the second, in C, executes that bytecode in both directions.

	stage	**output**
Python	lexical analysis	token sequence
	parsing	syntax tree
	rewriting of `try`	tree without `try`
	static checks	well-formedness errors
	compiled inverses	`p__inv` procedures
	generation	textual bytecode
C	interpreter	execution, forward and backward

The boundary lies in the bytecode, a string of text that the frontend passes to the interpreter in memory. Using text is a choice made for inspectability: one can read what the machine executes, and every line carries the line number of the source it comes from, which is all the debugger needs to stop at the right place.

5.2 A step-by-step example

Consider a procedure that sums the first n integers, with the two-body loop (Section 2.5):

```
procedure sum(int n, int s)
    local int i = 0
    from i == 0 do
        s += i
    loop
        i += 1
    until i == n
    delocal int i = n
```

Lexical analysis. The first stage reduces the text to a sequence of tokens, discarding whitespace and comments. Keywords are not separate regular expressions: an identifier

is recognized and then looked up in a table of reserved words, so adding a new one costs a single line.

```
PROCEDURE ID(sum) LPAREN INT ID(n) COMMA INT ID(s) RPAREN
LOCAL INT ID(i) EQUALS NUMBER(0)
FROM ID(i) EQEQ NUMBER(0) DO
  ID(s) PLUSEQUALS ID(i)
LOOP
  ID(i) PLUSEQUALS NUMBER(1)
UNTIL ID(i) EQEQ ID(n)
DELOCAL INT ID(i) EQUALS ID(n)
```

Parsing. The grammar is LALR(1), and each production directly builds the corresponding node. The loop production clearly shows the relationship between the two:

```
statement : FROM condition opt_do opt_loop UNTIL condition

p[0] = ('from', p[2], p[3], p[6], p.lineno(1), p.lineno(5), p[4])
// b1 c1 b2 c2
```

The two line numbers, that of the `from` and that of the `until`, allow the debugger to stop at either end of the loop.

Syntax tree. Nodes are tuples, not objects; statements carry their line number (conditions do not), and the `from` carries two, followed by the `loop` body.

```
('procedure', 'sum', [('int','n'), ('int','s')],
 [('local', 'int', 'i', 0, 2),
  ('from', ('cond','==','i',0), // entry guard
           [('assign','s','+=','i',4)], // do body
           ('cond','==','i','n'), // exit guard
           3, 7,
           [('assign','i','+=',1,6)]), // loop body
  ('delocal', 'int', 'i', 'n', 8)], 1)
```

`opt_do` and `opt_loop` are the optional clauses: if they are absent, the corresponding body is empty.

Bytecode. Each line consists of a source line number, an opcode, and its arguments.

```
@1 PROC sum
@1 PARAM int n
@1 PARAM int s
@2 LOCAL int i 0
@3 EVAL i == 0 // entry guard
@3 JMPF FROM_ERR_5
@3 JMP FROM_START_5 // enter, skipping the second body
@3 LABEL FROM_BACK_5
@6 PUSHEQ i 1 // loop body
@3 LABEL FROM_START_5
@4 PUSHEQ s i // do body
@7 EVAL i == n // exit guard
@7 JMPF FROM_BACK_5 // false: another iteration, going through loop
@7 LABEL FROM_END_5
@7 LABEL FROM_ERR_5
@8 DELOCAL int i n
@1 END_PROC sum
```

The initial jump realizes the trace $c_1\,[c_2\,c_1]^*$: on first entry the second body is skipped, and control returns to it only via the back-edge. The labels have structured names (`FROM_START`, `FROM_BACK`, `FROM_END`) because they are also what inversion uses to reconstruct the loop.

The two assertions of the loop (Section 2.5), namely b_1 true on first entry (LoopError1) and false on every re-entry (LoopError2), are needed for inversion: without them the inverse `from` b_2 `do` $\mathcal{I}(c_1)$ `loop` $\mathcal{I}(c_2)$ `until` b_1 could exit after one iteration while the original performs n. This is the same choice made by Janus[YG07], and the interpreter checks both: if the entry guard is false, the jump leads to `FROM_ERR` and execution stops with LoopError1; when, after the second body, `FROM_START` is reached again, the interpreter re-evaluates b_1 and stops with LoopError2 if it is true. A loop without a `loop` clause has a single body, and the bytecode is shorter: no initial jump and no `FROM_BACK`, and a false exit guard leads directly back to `FROM_START`; LoopError2 is then checked on that jump. Likewise, the exit assertion of every conditional is checked forward after the executed branch and the entry guard backward, and `local` and `delocal` form a block within the same command sequence, as in Janus.

