Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

H-Calculus: Session Types for Hardware Analysis and Well-Definedness

Luiz Gustavo Soares de Sá

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Dr. Ricardo Pezzuol Jacobi

Coorientador
Dr. José Edil Guimarães de Medeiros

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

H-Calculus: Session Types for Hardware Analysis and Well-Definedness

Luiz Gustavo Soares de Sá

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Brasília, 2 de Junho de 2021

Dedicatória

Dedico essa tese à minha mãe Célia Maria e à minha vó Joana.

Agradecimentos

Agradeço aos meu orientador Ricardo Jacobi e o meu co-orientador José Edil. Ambos me apoiaram e me deram a liberdade que eu precisava para tornar essa tese uma realidade. Especialmente agradeço Jacobi por me apoiar desde o curso de graduação até hoje.

Resumo

Síntese de alto nível é considerada o próximo passo lógico em design de hardware, mas os resultados, em geral, ainda não são tão bons quanto ao que a indústria necessita. Conjecturamos que a falta de uma representação de hardware adequada, criada especificamente para análise automática de hardware, é um dos principais motivos pelos quais os resultados são difíceis de otimizar. Apresentamos o cálculo-h, cálculo tipado que usa tipos de sessão temporal para bem-definição e análise de hardware. Introduzimos os conceitos principais, formalizamos suas definições, demonstramos como a análise por meio de tipos funciona, e discutimos sua utilidade na síntese de alto nível.

Palavras-chave: Design de Hardware, Sistemas de Tipos, Tipos de Sessão, Cálculo de Processos

Abstract

High-Level Synthesis has been considered the next logical step for hardware design, but results are, in general, still not as good as the industry requires. We conjecture that the lack of a proper hardware representation crafted specifically for automatic hardware analysis is one of the key reasons why results are hard to optimize. We present the h-calculus, typed calculus that uses temporal session types for hardware well-definedness and analysis. We introduce the key concepts, formalize their definitions, demonstrate how analysis through types works, and discuss its utility within High-Level Synthesis.

Keywords: Hardware Design, Type System, Session Types, Process Calculi

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introduction

1.1 The Hardware Design Challenge

The current hardware industry demands optimized designs [1, 2, 3, 4]. This demand comes from two contrasting factors: the limitations of transistor technology and the increasing demand for computational power. The limitations of digital systems come from the end of Dennard Scaling, which effectively caps transistors' maximum frequency, and the current struggle to keep up with Moore's law. The demand for more computational power comes from the growing usage of techniques, such as Machine Learning and Digital Signal Processing, that require a high amount of calculations.

For projects depending on high throughput, CPUs and GPUs are often not enough to meet efficiency constraints. That is why there is an increasing demand for custom, application-specific hardware designs. However, designing custom hardware, especially when they need to be optimal according to application-specific constraints, is, in general, an arduous task that requires long development times and significant development costs.

Most of the challenges in designing optimal custom hardware come from the need for efficient Design Space Exploration (DSE). DSE is the step in which a designer, or an automated system, explores distinct designs — verifying their correctness, analyzing their efficiency parameters, applying transformations, and comparing them analytically — until it finds an optimal, or good enough, one (Fig 1.1).

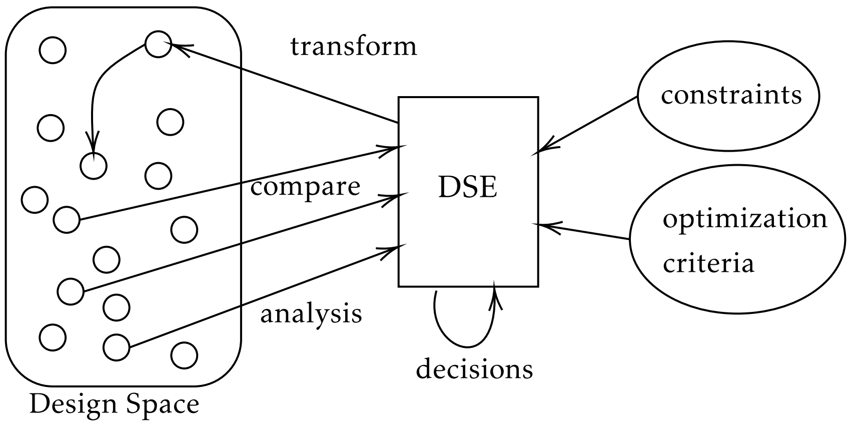


Figura 1.1: Design Space Exploration Schematics

A primary challenge in current hardware design is that there is no good automatic solution for DSE. In traditional hardware design flows, the designers are responsible for DSE. Semi-automatic tools help but do not perform DSE automatically. Humans, however, are notoriously bad at solving problems such as DSE that involve extensive search and analysis of different cases. Giving the responsibility of DSE to developers results in long, iterative, and error-prone development cycles.

1.2 High Level Synthesis

High-Level Synthesis (HLS) [5, 6, 7, 8, 9, 4, 10, 1, 2, 3] is an interesting attempt to increase automation in hardware design flows. HLS is based on the idea of transforming a high-level description of hardware, written in a *High-Level Language (HLL)*, into a low-level *Register Transfer Level (RTL)* specification. Since this transformation is the responsibility of the *HLS tool*, it moves the responsibility of DSE away from the designer, allowing for a more automated design flow (Fig. 1.2).

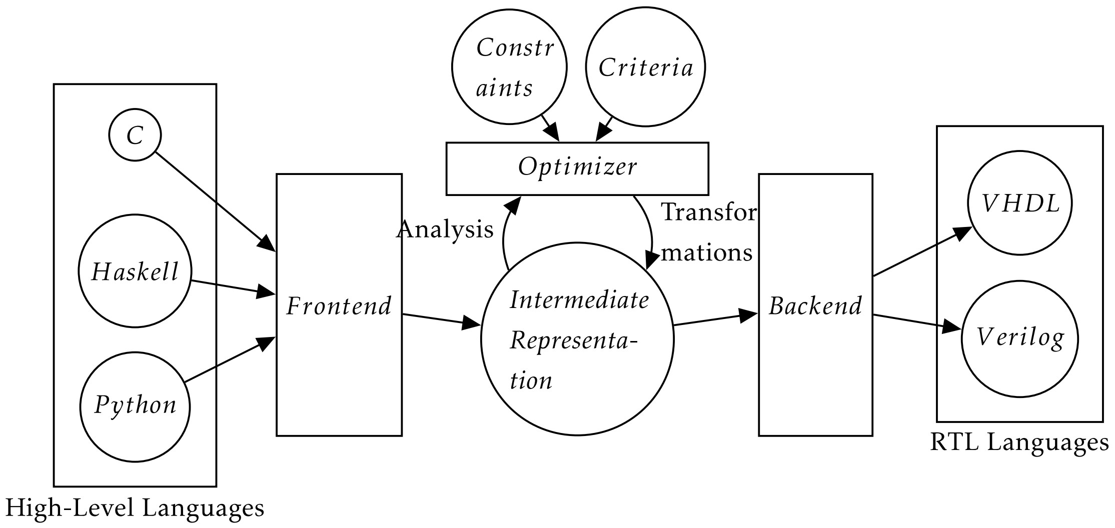


Figura 1.2: High-Level Synthesis Flow

However, HLS does not currently provide all the optimization demanded by current hardware applications [1, 2, 3, 5]. As a consequence, HLS tools do not take complete control of DSE. Instead, current HLS improves the traditional hardware design flow, focusing on being interactive rather than a complete automated solution.

Ideally, HLS should produce good enough results without the need for human interaction, but it turns out, reaching an ideal HLS poses complex challenges. HLS strategies defined as a predefined sequence of transformations (as traditional HLS theory, composed of scheduling → resource allocation→ binding → control generation [4]) are not enough to provide the efficiency demanded by the post-Dennard-scaling hardware industry. Instead, DSE is required, something that traditional HLS techniques and datatypes are not suitable.

To understand why HLS fails to implement efficient hardware, we need to understand its grounds. Conceptually, we can divide HLS into three steps (Fig. 1.3), each with a well-defined concern. The *translation* step transforms the high-level specification into an *intermediate representation (IR)* of hardware. The output of translation does not need to be efficient or optimized; it just needs to represent the computation described in the input specification correctly. The DSE step performs analysis, transformations, and comparisons on IR definitions until it finds one that meets all the design constraints. The synthesis step transforms the hardware IR into the target result, be it an HDL description (VHDL or Verilog) or a netlist.

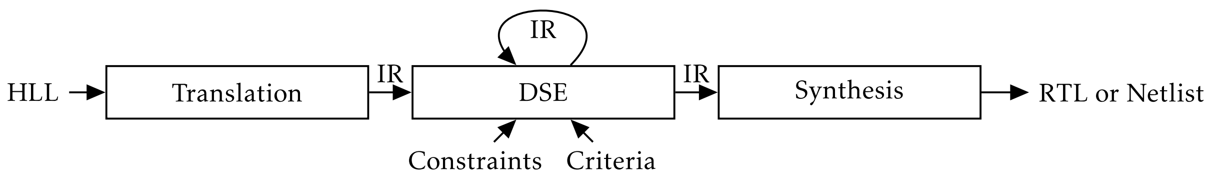


Figura 1.3: Simplified High-Level Synthesis Flow

1.3 An Ideal Intermediate Representation

Although the effectiveness of DSE depends on multiple factors — including the strategies applied and computational power (in case the strategy depends on it) — the most fundamental one is the IR choice.

The IR choice either allows or breaks the effectiveness of DSE by either enabling or disabling the feasibility of specific exploration strategies. For example, an IR without time representation does not enable an analysis (and optimization) of time-sensitive parameters; and an IR unaware of resource usage cannot optimize resource sharing effectively. Although an ideal IR is not enough for effective DSE, the effectiveness of DSE is limited without an ideal IR.

An ideal IR for hardware DSE would have the following characteristics:

1. **Has to be Correct-by-construction**, which takes away the responsibility of checking for correctness from the design exploration phase

2. **Describe low-level hardware details**, such as registers, digital signals, clock period, concurrency, and others, which allow the definition of low-level optimizations that would be performed in low-level hardware design.

3. **Includes Model system-level properties relevant for analysis and optimization**, such as resource usage, throughput, conditional branching, communication patterns, temporal behavior, among others, which enable the definition of design exploration strategies at a system-level.

4. **Should be able to fetch analysis information trivially from the model**, which will be used as input to the design exploration system.

5. **Provide efficient ways to transform models**, allowing design exploration to apply many optimizations efficiently and compare their results.

6. **Model (time-aware) communication patterns between hardware modules** so that components can be connected efficiently without the need for a communication template (e.g., FIFO buffers), used in most cases.

When analyzed through the prism of DSE suitability, most IRs used in HLS tools, such as Control Dataflow Graphs (CDFGs), do not meet many of the requirements above, making effective DSE more challenging.

This thesis introduces the h-calculus, purposely created to meet all the requirements above, thus being an ideal IR for hardware DSE. The h-calculus achieves this by using a type system that models hardware concepts. Besides helping to ensure correctness, the type system also produces detailed analytical reports of the definition encoded within the types, which allows for low-effort whole-system analysis.

By enabling effective DSE, the h-calculus aims to reduce the need for human interaction in HLS tools. Designers would then focus on the high-level specification only, making hardware design accessible for designers who understand the high-level language but do not master the details of low-level hardware design.

This thesis does not focus on any particular DSE system or implementation of an HLS system (although Chapter 5 touches upon these subjects). Instead, it focuses on the definitions, properties, and use cases of the h-calculus, demonstrating why and how it is suitable for hardware DSE and what it does differently than other frameworks.

1.4 Overview

The contributions of the thesis, and the way it is structured:

- We start by introducing the key concepts of the h-calculus in Chapter 2; we explain the process calculus, the types, and how they relate to hardware modeling through examples.
- Before discussing the details of the h-calculus, Chapter 3 explains the background necessary to understand the h-calculus. It includes discussions about type systems and the λ-calculus, commonly used hardware models of computation, and *Session Types*.
- Chapter 4 introduces h-calculus typing and semantic rules, and discusses the properties that make it a computational model for hardware.
- Chapter 5 discusses the advantages and disadvantages of using the h-calculus as an IR for HLS instead of a more traditional hardware representation. The chapter discusses translation from high-level languages to h-calculus, the effectiveness of DSE using h-calculus, and transformation from h-calculus into RTL.

- Related work is found in Chapter 6. We analyze several other models of computation in the context of hardware modeling and compare them to the h-calculus.
- Chapter 7 concludes the thesis, pointing out future work directions.

Capítulo 2

Key Concepts of the H-Calculus

This chapter introduces the fundamental concepts behind the h-calculus in an informal manner, using examples to develop intuition about the technical mechanisms of the calculus and how they relate to hardware modeling, verification, and analysis.

A simplified view of hardware architecture is a network of components that communicate through signals (Fig. 2.1). These signals carry complex data encoded as sequences of bits that change over time. Components react to input signals and produce output signals, effectively performing computation. At the lowest level, components are transistors. At the Register Transfer Level (RTL) components are logic gates (such as `AND`, `OR`, `NAND`, `NOT`) and registers (Fig. 2.2).

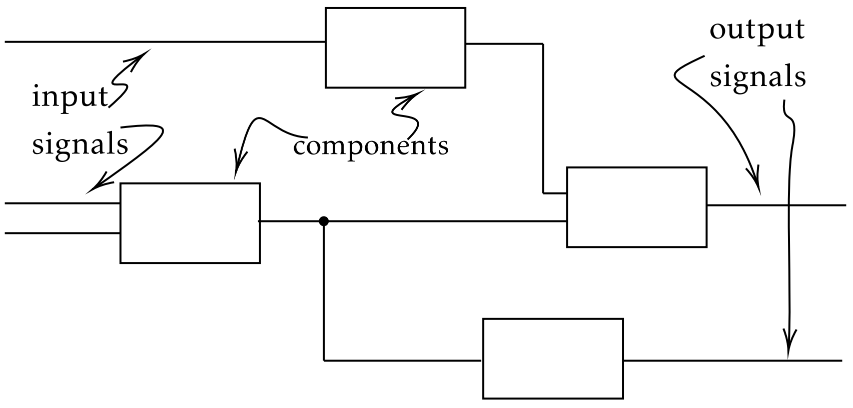


Figura 2.1: Hardware Components and Signals

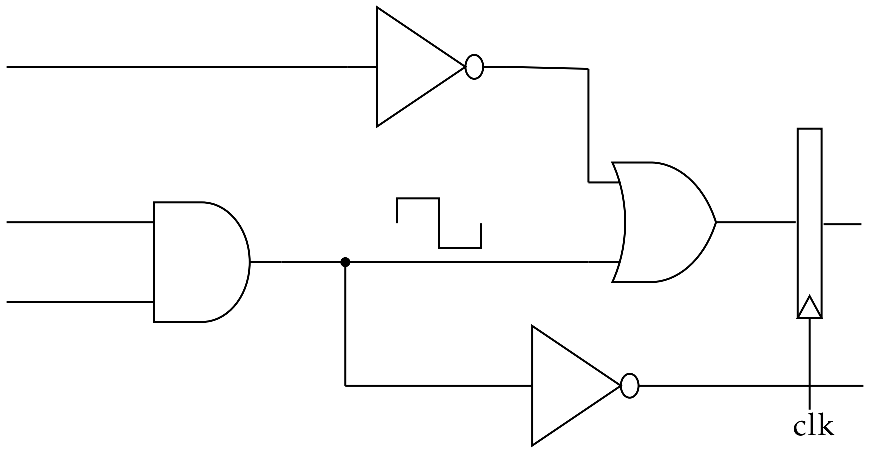


Figura 2.2: RTL components

The objective of the h-calculus is to effectively model signals and components at the RTL in a way that it is easy to spot incorrectness and performance inefficiencies. The h-calculus' approach is to model digital signals using *temporal session types* and then use them to model hardware components effectively. This kind of modeling is only possible due to the expressiveness of the type system.

2.1 Hardware Modeling with Temporal Session Types

2.1.1 Signals as Temporal Sequences

A practical way to model digital signals without losing information relevant to the analysis is to use *temporal sequences (TSs)*. Temporal sequences are (ordered) sequences of pairs, each containing a value and a temporal index indicating how long the value is stable.

Generally, $\langle v_1^{\tau_1}, v_2^{\tau_2}, v_3^{\tau_3}, \cdots \rangle$ denotes a temporal sequence that starts with value v_1, that is stable for τ_1 units of time; after that, the signal transitions into value v_2, that is stable for τ_2 units of time; and so on.

Example 2.1 (Temporal Sequence). The temporal sequence denoted $\langle 1^5, 2^5, 3^5, 4^5, 5^5 \rangle$ is graphically represented in Fig. 2.3, where the time advances from left to right.

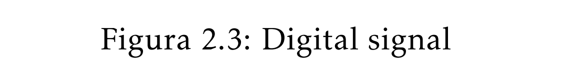


Figura 2.3: Digital signal

Since abrupt changes of values in digital signals are not natural, temporal sequences use the special value •. It represents values that are uncertain, unstable, or transitional, therefore lacking functional significance. When a component considers a • value meaningful, the result is unpredictable, resulting in incorrect behavior. It is used to model transitions between stable values, allowing for more realistic signal modeling.

Example 2.2 (Temporal Sequence with $\cdot$). The temporal sequence $\langle 1^4, \cdot^1, 2^4, \cdot^1, 3^4, \cdot^1, 4^4, \cdot^1, 5^5 \rangle$ represents the signal of Fig. 2.4.

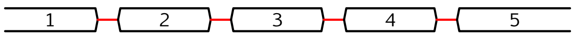


Figura 2.4: Digital signal with the transitional value (red line)

The h-calculus uses temporal sequences to describe signals between components similar to the way Kahn networks are formulated around sequences [11], the difference being that TSs keep temporal information within the model. This temporal information is crucial for hardware efficiency analysis, as we are going to see.

2.1.2 Clock Cycles and Registers

Temporal sequences are also able to model clock cycles. Clock cycles are an essential part of synchronous digital design and a crucial parameter for measuring the efficiency of hardware architectures. Within one clock cycle, an arbitrary amount of computation can occur, but for the computation results to be carried on to the next cycle (and not get lost), they have to be given to a *register* before the cycle ends.

The register is a unique hardware component with the sole purpose of saving values from a previous cycle to the next (Fig. 2.5). A clock signal, an 1-bit signal that changes from 0 to 1 and from 1 to 0 periodically, is directly connected to registers, controlling when they should and should not forward an input through the next cycle.

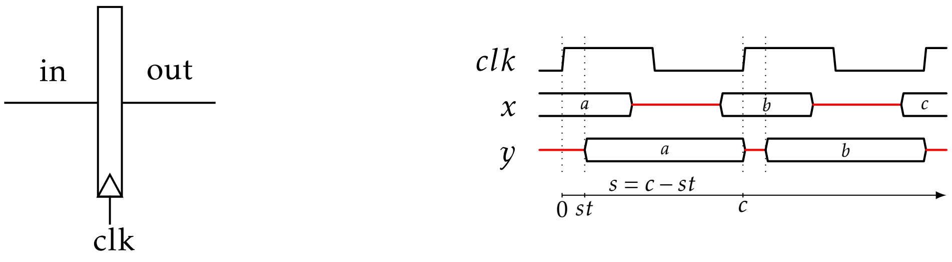


Figura 2.5: Register signal behavior and cycle dynamics: c is the **clock period**, st is the **setup time** needed for all registers to process their outputs, and s is the **stable period** in which all registers outputs are stable and therefore computation can be correctly performed

Every cycle goes as follows (Fig. 2.6): at the start of the cycle, output signals from registers carry stable values. These values are then fed to input ports of components which compute new values. These freshly computed values can be fed to other components or not; however, they must be fed to a register to be available next cycle (as the register output). Thus it is crucial that values computed within a cycle get stable before the cycle ends; otherwise, the register will possibly save a transitional value instead, resulting in incorrect computation.

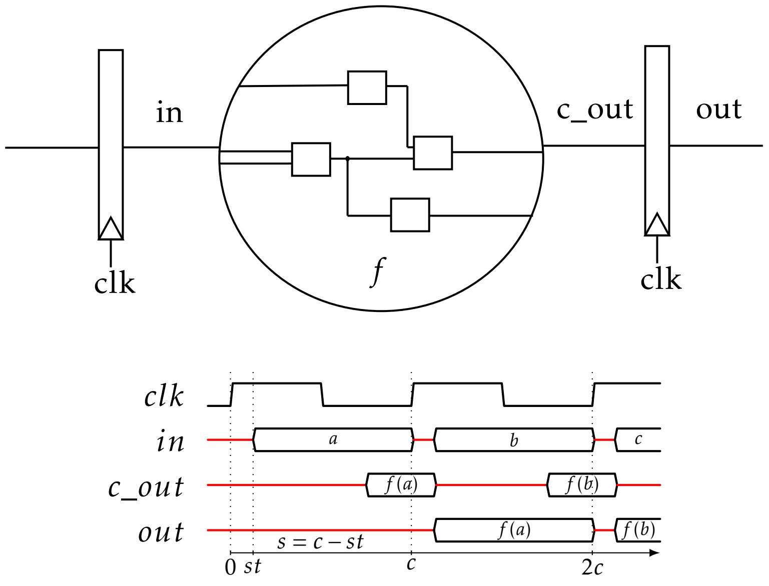


Figura 2.6: Clock period and periodic signals

Compared to asynchronous digital design, synchronous design is more straightforward, predictable, and commonly used, which is why the h-calculus focuses on synchronous design.

Synchronous circuitry can perform stateful computations using registers, but only stateless computation can be performed within one cycle.

The nature of the clock cycle dictates the temporal form of every signal within the synchronous system. Every signal has the form $\bullet^{st} A_1 \bullet^{st} A_2 \bullet^{st} A_3 \bullet^{st} A_4 \cdots$, where every A_i is a temporal sequence belonging to the ith cycle with total duration s. Because it would be cumbersome to repeatedly write $\bullet^{st}$ every cycle in every sequence, the h-calculus omits the setup time, denoting signals as $A_1A_2A_3A_4\cdots$ instead. If the real time value needs to be retrieved from the more succinct definition, it suffices to multiply the temporal duration by the number c/s.

Example 2.3 (Clock Cycle Signals). (Fig 2.7) If $c = 6$, $s = 5$ and $st = 1$, then the temporal sequence

$$\langle \bullet^1, 1^5, \bullet^2, 2^4, \bullet^1, 3^5, \bullet^2, 4^4, \bullet^1, 5^5 \rangle$$

would be denoted succinctly, removing the setup periods, as

$$\langle 1^5, \bullet^1, 2^4, 3^5, \bullet^1, 4^4, 5^5 \rangle.$$

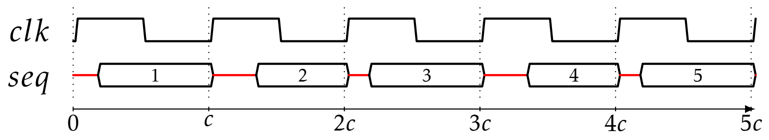


Figura 2.7: Temporal sequence example

2.1.3 Components and Temporal Session/Sequence Types

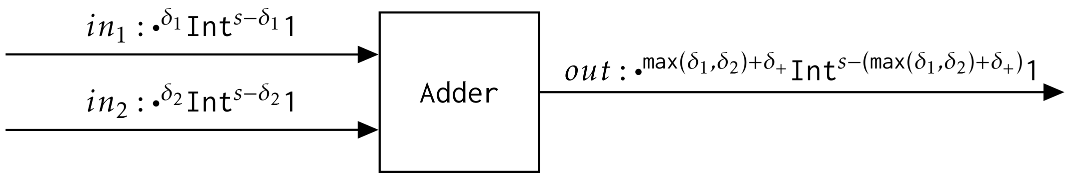
Components only work the way we expect when input signals follow a particular valid pattern; otherwise, the component outputs an undefined signal. The h-calculus captures this concept using types to model particular signal patterns and how components interact with them. For instance, the small type grammar

$$
\begin{aligned}
S \; ::= \; & T^{\delta} S && \textit{(Sending value of type T for } \delta \textit{ units of time)} \\
\mid \; & \bullet^{\delta} S && \textit{(Lack of meaningful value for } \delta \textit{ units of time (either noise or transitional values))} \\
\mid \; & 1 && \textit{(End of the signal)}
\end{aligned}
$$

, where T is a functional non-recursive type, is enough to model simple hardware signals and patterns effectively.