Inversion. `uncall` has two implementations. For a recursive procedure, even mutually recursive, the frontend applies $\mathcal{I}(\cdot)$ to the syntax tree and emits a second procedure, `p__inv`, next to `p`: `uncall` p becomes an ordinary forward call to `p__inv`, with the same loop and the same parameters. For the other procedures the interpreter inverts the body at run time: it walks through its bytecode from the last line to the first, using the line index, and executes for each instruction its inverse (`PUSHEQ` becomes `MINEQ`, `PUSH` becomes `POP`, `SSEND` becomes `SRECV`), reconstructing loops and conditionals from the labels. This is the default behavior. With the `-compiled-uncall` option the frontend emits `p__inv` also for every procedure that contains neither `par` nor channels, not even through indirect calls, and `uncall` p becomes a forward call, without reconstructing the inverse at every execution. Procedures with `par` or channels remain on the run-time path. The branch of a conditional is read from the exit assertion and the number of iterations of a loop from its guards, so here too no history of the data is needed. When, while inverting a body, the interpreter meets a call to a recursive procedure, it executes forward its compiled inverse body.

For `par` the inverse is `par` $\mathcal{I}(c_1)$ `and` $\ldots$ `and` $\mathcal{I}(c_n)$ `rap`, as in Chapter 3: the interpreter relaunches the branches, each executed backward, with `ssend` and `srecv` swapped.

5.3 Structures and execution loop

Structures. The store σ of the semantics is a `Frame`: a vector of variables, each with a type, a value and, if it is a stack or a channel, the associated structure. A configuration $\langle \sigma, c \rangle$ is the pair consisting of the current frame and a pointer to the bytecode line.

The table that translates names into indices is a hash table per frame. It is read without locks, even by the branches of a `par`: entries are only added, and when the table grows a larger copy is built and published with a single atomic write, so a reader always sees either a complete entry or none.

No structure has a size fixed in advance: neither the recursion depth, nor the number of branches of a `par`, nor the parameters of a procedure, nor the names a frame can contain, nor their length. All of them grow on demand, and the only limit is the system memory: a branch whose thread cannot be created is a reported error, not an endless wait.

Every recursive activation works on its own frame, cloned from that of the procedure with the same name index; the same holds for mutual recursion and for calls made from a branch of a `par`. A `local` in a nested activation therefore lives in the frame of that activation and does not touch the outer variable of the same name: this is the per-activation store of the semantics (§2.5). A second `local` on the same name in the *same* activation remains the error that the semantics prescribes.

Main loop. The bytecode remains text, but at startup the interpreter scans it once and builds a line index: for each line, where it begins, where it ends, where its operands begin and its instruction code. The loop no longer searches for lines nor recognizes the opcode by comparing strings: it reads the code from the index and selects the instruction with a `switch`.

```
opc = g_line_index.op[k]; /* code from the index */
switch (opc) {
case OPC_PUSHEQ: vm_x_pusheq(vm, fname, fi, d); break;
case OPC_MINEQ: vm_x_mineq (vm, fname, fi, d); break;
case OPC_XOREQ: vm_x_xoreq (vm, fname, fi, d); break;
...
```

A line executed for the first time is parsed from the text. On the second execution the interpreter decodes it and keeps the result, per thread: operands already separated and variable and label names already resolved. Expressions are compiled once into a tree, kept in a cache keyed by content, and are evaluated from that, not from the text. The labels of each procedure are collected in a first pass, together with parameters and declarations, and a jump reaches the label's line through the index, in constant time. The same loop executes the compiled inverse bodies, which are bytecode like any other; run-time inversion is handled by a second path of the interpreter that reads the same text backward. None of this changes what the program computes: the semantics of Chapter 2 remains the same.

5.4 Parallelism

A `par` block spawns one system thread per branch. The branches operate on disjoint portions of the store (Section 2.5), so the store needs no locks. The only synchronization between branches is the channel, implemented with a mutex and two wait queues, one for senders and one for receivers: whoever arrives first enqueues itself, whoever arrives second finds the partner and wakes it, and the exchange takes place. This is the rendez-vous of Par-Comm.