This typing scheme can model hardware components and verify if their usage is correct or incorrect.

Example 2.4 (Modeling an Adder). Depicted in Fig. 2.8, where δ_i is the arrival time of input i (for $i = 1$ or 2), δ_+ is the statistical worst-case processing time of the Adder and s is the duration of the stable period.



$in_1 : \bullet^{\delta_1}\mathtt{Int}^{s-\delta_1}1$
$in_2 : \bullet^{\delta_2}\mathtt{Int}^{s-\delta_2}1$
Adder
$out : \bullet^{\max(\delta_1,\delta_2)+\delta_+}\mathtt{Int}^{s-(\max(\delta_1,\delta_2)+\delta_+)}1$

Figura 2.8: Adder modeled with types

The Adders's output would be ready after all the inputs arrive plus the Adder processing time. Since the Adder is used to compute inside a cycle, the numerical values (s,δ_1,δ_2,δ_+) must be such that the output value is stable before the cycle ends ($\mathtt{max}(\delta_1, \delta_2) + \delta_+ < s$).

This typing scheme makes it easy to spot incorrect applications of components before they are synthesized or even simulated.

Example 2.5 (Verifying applications of components). Fig 10 depicts the examples above assuming $\delta_+ = 1$, the stable period is 5 units of time.

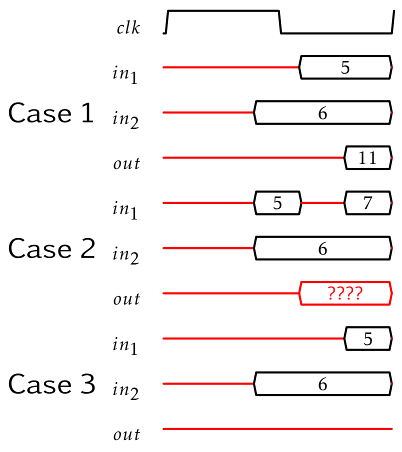


Figura 2.9: Component usage verification

Case 1 If $in_1 = \langle \bullet^3 5^2 \rangle$ and $in_2 = \langle \bullet^2 6^3 \rangle$, then input types are $in_1 : \bullet^3\mathtt{Int}^2\mathtt{1}$ and $in_2 : \bullet^2 Int^3\mathtt{1}$, the application is considered correct and the result is $out = \langle \bullet^3 \bullet^1 11^1 \rangle$ typed $out : \bullet^4 Int^1\mathtt{1}$.

Case 2 If $in_1 = \langle \bullet^2 5^1 \bullet^1 7^1 \rangle$ and $in_2 = \langle \bullet^2 6^3 \rangle$, then the input types are $in_1 : \bullet^2\mathtt{Int}^1 \bullet^1 \mathtt{Int}^1\mathtt{1}$ and $in_2 : \bullet^2\mathtt{Int}^3\mathtt{1}$. The application is then incorrect since in_1 does not follow the pattern accepted by the `Adder`. In type system terms, that would constitute a type mismatch.

Case 3 If $in_1 = \langle \bullet^4 5^1 \rangle$ and $in_2 = \langle \bullet^2 6^3 \rangle$, then input types are $in_1 : \bullet^4\mathtt{Int}^1\mathtt{1}$ and $in_2 : \bullet^2\mathtt{Int}^3\mathtt{1}$. Although the inputs follow the pattern, the result would not be ready before the clock cycle ends. In formal terms, the constraint $\mathtt{max}(4, 2) + 1 < 5$ does not hold, constituting another type mistmatch.

Furthermore, these types can also model the **usage of processes** across multiple executions and spot incorrect usages by correctly extending all the input and output types through time. This is an essential feature of the h-calculus (that will be explored in detail later this chapter) because it enables the verification and analysis of component usage and sharing.

Example 2.6 (Resource Usage Examples). Figures 2.10 and 2.11 show some examples of usage types for the `Adder` process.

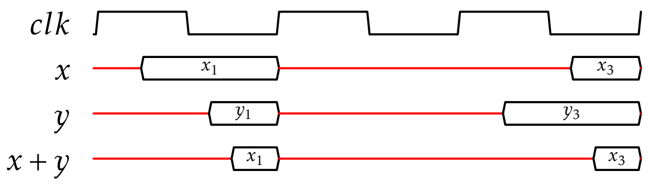


Figura 2.10: Resource Usage Example

Case 1 (Fig. 2.10) The `Adder` receives inputs in the first and third cycles but does not receive anything during the second cycle. This usage leads to correct behavior since not using components for some periods is valid.

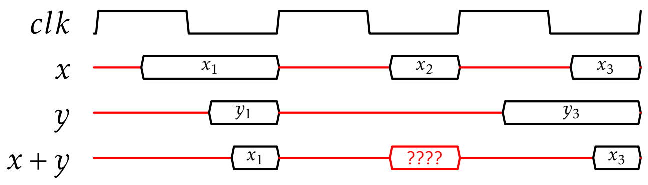


Figura 2.11: Resource Usage Example

Case 2 (Fig. 2.11) In this case, only one input is given to the component during the second cycle. This usage will result in incorrect behavior since the `Adder` requires two input values – or none at all – every time according to its type definition.

Although this typing scheme models "linear"signals well, temporal session types become powerful when they are extended with type operators — $\otimes$ (parallelism) and $\oplus$ (choice, or branching) — inspired by linear logic, and type actions — $\mu x.S\,[x]$ (recursion), $\overrightarrow{T}$ (send message), and $\overleftarrow{T}$ (receive message). Together, these additions provide the flexibility needed to accurately model complex temporal protocols, which naturally represent the way hardware modules communicate. Definition 1 shows the complete grammar for TSTs.

Definition 1 (Temporal Session Type Grammar)

$$
\begin{array}{lll}
T & ::= \textit{Functional (non-recursive) Type} & \\
x & ::= \textit{Channel variable} & \\
\tau & ::= \textit{Temporal value (Real number)} & \\
\ell & ::= \textit{Label} & \\
L & ::= \textit{Finite set of labels} & \\
S & ::= \overrightarrow{T}^{\tau} S \mid \overleftarrow{T}^{\tau} S \mid T^{\tau} S & \textit{(Write/Read/Interval value)} \\
 & \mid S \overrightarrow{\oplus}_x S \mid S \overleftarrow{\oplus}_x S & \\
 & \mid \overrightarrow{\oplus}_x \{\ell : S_\ell\}_{\ell \in L} \mid \overleftarrow{\oplus}_x \{\ell : S_\ell\}_{\ell \in L} & \\
 & \mid S \otimes S & \textit{(Parallelism)} \\
 & \mid \bullet^{\tau} S & \textit{(Delay)} \\
 & \mid \mu x.S & \textit{(Recursion)} \\
 & \mid 1 & \textit{(End of Process)}
\end{array}
$$

The operations $\overrightarrow{T}^{\tau} S$ (write a value) and $\overleftarrow{T}^{\tau} S$ (read a value) extend temporal sequences to model bidirectional channels instead of signals (it is possible to receive and send data from the same channel). This extension allows temporal session types to describe more realistic time-based communication protocols among processes.

The already introduced types $T^{\tau} S$ (internal value) and $\bullet^{\tau} S$ (transitional value) have the same meaning as before. The $T^{\tau} S$ type represents an internal value used as input/output of combinational components — such as logic gates, adders, registers, and others — that complete their execution within one clock cycle and $\bullet^{\tau} S$ means the channel does not currently carry valuable information.

The distinction between an internal value and messages exists for formal reasons: for the type system to keep being sound (as we are going to explain in full detail in Section 4) and also to allow multiple components to use the same value at the same clock cycle.

Operators $S \overrightarrow{\oplus}_x S$ (internal choice) and $S \overleftarrow{\oplus}_x S$ (external choice) and their n-ary forms $\overrightarrow{\oplus}_x \{\ell : S_\ell\}_{\ell \in L}$ and $\overleftarrow{\oplus}_x \{\ell : S_\ell\}_{\ell \in L}$ describe protocol branching based on decisions. Operationally, the choice operators either send or receive a message containing a decision from the set of all possible decisions L, which determines the next type of the channel — for example if a process is interfacing with a channel typed $\overleftarrow{\oplus}_x \{\ell : S_\ell\}_{\ell \in L}$ and it receives a message $k \in L$, the type of the channel becomes S_k for every process interacting with the channel. The channel identifier x allows the type system to know when multiple decisions are the same.

Choice operators allow complex temporal protocol modeling. $\overrightarrow{\oplus}_x$ and $\overleftarrow{\oplus}_x$ are inspired by the operators $\oplus$ and &, respectively, from linear logic and session types.

$S \otimes S$ represents parallel composition of channels inspired by the linear-logic operator with the same denotation. It allows for multiple channels to be treated as one single channel. Recursive types $\mu x.S$ allow for protocol repetition (loops), and type 1 indicates the end of a channel, meaning it will not carry valuable information anymore.

In short, temporal session types are very expressive, and they model complex temporal behavior among communicating processes. Now let us see some examples of hardware modeling using TSTs.

Example 2.7 (Modeling the clock and combinational processes). Hardware designers use the clock for time synchronization and state transitions. Because of the clock, every channel that performs an action A within a cycle follows the general temporal sequence pattern $\bullet^{\tau} A^{s-\tau} \cdots$, where s is the duration of the stable period, and τ is the instant, from 0 to s, when the action begins (Fig. 2.12). After the start of an action, it can only end after the end of the cycle because state transitions cannot occur during clock cycles. If the action starts right at the beginning of the cycle, then $\tau = 0$ and the pattern becomes $A^{s} \cdots$.

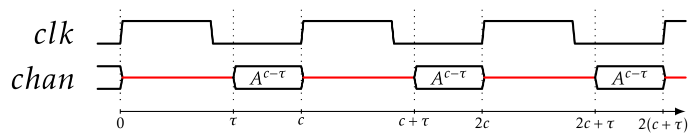


Figura 2.12: Clock and periodic actions

A combinational process gets inputs, computes, and outputs the result within the same clock period. For example, process Inc, capable of incrementing an integer, has the channel x as input and the channel y as output, shown in Fig. 2.13a. Channel x is typed $\bullet^{\tau}\overrightarrow{Int}^{s-\tau}\cdots$ and channel y is typed $\bullet^{\tau}\bullet^{\delta_{inc}}\overrightarrow{Int}^{s-(\tau+\delta_{inc})}\cdots$ with an additional delay δ_{inc} which is the time Inc takes to compute (see Fig. 2.14). Using Inc only works if $s > \tau + \delta_{inc}$.

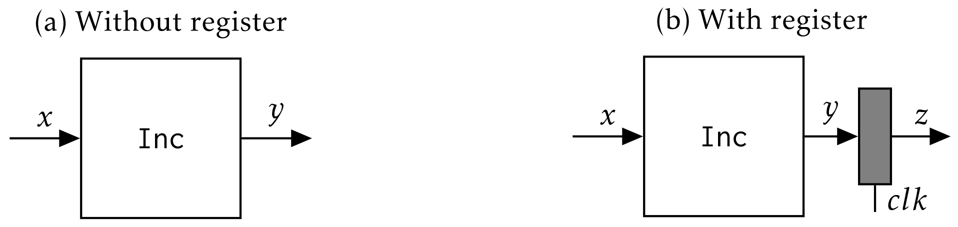


Figura 2.13: Different increment process implementations

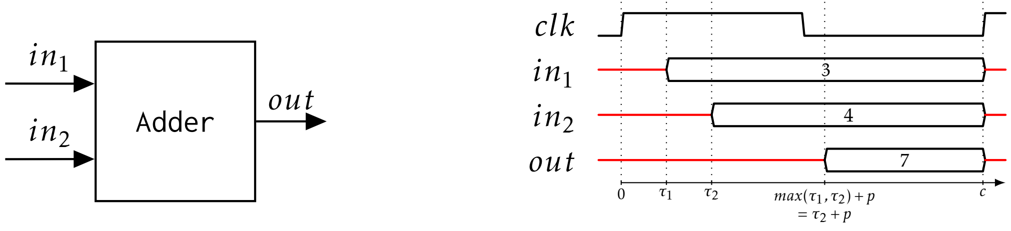


Figura 2.16: Adder process behavior

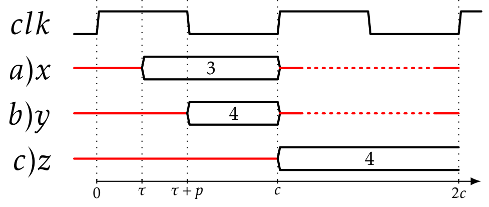


Figura 2.14: Channels interacting with INC

Let us suppose the result of Inc is fed to another process that needs almost all stable period s to complete its computation: in this case, perhaps the duration in which the Inc's result is stable, $c-(\tau+\delta_{inc})$, may be too short. The use of a register, as shown in Fig. 2.13b, solves this problem by delaying the result to the next cycle, when it is ready right at the beginning (Fig. 2.14). A register is a process with input typed $\bullet^{\tau}\overrightarrow{T}^{s-\tau}\cdots$ and output $\bullet^{s}\overrightarrow{T}^{s}\cdots$, for any $\tau < s$.

Example 2.8 (Sequential Multiplier). It is also possible to model machines that take more than one cycle to compute. While the Adder process seen in Example 4 is a combinational process that computes in one cycle, a Multiplier (Fig. 2.15), is an example of a process that might take more than one cycle to compute dependent on the implementation. Its inputs are similar to the adder ones, with $\overrightarrow{\texttt{Int}}^{\delta_i}$ instead of $\texttt{Int}^{\delta_i}$, but the output takes 8 cycles to complete. At the 8th cycle, the Multiplier outputs the result after $\delta_\times$ unit of time.

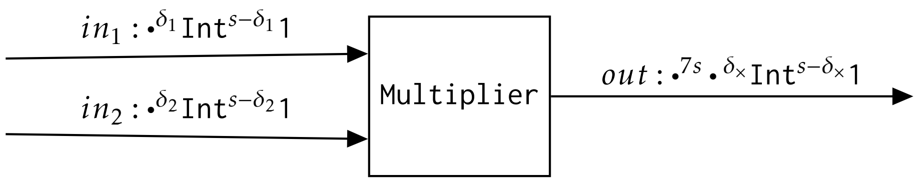

$in_1 : \bullet^{\delta_1}\texttt{Int}^{s-\delta_1}1$
$in_2 : \bullet^{\delta_2}\texttt{Int}^{s-\delta_2}1$
Multiplier
$out : \bullet^{7s}\bullet^{\delta_\times}\texttt{Int}^{s-\delta_\times}1$

Figura 2.15: Multiplier modeled with TSTs

Example 2.9 (Looping Processes - Sum). The Sum process takes a value as input every cycle and feeds it to an Adder. The result is stored in a register and fed back to the Adder next cycle. Since the register is initially set to 0, an input sequence $1,2,3,4,5$ would produce a result sequence $1,3,6,10,15$. To model the repetitive behavior of `Adder` with the type system, we define an infinite type loop using the recursive μ operator, denoted by an overline $\overline{S} = \mu x.Sx = SSSS\cdots$, which means once S ends, it starts again.

According to Fig.2.17, x_1 is typed $\overline{\bullet^k \overrightarrow{Int}^{s-k}}$, because the correct value only becomes stable after k units of time, x_2 is typed $\overline{\overrightarrow{Int}^{s}}$, ready at every start of cycle because it comes from a register (and starts already with 0), z is typed $\overline{\bullet^k \bullet^p \overrightarrow{Int}^{s-(k+\delta_+)}}$ where δ_+ is the process time of `Adder`, and out is typed $\bullet^s \overline{\overrightarrow{Int}^{s}}$ (instead of $\overline{\overrightarrow{Int}^{s}}$, because the first value equal to 0 shall not be considered part of the result).

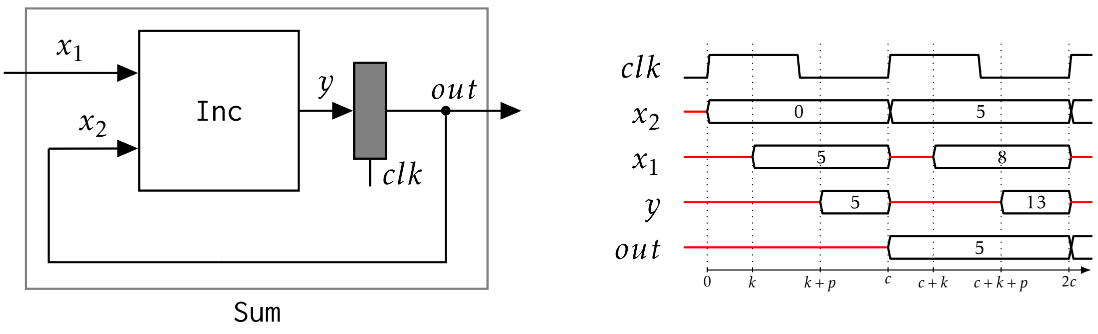


Figura 2.17: `Sum` process behavior

Example 2.10 (Arithmetic Logic Unit (ALU) implementation using choice and parallel operators). An `ALU` generally performs many different operations, depending on the value of a control signal. In this example, the `ALU` performs addition, multiplication, and *nop* (no operation). For generality, the operations have different computation timings and, therefore, temporal types: the addition result gets ready within the same cycle, while multiplication requires eight cycles to compute (Figs. 2.18 and 2.19).

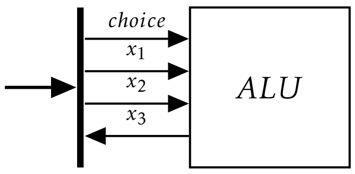


Figura 2.18: ALU channel signal decomposition

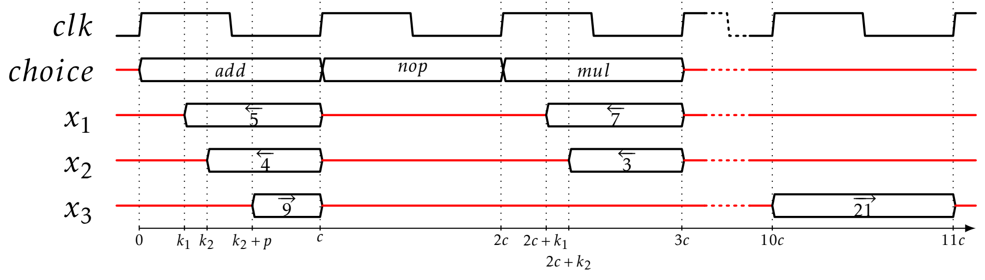


Figura 2.19: *ALU* process behavior

The input and output TSTs for ALU are defined by

$$
\begin{aligned}
S_{ALU} = \mu X.\overleftarrow{\oplus}\{ & \\
& add : \left(\bullet^{k_1}\overleftarrow{Int}^{s-k_1}\mathbf{1}\right)\otimes\left(\bullet^{k_2}\overleftarrow{Int}^{s-k_2}\mathbf{1}\right)\otimes\left(\bullet^{\max(k_1 k_2)}\bullet^{\delta_+}\overrightarrow{Int}^{s-(\max(k_1,k_2)+\delta_+)}X\right)\\
& mul : \left(\bullet^{k_1}\overleftarrow{Int}^{s-k_1}\mathbf{1}\right)\otimes\left(\bullet^{k_2}\overleftarrow{Int}^{s-k_2}\mathbf{1}\right)\otimes\left(\bullet^{8s}\overrightarrow{Int}^{s}X\right)\\
& nop : \bullet^{s}X\\
& \}
\end{aligned}
$$

The S_{ALU} type represents the protocol for correct interaction with the ALU. Any process that wants to communicate with the ALU needs to interact with this channel according to its type, or else correct communication is not possible.

The protocol starts with a recursion ($\mu X.\cdots$), so that every time the type variable X is reached it goes back to the beginning. The protocol also uses an external choice ($\overleftarrow{\oplus}$) that defines which operation the ALU must execute, and parallel channels ($\otimes$), used separately to receive/send data in parallel.

In the case of an *add* operation, 3 channels must be provided: the first 2 are input channels, in which a process must provide 2 integers with possibly different arrival times (k_1 and k_2), while the third one is the output channel, in which the result is returned after $\max(k_1,k_2)+\delta_+$ time units. Note that the first two channels end with 1 while the third channel ends with X, which means the third channel continues the protocol. The *mul* operation works similarly, because it also uses 3 channels, 2 for input and 1 for output, but the result is only ready after 8 cycles. While the computation occurs, the ALU protocol ignores inputs (this is the meaning of $\bullet$) and, after 8 cycles, the ALU outputs the result. If one choses the *nop* (no operation), the ALU becomes idle for one cycle. After the ALU computes, is goes back to its initial state, and the protocol never actually ends.

2.2 Type Merge

Sharing wires/channels among more than two components is a basic design pattern in hardware design that needs to be represented within our temporal session types. TSTs achieve sharing using the *type merge*, a simple operation on TSTs used to verify and analyse connected channels.

When a channel $c : S$ (notation for "channel c has type S") splits into n subchannels $c_1 : S_1, c_2 : S_2, \ldots, c_n : S_n$ (Fig. 2.20) we say the split is *well defined* if $S_1 \times S_2 \times \ldots \times S_n = S$, where $\times$ is the binary *merge* operation, which either returns a TST or is *undefined*. When the merging of two session types is undefined, we say that the two types are *not mergeable*, meaning they are not compatible with each other to provide correct channel behavior. We use "c split into $c_1, \cdots c_n$" and "$c_1, \cdots c_n$ merged into c" interchangeably.

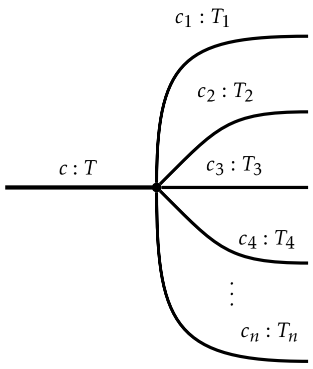


Figura 2.20: Channel splitting/merging

The formal definition of merge is shown in Definition 2. Informally, the merge is well defined if there are no simultaneous writes among the channels (no collisions), and every time there is a write, at least one of the channels reads the information (no discarding of useful data). Writes and reads are represented by the types $\overrightarrow{Int}^{\tau} \cdots$ and $\overleftarrow{Int}^{\tau} \cdots$ and the choice operators ($\overrightarrow{\oplus}_x$ and $\overleftarrow{\oplus}_x$). Furthermore, the parallel operator $\otimes$ requires all endpoints to spawn subchannels, and the end type **1** is mergeable with any other channel because $\mathbf{1} \times T = T$ for all T.