For the branches to be truly independent, however, they must share nothing even inside the interpreter: a service structure common to all branches would make the result depend on the order in which the threads access it. For this reason every structure that a branch uses is private to the thread, or is read without locks because it is

never modified: the tokenization state is per thread, and so are the decoded lines, the expression caches and the pool of freed integer variables; the commands written in the body of a branch use the frame of the procedure containing the `par` but each branch has its own stack of open `locals`, the check for concurrent writes resides in the individual variable and not in a global lock, and the name table is consulted without exclusion, because entries are only added and become visible when they are complete. A few global locks remain, held for the duration of a short operation and without the result depending on the order in which threads acquire them: the one that adds a name or a cloned frame to the tables, the one for the session state of the channels, the one for the channels to be restored backward and the one for the output, which keeps the line of each `show` intact even when two branches print at the same moment.

The bytecode is a single one, read by all branches. The program contains only instructions and no state: everything that changes during execution resides in the frames. The interpreter never modifies the text or its index, which the branches share read-only, without locks and without copies. What a branch derives from a line, that is, its decoding, resides in structures owned by the thread and is not shared.

5.5 The DAP debugger

The interpreter exposes a debug interface, and on top of it an extension for Visual Studio Code[1] speaks the Debug Adapter Protocol, the protocol through which the editor talks to any debugger. One sets breakpoints on the Kairos source, steps through the execution forward and backward one step at a time or up to the next breakpoint, inspects variables as they change, and reads the program output in the debug console.

The link between the two sides is what the textual bytecode makes immediate: every line carries the number of the source line it comes from (§5.3), so a breakpoint placed on the source becomes a condition on the executed lines, with no correspondence table to build and keep aligned. The state shown to the editor is the same store as the execution, serialized when the editor asks for it: not a copy kept updated in parallel, but the current frame read at that moment.

Backward movement is what distinguishes this debugger from an ordinary one, and it is not an addition to the interface: a step back executes the *inverse* of the last instruction, that is, the same direction of execution that the language possesses. The inverse of an operation on data is another operation on data, and control instructions do not touch the store, so backward they are only position. A time-travel debugger for an ordinary language must record and replay; here there is no data to record, because the information needed to go back is already in the state: the debugger keeps only the trace of the executed instructions, that is, where to return, not what to restore. The step back undoes the last instruction that acted on the store or on the output: control instructions are not stopping points, whereas a `show` is undone by printing again, because it is its own inverse (§3.2). A call that has already completed is undone as a whole, with its inverse, and the same holds for a completed `par`: the inverse of `par` c_1 `and` c_2 `rap` is the `par` of the inverses, executed by the threads as on the way forward.

[1] https://github.com/nicologiuliani6/kairos-vscode-debugger

One implementation constraint deserves mention, because it is not accidental: the step back takes place inside the interpreter loop and not in the layer that speaks the protocol. The pointer to the current instruction lives in the former, and modifying the store from the latter would leave it where it was, with position and memory telling two different stories. Crossing the boundary of a call is the case that remains out of scope: going up from the beginning of a procedure to the caller requires reconstructing the activation record, which is not part of the store but of the machine that executes it. One cannot step back inside a branch of a running `par`: an exchange on a channel has an inverse that requires the partner, and the `par` is undone only as a whole. For the same reason the step back is not available when execution has stopped inside an `uncall` in progress: there an inversion is already being executed, and it is completed by going forward.

Chapter 6

Related work

Three lines of work are close to Kairos: a concurrent extension of Janus, a reversible concurrent intermediate language, and process calculi with rollback. We compare them one at a time.

Concurrent Janus. The work closest in its goals is an earlier concurrent extension of Janus[Sam19], which adds a `fork ... and ... join` construct and message passing, and implements them with a compiler to C++ that uses POSIX threads. The starting problem is the same, and the differences lie in four choices:

- **Definition.** There, the constructs are defined by the C++ code into which they are translated; here, by reduction rules. Having a formal semantics is what allows us to state and prove reversibility and confluence as properties of the language, rather than as the observed behavior of a translation.
- **State.** There, threads share a single global state; here, the branches operate on disjoint portions of the store. This is a restriction, but it is the assumption from which the confluence of interleavings follows (§2.5), which could not be proved with shared memory.
- **Channels.** There, messages pass through queues with a maximum capacity; here, through an unbuffered rendez-vous. This is what makes `ssend` and `srecv` exact inverses of each other: on a queue they would act on opposite ends, and the inverse of an enqueue would not be a receive (§2.5).
- **Execution.** There, the program is compiled and runs like any other executable, from start to finish: it cannot be stopped halfway, nor can it go backward while it is running. Kairos, by contrast, is an interpreter, which executes one instruction at a time: the program can be stopped at any instruction, its state inspected, and from there execution can proceed forward or step back one instruction at a time. This is what the debugger offers (Section 5.5), and it is possible because every instruction has an inverse that the interpreter knows how to execute.