Definition 2 (Type Merge) The type merge operation, denoted $\times$, is an partial function that takes two TSTs as input and outputs another TST. Is is defined by

$$
\begin{aligned}
&(\overrightarrow{T}^k S_1) \times (\overrightarrow{T}^k S_2) = \overrightarrow{T}^k (S_1 \times S_2) && \textit{(Multiple reads)}\\
&(\overrightarrow{T}^k S_1) \times (\bullet^k S_2) = \overrightarrow{T}^k (S_1 \times S_2) && \textit{(One read)}\\
&(\bullet^k S_1) \times (\overrightarrow{T}^k S_2) = \overrightarrow{T}^k (S_1 \times S_2) && \\
&(\overleftarrow{T}^k S_1) \times (\bullet^k S_2) = \overleftarrow{T}^k (S_1 \times S_2) && \textit{(One write)}\\
&(\bullet^k S_1) \times (\overleftarrow{T}^k S_2) = \overleftarrow{T}^k (S_1 \times S_2) && \\
&(\overleftarrow{T}^k S_1) \times (\overrightarrow{T}^k S_2) = \overleftarrow{T}^k (S_1 \times S_2) && \textit{(Cross read/write)}\\
&(\overrightarrow{T}^k S_1) \times (\overleftarrow{T}^k S_2) = \overleftarrow{T}^k (S_1 \times S_2) && \\
&(S_{11} \overrightarrow{\oplus}_x S_{12}) \times (S_{21} \overrightarrow{\oplus}_x S_{22}) = (S_{11} \times S_{21}) \overrightarrow{\oplus}_x (S_{12} \times S_{22}) && \textit{(Multiple choice reads)}\\
&(S_{11} \overrightarrow{\oplus}_x S_{12}) \times S_2 = (S_{11} \times S_2) \overrightarrow{\oplus}_x (S_{12} \times S_2) && \textit{(One choice read)}\\
&S_1 \times (S_{21} \overrightarrow{\oplus}_x S_{22}) = (S_1 \times S_{21}) \overrightarrow{\oplus}_x (S_1 \times S_{22}) && \\
&(S_{11} \overleftarrow{\oplus}_x S_{12}) \times S_2 = (S_{11} \times S_2) \overleftarrow{\oplus}_x (S_{12} \times S_2) && \textit{(One choice write)}\\
&S_1 \times (S_{21} \overleftarrow{\oplus}_x S_{22}) = (S_1 \times S_{21}) \overleftarrow{\oplus}_x (S_1 \times S_{22}) && \\
&(S_{11} \overleftarrow{\oplus}_x S_{12}) \times (S_{21} \overrightarrow{\oplus}_x S_{22}) = (S_{11} \times S_{21}) \overleftarrow{\oplus}_x (S_{12} \times S_{22}) && \textit{(Cross choice read/write)}\\
&(S_{11} \overrightarrow{\oplus}_x S_{12}) \times (S_{21} \overleftarrow{\oplus}_x S_{22}) = (S_{11} \times S_{21}) \overleftarrow{\oplus}_x (S_{12} \times S_{22}) && \\
&(S_{11} \otimes S_{12}) \times (S_{21} \otimes S_{22}) = (S_{11} \times S_{21}) \otimes (S_{12} \times S_{22}) && \textit{(Parallel)}\\
&(\mu x.S_1) \times S_2 = S_1 [x/\mu x.S_1] \times S_2 && \textit{(Recursion)}\\
&S_1 \times (\mu x.S_2) = S_1 \times S_2 [x/\mu x.S_2] && \\
&(\bullet^k S_1) \times (\bullet^k S_2) = \bullet^k (S_1 \times S_2) && \textit{(Idle)}\\
&\mathbf{1} \times S = S && \textit{(End of channel)}\\
&S \times \mathbf{1} = S && \\
&S \times S' = \texttt{undefined} \ \text{otherwise} &&
\end{aligned}
$$

Now we show some examples of channel merging.

Example 2.11 (Channel Merging). Channel c is provided by process Q and used by process P, consisting of 2 parallel processes, P_1 and P_2, both with access to c (Fig. 2.21). For processes Q and P, channel c is typed T, while for P_1 and P_2, c is typed T_1 and T_2. If $T_1 \times T_2 = T$, the composition is well-typed, otherwise, it is incorrect.

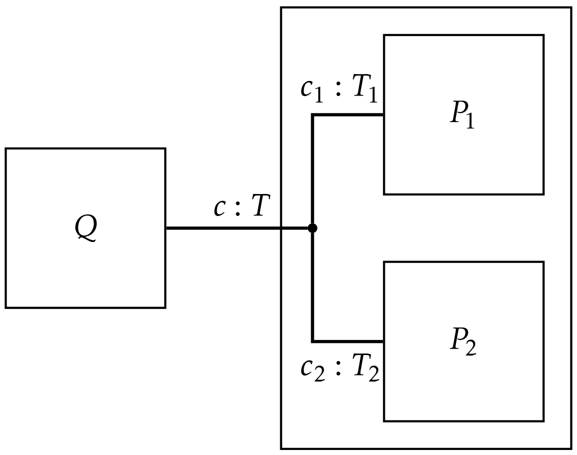


Figura 2.21: Channel splitting/merging

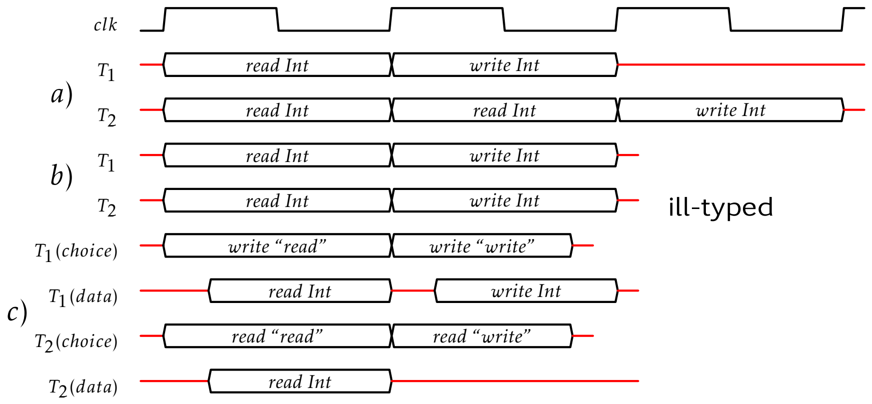


Figura 2.22: Examples of merge

Case I (Correct merging). *Say* $T_1 = \overrightarrow{Int}^5\overleftarrow{Int}^5 \bullet^5 \mathbf{1}$ *and* $T_2 = \overrightarrow{Int}^5\overrightarrow{Int}^5\overleftarrow{Int}^5\mathbf{1}$ *(Fig. 2.22a).*

- *From* 0 *to* 5*:* P_1 *and* P_2 *read an* Int *from channel*
- *From* 5 *to* 10*:* P_1 *writes an* Int *and* P_2 *reads it*
- *From* 10 *to* 15*:* P_1 *does not interact with c, while* P_2 *writes an* Int *to it*

In this case, $T_1 \times T_2$ *is well defined because all the* 3 *moments describe well-behaved situations. Simultaneous reads are allowed, simultaneous read and write are allowed and write/read while the other process does not interact with the channel is also allowed. If* T *satisfies the constraint* $T = T_1 \times T_2 = \overrightarrow{Int}^5\overleftarrow{Int}^5\overleftarrow{Int}^5\mathbf{1}$*, the system is well-typed.*

Case II (Conflicting merge). *Say* $T_1 = \overrightarrow{Int}^5\overleftarrow{Int}^5\mathbf{1}$ *and* $T_2 = \overrightarrow{Int}^5\overleftarrow{Int}^5\mathbf{1}$ *(Fig. 2.22b).*

- *From* 0 *to* 5*:* P_1 *and* P_2 *read an* Int *from channel*
- *From* 5 *to* 10*:* P_1 *and* P_2 *write an* Int *in c*

In this case, T_1 and T_2 cannot be merged because from 5 to 10, both write at the same time, which is a violation of correct channel behavior. $T_1 \times T_2 =$ undefined and the system is not well-typed regardless of the value of T.

Case III (Choice merge). *Say $T_1 = \overleftarrow{\oplus}_c\{read : \bullet^1 \overrightarrow{Int}^4\mathbf{1}, write : \bullet^1 \overleftarrow{Int}^4\mathbf{1}\}$, meaning P_1 decides (internal choice, coming from P_1 itself) whether it reads or writes a value. The 1 unit time delay is for the processes involved to compute the choice. What could be a type T_2 so that both types are mergeable? (Fig. 2.22c).*

In this case, it suffices P_2 to read the decision made by P_1 and act accordingly. A possible definition of T_2 could be $\overrightarrow{\oplus}_c\{read : \bullet^1 \overrightarrow{Int}^4\mathbf{1}, write : \bullet^5\mathbf{1}\}$, that is, P_2 reads the choice made by P_1 (external choice) and acts so that each branch is independently mergeable. In this case, $T = T_1 \times T_2 = \overleftarrow{\oplus}_c\{read : \bullet^1 \overrightarrow{Int}^4\mathbf{1}, write : \bullet^1 \overleftarrow{Int}^4\mathbf{1}\}$, which means that Q also needs to read P_1's choice, as if it is P's choice.

P_2 does not necessarily need to read the choice if T_2 suits all the possible branches: for instance, $T_2 = \bullet^5\mathbf{1}$ would satisfy both read and write cases. Also, if the choice were given by Q instead of P_1, both T_1 and T_2 could read the choice simultaneously.

In summary, the merge operation allows us to verify and analyse the sharing of channels before any testing or simulation. It is a crucial part of h-calculus since it enables sharing to be encoded seamlessly within the type rules as we will see in Section 4.

2.3 Untyped Processes

So, we understand that types describe/model hardware component properties, but we still do not know how to define custom components from simpler ones. In this section we will explain how *untyped temporal processes* are built and how they evolve through time (their *semantics*).

Definition 3 (Untyped Process Syntax)

$x, y, x_1, x_2 ::=$ *Channel/Protocol variable*

$r ::=$ *Resource variable*

$\tau ::=$ *Temporal (real) value*

$k, \ell ::=$ *Choice variable*

$a ::=$ *Constant value*

$L ::=$ *Label*

$$
\begin{array}{rcll}
f & ::= & \mathit{Function} & \\
P,Q & ::= & \texttt{Main}(P) & \text{(Main process definition)} \\
 & \mid & x \leftarrow y & \text{(Channel forwarding)} \\
 & \mid & x \leftarrow P; Q & \text{(Forking/Cut)} \\
 & \mid & x \leftarrow r \leftarrow \{\Sigma;\Delta\}; Q & \text{(Instance Usage)} \\
 & \mid & \texttt{tick}\ \tau; P & \text{(Tick)} \\
 & \mid & \texttt{clock}; P & \text{(Clock)} \\
 & \mid & x \leftarrow \texttt{put}\ y; P \ \mid\ y \leftarrow \texttt{get}\ x; P & \text{(Sending/Receiving messages)} \\
 & \mid & x.k; P \ \mid\ \texttt{case}\ x\ \texttt{of}\ \{\ell \Rightarrow Q_\ell\}_{\ell \in L} & \text{(Internal/External Choice)} \\
 & \mid & P \parallel Q \ \mid\ (x_1, x_2) \leftarrow x; P & \text{(Parallelism/Channel separation)} \\
 & \mid & (x \rightarrow (x_1, x_2)).\big(P \parallel Q\big) & \text{(Parallelism with internal channels)} \\
 & \mid & L : P & \text{(Recursion - Label Loop)} \\
 & \mid & \texttt{end}\ x & \text{(End of Process)} \\
 & \mid & \texttt{Sig}(\tau, x \leftarrow a) & \text{(Signal Process)} \\
 & \mid & \texttt{Comb}(f, \tau, y \leftarrow (x_1, x_2, \cdots, x_n)) & \text{(Combinational Process)} \\
 & \mid & \texttt{Reg}(y \leftarrow x) & \text{(Register Process)}
\end{array}
$$

While the grammar is a syntactic description of untyped processes, their *semantics* describes their meaning. Their *operational semantics* is defined using the partial function $\longmapsto$, which relates the current state of the system to its next state, representing how processes evolve with time. We will discuss the complete, formal, definition of the semantics in Section 4.

2.4 Processes Types

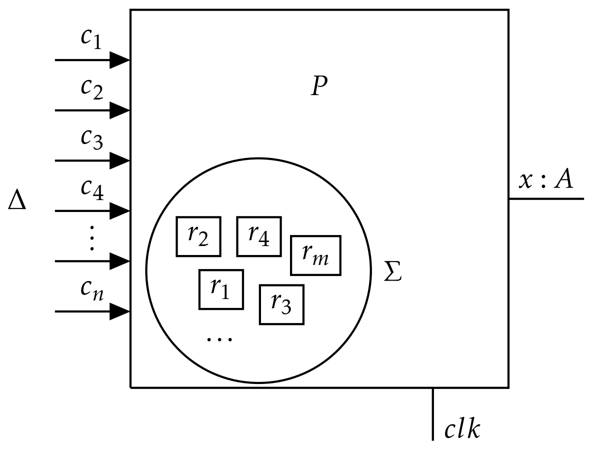
Channel types, as the name suggests, describes how channels behave. However, hardware components interact with many channels simultaneously, which is why their types are more complex. The mathematical construct that describes the behavior of processes, and therefore the *process type* is a *type sequent*.

After the *type checking* phase, every well-typed process P is attributed a *type sequent* that tells us the type of P, which includes the types of its channels and resources.

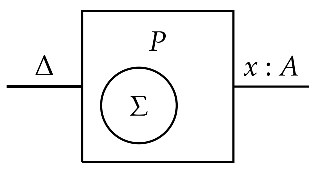
Definition 4 (Process Sequent) A sequent has the form

$$\Sigma;\Delta \mid\!\frac{k,t}{c,s}\, P :: (x : A),$$

where Σ is a *resource context* containing all resources P must have access to in order to function correctly, Δ is a channel context with the channels P interacts with, and $x : A$ is the channel provided by P (note that every process has only one provided channel). c, s, k, and t are numbers representing the clock period, the useful cycle duration, the current clock cycle count, and the current instant (from 0 to s), respectively. Two graphical representations for the sequent, one more complete (a) and another more simplified (b), are shown in Fig. 2.23.



(a) Complete



(b) Simplified

Figura 2.23: Graphical representations of a well-typed process

A channel context $\Delta = c_1 : S_1, \cdots, c_n : S_n$ is a set of channel typing judgements while the resource context $\Sigma = r_1 : R_1, \cdots, r_m : R_m$ is a set of resource typing judgements. Resource types are different from channel types as we are going to see.

When a process P with sequent $\Sigma;\Delta \mid\!\frac{k,t}{c,s}\, P :: (x : A)$ is instantiated, or used, by a parent process, the parent needs to provide all of the resources in Σ and channels in Δ, which means the parent process contains at least the entire Σ and Δ within its own channel and resource contexts respectively. The consequence of this is that the highest level process, the one that defines the entire system, will have inside its contexts all channels and resources of the entire system, along with their informative types, which is very useful for analysis and optimization.

2.5 Processes as resources

The definition of a process and its sequent provides us with insights on how a process operates internally, what is required for it to compute properly and what kind of information its channels carry. However, it does not tell us how processes are used, as resources, by other higher-level processes.

A *process definition* is a template from which process instances are created. Multiple different instances (sometimes even running in parallel) are linked to the same process definition. For higher-level processes, an instance of process is a *resource*, an object that can be used and shared by other processes.

Every well-typed process definition $\Sigma;\Delta \vdash^{k,t}_{s,c} P :: (x : A)$ has a resource context Σ, which is the set of all resources required for the process to compute correctly. Each one of the objects $r : R$ in Σ has, along with its name r, a resource type R which tells us how the higher-level process P interacts with r.

Definition 5 (Resource Types) Resource types are triples $(\Sigma_{\mathtt{ext}};\Delta_{\mathtt{ext}};A_{\mathtt{ext}})$, where $\Sigma_{\mathtt{ext}}$ and $\Delta_{\mathtt{ext}}$ are resource and channel context that must be provided for the resource to compute correctly, and $A_{\mathtt{ext}}$ is a temporal session type.

Not coincidentally, a resource type contains the same three objects represented in a typed process definition. Process definitions can be interpreted as resources used only once, and resources can be interpreted as process definitions used generally more than once.

Although process definitions and resource types carry the same kind of information, resource types generally carry more information. The difference is that $\Sigma_{\mathtt{ext}}$, $\Delta_{\mathtt{ext}}$ and $A_{\mathtt{ext}}$ are *temporally extended* versions of Σ, Δ and A respectively, containing information about many executions, instead of just one — which is formally defined by Def. 6. While $\Sigma;\Delta \vdash^{k,t}_{s,c} P :: (x : A)$ models one full computation from start to finish, $p : (\Sigma_{\mathtt{ext}};\Delta_{\mathtt{ext}};(x_{\mathtt{ext}} : A_{\mathtt{ext}}))$ models what happens before, during and after full computations, until p stops being used by the parent process. Another way of looking at it is that resource types model the `idle` time and computation time, while process definitions only model computation time.

Resource types and process definitions consisting of similar parts is convenient, as they can be interpreted interchangeably. For example, a process $\Sigma;\Delta \vdash^{k,t}_{s,c} P :: (x : A)$ can be seen as a resource typed $(\Sigma;\Delta;(x : A))$ that is used only once and a resource typed $(\Sigma_{\mathtt{ext}};\Delta_{\mathtt{ext}};(x_{\mathtt{ext}} : A_{\mathtt{ext}}))$ can be interpreted as a process $\Sigma_{\mathtt{ext}};\Delta_{\mathtt{ext}} \vdash^{k,t}_{s,c} P' :: (x_{\mathtt{ext}} : A_{\mathtt{ext}})$ with longer computation (as if `idle` time is considered computation time). This allows instance types to be extended more than once, which is a simple way to model the way instances are used by processes higher and higher in the hierarchy.

Example 2.12 (Resource Type). (Fig. 2.24) Consider that process P uses resource r three times per execution and process Q uses an instance of P two times per execution. Both P and Q contain r within their resource contexts, but with different types: in P, r has the resource type of being used 3 times, while in Q it has a more detailed type representing six uses, because it is extended even further. This way, the use of r can be traced to the highest-level process, which contains all of the system's resources, along information on how each one is used over time (very useful information for global hardware analysis and optimization).

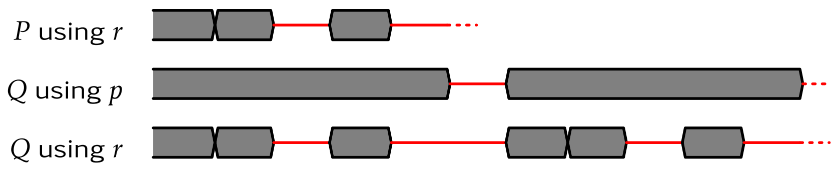


Figura 2.24: Example of resource type extension

Definition 6 (Resource Type Extension) A resource $p : (\Sigma_p;\Delta_p;(x_p : A_p))$ models a process $\Sigma;\Delta \vdash^{k,t}_{s,c} P :: (x : A)$ if and only if $\mathtt{ext}((\Sigma;\Delta;A)),(\Sigma_p;\Delta_p;A_p))$ holds, meaning the usage of p does not contradict the type of P. $\mathtt{ext}$ and its auxiliary definitions $\mathtt{ext}_\Sigma$, $\mathtt{ext}_\Sigma$, and $\mathtt{ext}_c$ are defined as

•$\mathtt{ext}((\Sigma_p;\Delta_p;A_p),(\Sigma_r;\Delta_r;A_r)) \iff \mathtt{ext}_\Sigma(\Sigma_p,\Sigma_r)$ and $\mathtt{ext}_\Delta(\Delta_p,\Delta_r)$ and $\mathtt{ext}_c(A_p,A_r)$

•$\mathtt{ext}_\Sigma(\Sigma_p,\Sigma_r) \iff \forall\rho.($if $(\rho:(\Sigma_1;\Delta_1;A_1)) \in \Sigma_p$ and $(\rho:(\Sigma_2;\Delta_2;A_2)) \in \Sigma_r$ then

$\mathtt{ext}_\Sigma(\Sigma_1,\Sigma_2)$ and $\mathtt{ext}_\Delta(\Delta_1,\Delta_2)$ and $\mathtt{ext}_c(A_1,A_2))$

•$\mathtt{ext}_\Delta(\Delta_p,\Delta_r) \iff \forall c.($if $(x:A_1) \in \Delta_p$ and $(x:A_2) \in \Delta_r$ then $\mathtt{ext}_c(A_1,A_2))$

•$\mathtt{ext}_c(p,r) \iff$ case (p,r) of

$(A\cdot 1, A\cdot B) \to \mathtt{ext}_c(A\cdot 1,B)$

$(A, \bullet^\tau B) \to \mathtt{ext}_c(A,B)$

$(A, \mu x.B) \to \mathtt{ext}_c(A, B[x/\mu x.B])$

$(A, \overleftrightarrow{\otimes}_c\{B_\ell\}_{\ell\in L}) \to \forall \ell \in L.(\mathtt{ext}_c(A,B_\ell))$

$(\overleftrightarrow{\otimes}_c\{A_\ell\}_{\ell\in L}, \overleftrightarrow{\otimes}_c\{B_\ell\}_{\ell\in L}) \to \forall \ell \in L.(\mathtt{ext}_c(A_\ell,B_\ell))$

$(A_1 \otimes A_2, B_1 \otimes B_2) \to \mathtt{ext}_c(A_1,B_1)$ and $\mathtt{ext}_c(A_2,B_2)$

$(A, 1) \to \mathtt{true}$

$\mathtt{otherwise} \to \mathtt{false}$

where $\overleftrightarrow{\otimes}_c$ matches both $\overrightarrow{\otimes}_c$ and $\overleftarrow{\otimes}_c$.

Example 2.13 (Examples of resource types). Let us say that the process Q is types as

$$-;x:\overrightarrow{A}^{s}1 \,\Big|\!\frac{k,0}{s,c}\, Q :: (y : \bullet^{s}\bullet^{s}\overrightarrow{B}^{s}1)$$

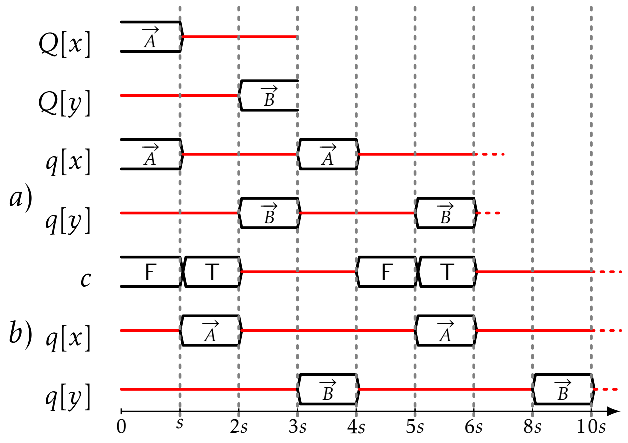


Figura 2.25: Example waveforms (a) Case IV (b) Case V

Case IV (Infinite usage (Fig. 2.25a)). *Possible correct types for extended x and y and an instance q could be (where $\mu x.T$ is the recursive, or loop, operator)*

$$x : \overrightarrow{A}^{s}\bullet^{s}\bullet^{s}\overrightarrow{A}^{s}\bullet^{s}\bullet^{s}\overrightarrow{A}^{s}\bullet^{s}\bullet^{s}\overrightarrow{A}^{s}\bullet^{s}\bullet^{s}\dots = \mu L.(\overrightarrow{A}^{s}\bullet^{s}\bullet^{s}L)$$

$$y : \bullet^s \bullet^s \overrightarrow{B}^s \bullet^s \bullet^s \overrightarrow{B}^s \bullet^s \bullet^s \overrightarrow{B}^s \bullet^s \bullet^s \overrightarrow{B}^s \cdots = \mu L.(\bullet^s \bullet^s \overrightarrow{B}^s L)$$

$$q : \Big(-; x : \mu L.(\overrightarrow{A}^s \bullet^s \bullet^s L); y : \mu L.(\bullet^s \bullet^s \overrightarrow{B}^s L) \Big)$$

Case V (Conditional usage (Fig. 2.25b)). *Another possibility including a choice (external or internal) c (with the branches written vertically for better visualization) could be*

$$x : \mu L. \oplus_c \begin{cases} \overrightarrow{A}^s \bullet^s \bullet^s L, & \textit{if } c \textit{ is } \mathsf{T} \\ \bullet^s L, & \textit{if } c \textit{ is } \mathsf{F} \end{cases} \qquad y : \mu L. \oplus_c \begin{cases} \bullet^s \bullet^s \overrightarrow{B}^s L, & \textit{if } c \textit{ is } \mathsf{T} \\ \bullet^s L, & \textit{if } c \textit{ is } \mathsf{F} \end{cases}$$

With resulting resource type

$$q : \Big(-; x : \mu L.((\overrightarrow{A}^s \bullet^s \bullet^s L) \oplus_c (\bullet^s L)); y : \mu L.((\bullet^s \bullet^s \overrightarrow{B}^s L) \oplus_c (\bullet^s L)) \Big).$$

Bear in mind that if the choice c were not the same for x and y or if one of the clauses were flipped (T case with answer for F and vice versa), the system would not be well typed.

2.6 Resource Sharing

If P has access to channel $c : S$ and spawns two children processes in parallel P_1 and P_2, we saw in Section 2.2 that both would have access to c, but with different types S_1 and S_2 such that $S_1 \times S_2 = S$. A very similar thing happens with resources: if P has access to resource $r : R$, both P_1 and P_2 will have access to it, but with different resource types R_1 and R_2 such that $R_1 \times R_2 = R$, where the operator $\times$ denotes, in this case, resource type merge. The formal definition of resource merging involves channel type, channel context, and resource context merging (Definition 7).