CRIL. Among reversible concurrent languages, the closest in construction is CRIL[OY24], which is, however, an *intermediate language*, not a language in which one writes programs. It extends Mogensen's RIL: a program is a collection of *basic blocks*, each with a single three-address instruction and labels for control flow in both

directions, and several block calls can be executed concurrently in both directions. Its correctness is expressed by *causal safety* and *causal liveness*.

	CRIL [OY24]	**Kairos**
Level	intermediate (basic blocks, three-address)	high-level, structured (Janus style)
Concurrency	concurrent block calls	explicit `par ... and ... rap`
Communication	shared variables	synchronous channels, *disjoint* stores
Correctness	causal safety / causal liveness	reversibility (§3), confluence (§2.5)

Table 6.1: Kairos and CRIL compared.

The most important difference is communication: CRIL uses shared variables, Kairos only channels between disjoint portions of the store, and confluence follows from this choice. A detailed comparison of the two correctness criteria remains to be carried out.

Transactions. For transactions, the point of comparison is the $\rho\pi$ and croll-π calculi [LMSS11;LLM+13], where backward execution is governed by explicit primitives and behaves like a transaction with compensation. There, a rollback propagates: if a process to be undone has communicated with another, the other must be undone as well, and handling this propagation is a large part of the difficulty. Kairos avoids it by forbidding communication and concurrency inside a `try ... rollback` (§2.5): a transaction remains local to a branch and there is nothing to propagate. This is a strong restriction, and it is the reason why our transactions reduce to syntactic sugar; removing it would mean tackling precisely the problem that those works solve.

Chapter 7

Conclusions and future work

We have presented Kairos, a structured imperative language that is both reversible and concurrent, and we have given its small-step structural operational semantics, defined as an extension of that of Janus by Lami, Lanese and Stefani. What Kairos adds to the base are the constructs needed by a concurrent language and by a language with data: parallel composition, synchronous communication over channels with inferred protocols, transactions with compensation, stacks, procedure parameters and local scope. We have defined the inversion operator, syntactic and computable in linear time, and shown that backward execution of a parallel block is well defined: the disjointness of the portions of the store touched by the branches gives the confluence of the interleavings that do not communicate, and the binary session constraint makes unique the pairing between a send and the receive that consumes it. That constraint is guarded as a property of the execution and not of the text, which allows delegation, that is, the passing of a channel from one holder to another. The language is realized by an interpreter, on which we have carried out a case study: the reversible Burrows–Wheeler transform, with the blocks transformed in parallel, verified on the test vector of the work it starts from, and measured.

Practical maturity. The sharpest gap is between what the theory says is possible and what a real compressor would need, and the two algorithms of the zip standard show it in opposite ways. Dictionary compression holds up: in the reference reversible version it runs in $\Theta(n)$, like the best irreversible one, so reversibility costs nothing in complexity. The transform does not: it costs $O(n^3)$, and the exponent we measure is about 2.8. There remains the problem of the naive sorting of rotations, which is the point on which the reference work itself states that improvements are possible and necessary: a better sorting is the direction we would indicate first. As long as it is missing, the blocks remain short, and with short blocks the transform lacks the material on which to produce the effect for which it exists. Moreover, the *clean* property in the strict sense is missing: the auxiliary structures are zeroed by undoing the procedures that produced them, but the input survives the output, whereas a compressor must consume it.

The language. The session discipline is partial by construction: since labelled choice is missing, the type of a channel is determined only when the two branches of an `if` that communicates perform the same actions on it; where the fragment is not enough, the

type remains undetermined. Adding choice would make total what today recognizes only the provable conflicts. Extending channels to more than two participants, instead, is not a direct step: with three holders an offer can have more than one possible partner, forward execution becomes subject to races and backward execution would no longer know which rendez-vous to undo. How to preserve a unique pairing in that case is an open question. More generally, the absence of races is here an assumption under which the properties hold, not a theorem: the compiler checks it conservatively and the interpreter reports the violations it observes, but giving it the form of a type system, separate from the calculus, is the usual way of guaranteeing it and remains to be done. Asynchronous communication is another natural extension, with the caveat that it is not a painless addition: it changes what can be proved about inversion, because a message in transit is a state that the rendez-vous does not produce.