Definition 7 (Resource Merge)

$$(\Sigma_1; \Delta_1; A_1) \times (\Sigma_2; \Delta_2; A_2) = (\Sigma_1 \times \Sigma_2; \Delta_1 \times \Delta_2; A_1 \times A_2)$$

Definition 8 (Channel context merge)

$(c : S_1, \Delta_1) \times (c : S_2, \Delta_2) = c : S_1 \times S_2, (\Delta_1 \times \Delta_2)$

$(c : S_1, \Delta_1) \times \Delta_2 = c : S_1, (\Delta_1 \times \Delta_2)$

$\Delta_1 \times (c : S_2, \Delta_2) = c : S_2, (\Delta_1 \times \Delta_2)$

$\Delta_1 \times - = \Delta_1$

$- \times \Delta_2 = \Delta_2$

Definition 9 (Resource context merge)

$(r : R_1, \Sigma_1) \times (r : R_2, \Sigma_2) = r : R_1 \times R_2, (\Sigma_1 \times \Sigma_2)$

$(r : R_1, \Sigma_1) \times \Sigma_2 = r : R_1, (\Sigma_1 \times \Sigma_2)$

$\Sigma_1 \times (r : R_2, \Sigma_2) = r : R_2, (\Sigma_1 \times \Sigma_2)$

$\Sigma_1 \times - = \Sigma_1$

$- \times \Sigma_2 = \Sigma_2$

Merging of resources (and channels) is used to define correct resource sharing among processes. Fig. 2.26 depicts what sharing a resource looks like: resource r, contained in both Σ_1 and Σ_2, has its input channels Δ^r (in this case only 2 channels) split into Δ_1^r and Δ_2^r, such that $\Delta_1^r \times \Delta_2^r = \Delta^r$ (context channel merge) and its resulting channel $x^r : A^r$ split into $x_1 : A_1^r$ and $x_2 : A_2^r$, such that $A_1^r \times A_2^r = A^r$ (channel merge). The resource requirements Σ^r are similarly provided by both P_1 and P_2, split into Σ_1^r and Σ_2^r, such that $\Sigma_1^r \times \Sigma_2^r = \Sigma^r$ (resource context merge).

This definition of resource merge enables very expressive modeling of correct resource sharing among parallel processes. It allows processes to interact "incompletely"with resources, not providing all resource and channel requirements, as long as other processes complete the interaction using their channels and resources. Using resource merge, we can correctly share all kinds of resources, including processes with endless execution, registers, and combinational processes.

Example 2.14 (Shared SUM process). The `SUM` process receives a number input each cycle and adds it to an accumulator, whose value is also the output of the next cycle. Since the initial value of the accumulator is 0, a sequence of inputs $1, 2, 3, 4, 5$ would

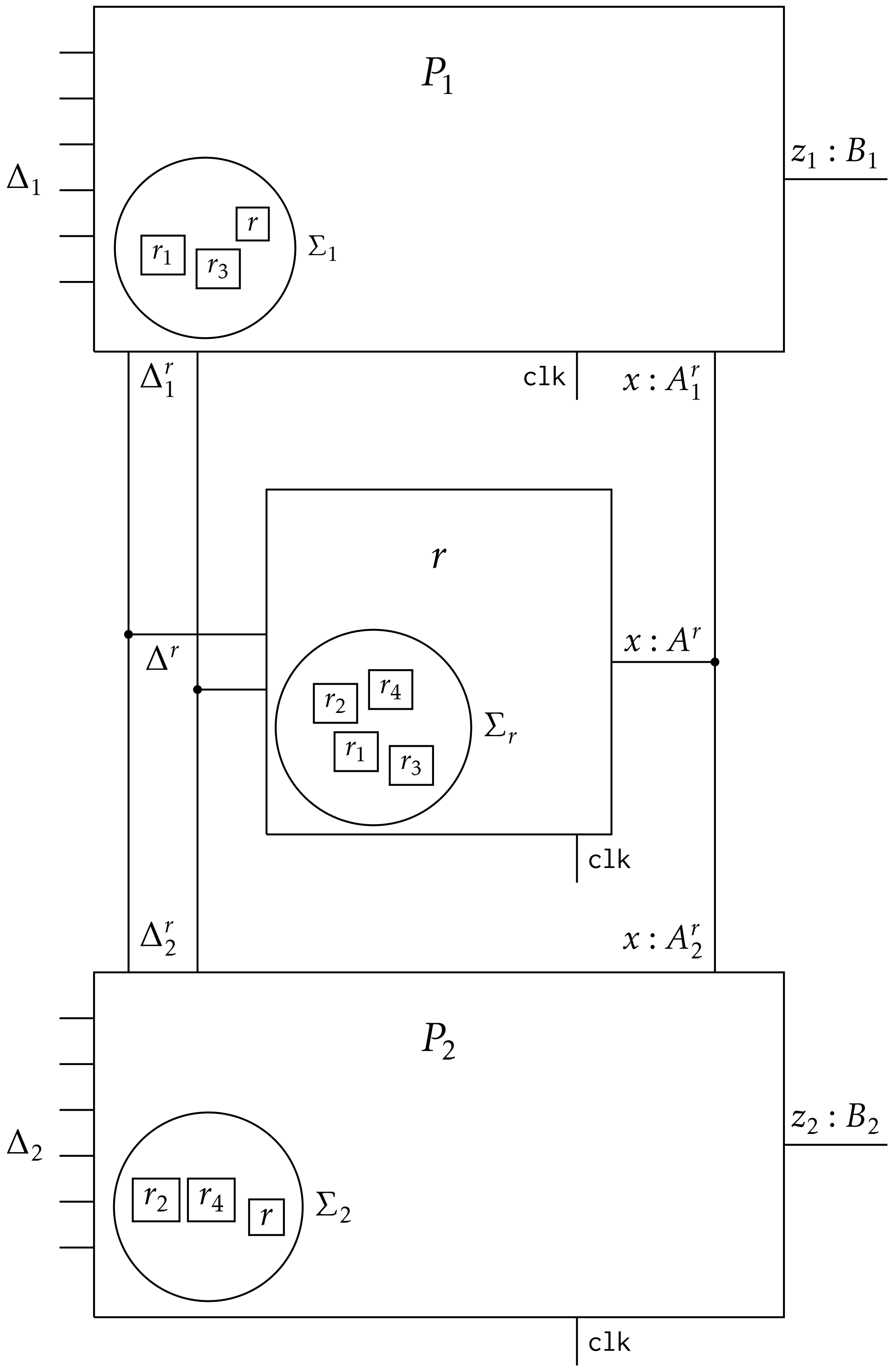


Figura 2.26: Graphical explanation of a resource sharing

produce a sequence $\bullet, 1, 3, 6, 10, 15$ of results. A possible type for `SUM` is

$$(\mathsf{Add} : \cdots); \left(x : \overrightarrow{Int^{s}}\right) \vdash^{k,0}_{s,c} \mathsf{SUM} :: \left(z : \bullet^{s}\overrightarrow{Int^{s}}\right).$$

In this example, the `SUM` process is shared (correctly) among processes P and Q (Fig. 2.27). For P, the resource `sum`of `SUM` has type

$$\mathsf{sum} : \left((\mathsf{Add} : \cdots); \left(x : \overrightarrow{Int^{s} \bullet^{s}}\right); \left(z : \bullet^{s}\overrightarrow{Int}\right)\right),$$

that is, P sends a number to `sum` every 2 cycles instead of every cycle, which is less than required by `SUM` for it to compute correctly. This is an example of incomplete, or partial, instantiation of process.

While only P does not completely satisfy `SUM`, the interactions of P combined with the interactions of Q do. For Q, `sum` has type

$$\mathsf{sum} : \left((\mathsf{Add} : \cdots); \left(x : \bullet^{s}\overrightarrow{Int^{s}}\right); \left(z : \bullet^{s}\overrightarrow{Int^{s}}\right)\right),$$

meaning Q sends a number to `sum` every cycle P does not, therefore completely satisfying the requirements of `SUM`.

Even though P and Q do not provide the complete requirements of `SUM` separately, both can interact with the resulting channel z completely, as if they had provided all the requirements (for example, if P provided all the input channels and resources while Q did not provide anything, Q would still be able to read z thoroughly).

Both P and Q do not "know"that `sum` is shared. If only one of them interacted with it, instead of both, the system would not type-check. The reason their combined interactions do type check is because the types of `sum` in P and Q merged is a correct instance of the process type of `SUM`:

$$\begin{aligned}
&\left((\mathsf{Add} : \cdots); \left(x : \overrightarrow{Int^{s} \bullet^{s}}\right); \left(z : \bullet^{s}\overrightarrow{Int^{s}}\right)\right)\\
\times &\left((\mathsf{Add} : \cdots); \left(x : \bullet^{s}\overrightarrow{Int^{s}}\right); \left(z : \bullet^{s}\overrightarrow{Int^{s}}\right)\right)\\
= &\left((\mathsf{Add} : \cdots); \left(x : \overrightarrow{Int^{s}}\right); \left(z : \bullet^{s}\overrightarrow{Int^{s}}\right)\right)
\end{aligned}$$

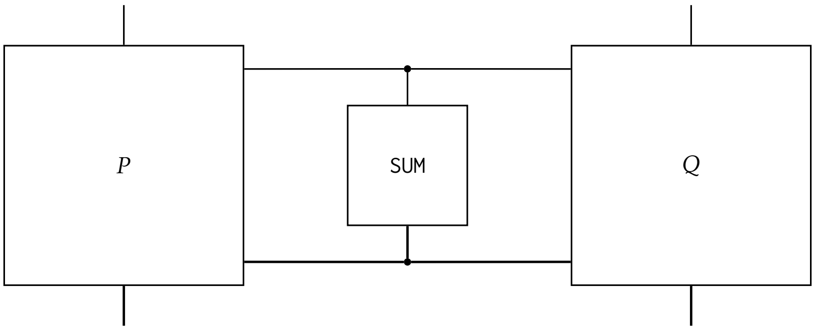


Figura 2.27: Sharing SUM process example

Example 2.15 (Sharing registers). An ALU process is capable of executing three operations, *increment* (plus 1), *negate* (multiply by -1) and *forward* (forwarding the input to the output). Each operation is performed by a different resource within the ALU, as shown in Fig. 2.28a. The INC and NEG resources have a similar layout composed of a combinational part (C_{inc} and C_{neg}) followed by a register (R_{inc} and R_{neg}), the FWD also has a register, but it does not have a combinational part.

The type of the ALU is (using the vertical notation of $\oplus$)

$$\Sigma_{\mathsf{alu}}; \left(c : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{\tau} \overrightarrow{Int}^{s-\tau} L, \\ neg : \bullet^{\tau} \overrightarrow{Int}^{s-\tau} L, \\ fwd : \bullet^{\tau} \overrightarrow{Int}^{s-\tau} L \end{cases} \right) \left|\frac{k,0}{s,c}\right. \mathsf{ALU} :: \left(d : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s} \overrightarrow{Int}^{s} L, \\ neg : \bullet^{s} \overrightarrow{Int}^{s} L, \\ fwd : \bullet^{s} \overrightarrow{Int}^{s} L \end{cases} \right)$$

where Σ_{alu} contains, among other things, the resources INC, NEG and FWD, each typed

$$\mathsf{C}_{\mathsf{inc}} : \cdots, \mathsf{R}_{\mathsf{inc}} : \cdots; \left(in_{\mathsf{inc}} : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{\tau} \overrightarrow{Int}^{s-\tau} L, \\ neg : \bullet^{s} L, \\ fwd : \bullet^{s} L \end{cases} \right) \left|\frac{t}{s,c}\right. \mathsf{INC} :: \left(out_{\mathsf{inc}} : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s} \overrightarrow{Int}^{s} L, \\ neg : \bullet^{s} L, \\ fwd : \bullet^{s} L \end{cases} \right),$$

$$\mathsf{C}_{\mathsf{neg}} : \cdots, \mathsf{R}_{\mathsf{neg}} : \cdots; \left(in_{\mathsf{neg}} : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s} L, \\ neg : \bullet^{\tau} \overrightarrow{Int}^{s-\tau} L, \\ fwd : \bullet^{s} L \end{cases} \right) \left|\frac{t}{s,c}\right. \mathsf{NEG} :: \left(out_{\mathsf{neg}} : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s} L, \\ neg : \bullet^{s} \overrightarrow{Int}^{s} L, \\ fwd : \bullet^{c} L \end{cases} \right), \text{ and}$$

$$\mathsf{R}_{\mathsf{fwd}} : \cdots; \left(in_{\mathsf{fwd}} : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s} L, \\ neg : \bullet^{s} L, \\ fwd : \bullet^{\tau} \overrightarrow{Int}^{s-\tau} L, \end{cases} \right) \left|\frac{t}{s,c}\right. \mathsf{FWD} :: \left(out_{\mathsf{fwd}} : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s} L, \\ neg : \bullet^{s} L, \\ fwd : \bullet^{s} \overrightarrow{Int}^{s} L \end{cases} \right).$$

All of these resources read the choice c, which is the input of ALU, and then either compute if the choice was the right one or stay idle (represented by $\bullet^s$).