The implementation. On the concurrent calculus, the measurement says that parallelism is achieved, with a speedup of 4.6× at twelve blocks, and points to what to improve in the part that remains sequential: the collector, which receives from the channels one at a time. A tree collector, in which the results are combined pairwise in parallel, would reduce it.

In the interpreter, the program text, with its line index, is a single one, shared among the branches. The inverse body is produced at compile time for recursive procedures; for the others the interpreter inverts the bytecode at run time, unless one asks with `-compiled-uncall` to produce it at compile time also for procedures with neither `par` nor channels. Every recursive activation, and every procedure called from a branch of `par`, works on a frame of its own cloned from that of the procedure; the commands written in the body of a branch use the frame of the procedure that contains the `par`, with a local stack of the branch's own (§5.3).

The missing comparison. In Kairos, correctness is the property of reversibility (§3): if c takes σ to σ', then $\mathcal{I}(c)$ takes σ' exactly back to σ, and confluence guarantees that the order in which the branches of a `par` alternate does not change the result. This is a guarantee on the entire execution: `uncall` undoes the whole computation. The literature on concurrent reversible languages, CRIL included, reasons instead on individual actions and uses two criteria. *Causal safety* requires that an action can be undone only after undoing all those that depend on it: a send is not undone if the receiver has already used the value. *Causal liveness* requires the opposite, that every action whose consequences have been undone can actually be undone, without getting stuck. The order of undoing therefore need not be the exact inverse of the forward one: it suffices that it respect the dependencies.

It is plausible that Kairos also satisfies these two criteria: actions of different branches do not depend on one another, because they touch disjoint portions of the store, and the only dependency between branches is the rendez-vous, which with the binary session is always undone as a pair. *This is not proved, however.* Proving it would also tell us where this property comes from: if, by allowing branches to share variables, the two criteria ceased to hold, it would be the choice of having the branches communicate only through channels that makes them true.

Reversibility and concurrency are usually studied separately: the former in program-

ming languages, the latter in process calculi, with few occasions for meeting. Kairos is an attempt to have them together in a language in which programs are written, and the case study is the check that the combination produces something: a single program that compresses and decompresses, on blocks transformed in parallel, is something that neither of the two lines of work achieves alone.

References

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems (ESOP 1998)*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[HYC08] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*, pages 273–284. ACM, 2008.

[LD82] Christopher Lutz and Howard Derby. JANUS: A time-reversible language. Technical report, California Institute of Technology, 1982. Descrizione del linguaggio e programmi di esempio, trasmessa a R. Landauer nel 1986.

[LLM+13] Ivan Lanese, Michael Lienhardt, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Concurrent flexible reversibility. In *Programming Languages and Systems (ESOP 2013)*, volume 7792 of *Lecture Notes in Computer Science*, pages 370–390. Springer, 2013.

[LLS24] Pietro Lami, Ivan Lanese, and Jean-Bernard Stefani. A small-step semantics for janus. In *Reversible Computation (RC 2024)*, volume 14680 of *Lecture Notes in Computer Science*, pages 105–123. Springer, 2024.

[LMSS11] Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Controlling reversibility in higher-order pi. In *CONCUR 2011 – Concurrency Theory*, volume 6901 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011.

[LNGY24] Therese Lyngby, Rasmus R. Nylandsted, Robert Glück, and Tetsuo Yokoyama. Towards clean reversible lossless compression: A reversible programming experiment with zip. In *Reversible Computation (RC 2024)*, volume 14680 of *Lecture Notes in Computer Science*, pages 94–102. Springer, 2024.

[LV26] Ivan Lanese and Germán Vidal. A reversible semantics for janus. *CoRR*, abs/2602.16913, 2026.

[OY24] Shunya Oguchi and Shoji Yuen. Concurrent RSSA for CRIL: Flow analysis for a concurrent reversible programming language. In *Reversible Computation (RC 2024)*, volume 14680 of *Lecture Notes in Computer Science*, pages 181–200. Springer, 2024.

[Sam19] Andrea Sammarchi. Il linguaggio janus concorrente: definizione, compilazione e sperimentazione. Tesi di laurea in informatica, Alma Mater Studiorum, Università di Bologna, 2019.

[YG07] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2007)*, pages 144–153. ACM, 2007.

Acknowledgements

[acknowledgements]