More importantly, we know, from the types, that when one of the resources (INC, NEG or FWD) is being used, the other ones are idle, which means there may be potential for resource sharing. INC, NEG or FWD use a register resource ($\mathsf{Reg}_{\mathsf{inc}}$, $\mathsf{Reg}_{\mathsf{neg}}$ and $\mathsf{Reg}_{\mathsf{fwd}}$), provided by the ALU process. Below are the types of the three registers inside Σ_{alu}:

$$\mathsf{Reg}_{inc} : \left(-; \left(a : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{\tau+p}\, \overrightarrow{Int}^{\,s-(\tau+p)} L, \\ neg : \bullet^{s} L, \\ fwd : \bullet^{s} L \end{cases} \right); \left(b : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s}\, \overrightarrow{Int}^{\,s} L, \\ neg : \bullet^{s} L, \\ fwd : \bullet^{s} L \end{cases} \right) \right)$$

$$\mathsf{Reg}_{neg} : \left(-; \left(a : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s} L, \\ neg : \bullet^{\tau+p}\, \overrightarrow{Int}^{\,s-(\tau+p)} L, \\ fwd : \bullet^{s} L \end{cases} \right); \left(b : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s} L, \\ neg : \bullet^{s}\, \overrightarrow{Int}^{\,s} L, \\ fwd : \bullet^{s} L \end{cases} \right) \right)$$

$$\mathsf{Reg}_{fwd} : \left(-; \left(a : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s} L, \\ neg : \bullet^{s} L, \\ fwd : \bullet^{\tau+p}\, \overrightarrow{Int}^{\,s-(\tau+p)} L \end{cases} \right); \left(b : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s} L, \\ neg : \bullet^{s} L, \\ fwd : \bullet^{s}\, \overrightarrow{Int}^{\,s} L \end{cases} \right) \right).$$

These types show that when one register is used, the other 2 are not, which means that sharing can take place correctly. Now that we know Regs can be shared, we slightly transform our ALU process, defining only one register named $\mathsf{Reg}_{\mathsf{shared}}$, instead of three, with type equals to the types of $\mathsf{Reg}_{\mathsf{inc}}$, $\mathsf{Reg}_{\mathsf{neg}}$ and $\mathsf{Reg}_{\mathsf{fwd}}$ merged and using it to instantiate all INC, NEG and FWD resources. $\mathsf{Reg}_{\mathsf{shared}}$ is a busy register that works every cycle. Fig. 2.28b shows the optimized version of the circuit.

$$\mathsf{Reg}_{shared} : \left(-; \left(a : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{\tau+p}\, \overrightarrow{Int}^{\,s-(\tau+p)} L, \\ neg : \bullet^{\tau+p}\, \overrightarrow{Int}^{\,s-(\tau+p)} L, \\ fwd : \bullet^{\tau+p}\, \overrightarrow{Int}^{\,s-(\tau+p)} L \end{cases} \right); \left(b : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^{s}\, \overrightarrow{Int}^{\,s} L, \\ neg : \bullet^{s}\, \overrightarrow{Int}^{\,s} L, \\ fwd : \bullet^{s}\, \overrightarrow{Int}^{\,s} L \end{cases} \right) \right).$$

2.7 Type rules and Properties

We know that every well-typed process has a type definition $\Sigma;\Delta \vdash^{k,t}_{s,c} P :: (x : A)$, but we still do not know how to construct it. Every well-typed process P is built according to *type rules*.

We explain type rules in the next Chapter 3, and the specific rules of the h-calculus are explained in Chapter 4.

For now, it suffices to say type rules and semantic rules are harmoniously related to each other in a way that *type preservation* and *global progress* hold. These properties ensure the type system is valuable and safe to use as a metric for correctness.

Next Chapter 3 will summarize all of the background knowledge required to understand the h-calculus, including type systems.

(a) Unoptimized (without sharing) (b) Optimized (with sharing)

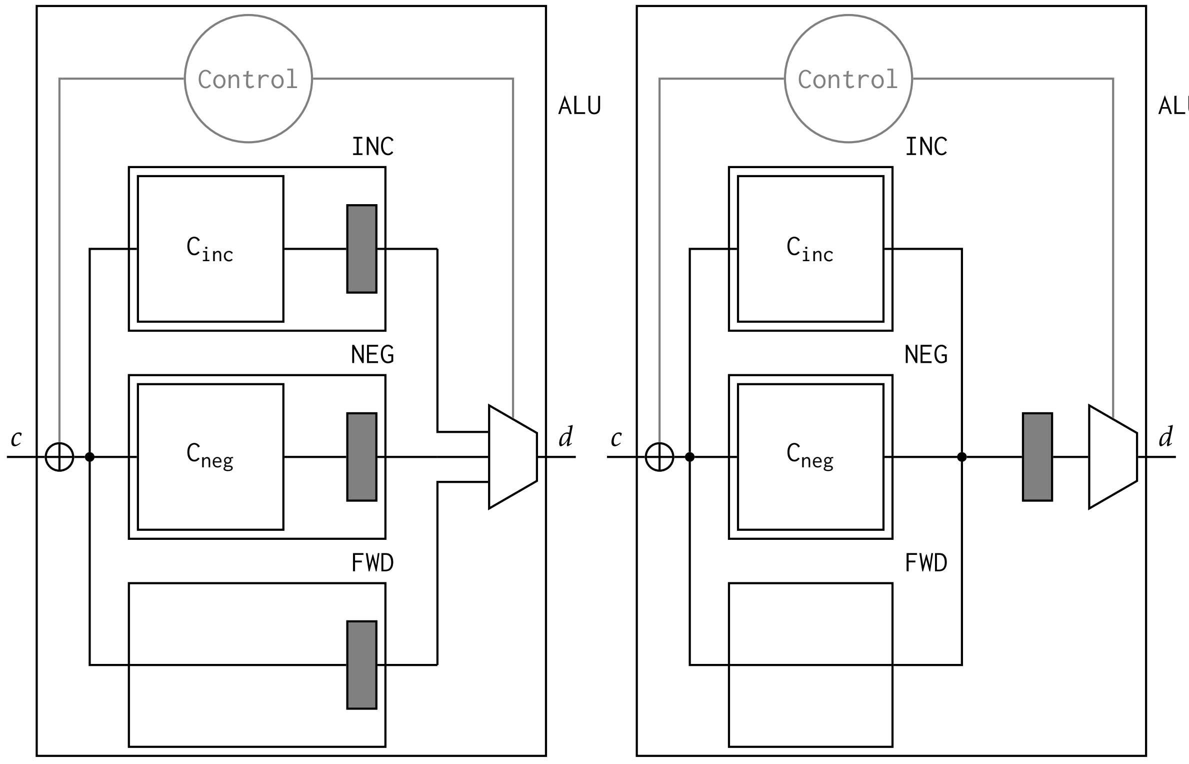


Figura 2.28: Graphical representation of the ALU process

Capítulo 3

Background

In this chapter, we will introduce type systems and explain the advantages of using them in the context of hardware systems. Furthermore, we will understand process calculi and other models of computation used for hardware modeling and compare them to the h-calculus.

To understand the usefulness of type systems, we will first use the Untyped Lambda Calculus (ULC), an untyped computational system, as a driving example.

3.1 Type Systems and the Lambda-Calculus

3.1.1 Untyped Lambda Calculus

The λ-calculus is a universal model of computation invented by Alonzo Church in the 1930s used to research the foundations of mathematics. It is attractive for formal reasoning because of its simple definitions and semantics, but it is also used in practical programming, as proven by multiple functional programming languages based on it.

The ULC grammar comprises variables, application, abstraction, and reduction rules τ-conversion, for avoiding naming collisions, and β-reduction, representing computation.

Definition 10 (ULC syntax)

$$\begin{aligned} M, N \ ::= \ & x && (\textit{Variables}) \\ | \ & c && (\textit{Constant}) \\ | \ & \lambda x.M && (\textit{Abstraction/Function}) \\ | \ & MN && (\textit{Application}) \end{aligned}$$

Definition 11 (ULC reduction) Considering $M[x]$ a term M containing a variable x in its body, and $M[a/x]$ the same term M with all instances of x replaced by a.

$$\begin{aligned} & (\lambda x.M[x]) \xrightarrow{\alpha} (\lambda y.M[y]) && (\alpha\textit{-conversion}) \\ & (\lambda x.M)a \xrightarrow{\beta} M[a/x] && (\beta\textit{-reduction}) \end{aligned}$$

with a either a variable or constant

Allowing integer constants $(\cdots, -1, 0, 1, 2, \cdots)$, boolean constants (`true`, `false`) and some operations such as + and ×, it is straightforward to write all sorts of useful functions using the ULC:

Example 3.1 (ULC examples).

a) $((\lambda x.\lambda y.x+y)5)4$ *Applying an adder to two arguments*

$\xrightarrow{\beta} (\lambda y.5+y)4$

$\xrightarrow{\beta} 5+4=9$

b) $(\lambda x.x+5)\texttt{true} \xrightarrow{\beta} \texttt{error?}$ *Applying wrong type to function*

c) $(\lambda x.xx)(\lambda x.xx)$ *Unsolvable term*

$\xrightarrow{\beta} (xx)[x/(\lambda x.xx)]$

$\xrightarrow{\text{subst}} (\lambda x.xx)(\lambda x.xx)$

$\xrightarrow{\beta} \cdots$

Although very expressive, the λ-calculus enables the construction of unwanted terms. For instance, there is no mechanism preventing the construction of terms such as $(\lambda x.x+5)\texttt{true}$ (1b), that would result in undefined behavior; and terms like $(\lambda x.xx)(\lambda x.xx)$ (1c), that do not terminate. In summary, no mechanism enforces a notion of correctness.

3.1.2 Simply Typed Lambda Calculus

The introduction of a type system solves this issue using a set of construction rules called **typing rules**. Valid terms are constructed through typing rules, while invalid terms cannot be derived from them. Of course, the definition of *valid term* depends on the specific type system.

A type scheme that restricts the construction of 1b and 1c terms is the *Simply Typed Lambda Calculus* (STLC). The STLC uses a simple type grammar (Def. 12) and only changes the ULC syntax by adding a type to variables.

Definition 12 (STLC syntax) The set of STLC types τ and terms M,N are given by

$$\begin{aligned} \tau &::= A \mid \tau \to \tau \\ M,N &::= x && (\textit{Variables}) \\ &\mid \lambda x:\tau.M && (\textit{Abstraction/Function}) \\ &\mid MN && (\textit{Application}) \end{aligned}$$

where A is a built in type — such as `Int` or `Bool` — and $\tau \to \tau$ is called a *function type*.

Type sequents are objects that tell us the type of a term, given a context type Γ.

Definition 13 (STLC Sequent) The STLC sequent types follow the structure

$$\Gamma \vdash M : \tau$$

where $\Gamma = x_1 : \tau_1, x_2 : \tau_2, \cdots, x_n : \tau_n$ is a context — a multiset containing judgements of form $x_i : \tau_i$, where x_i are variable terms and τ_i are types. Type sequents can be read as "M is typed τ given the context Γ".

Typing rules are the construction rules that allow the building of complex sequents from simpler ones. Typing rules follow the structure

$$\frac{\texttt{Premise}_1, \texttt{Premise}_2, \cdots, \texttt{Premise}_n}{\texttt{Conclusion}}\ \texttt{Rule Identifier}$$

where a rule may have more than one antecedent but only one result. If all antecedents are valid (i.e., they can be constructed through type rules), the type rule can be applied. Antecedents and results are, in most cases, *type sequents* (from Def. 13), but they may also be other judgements.

Definition 14 (STLC typing rules) The set of STLC's typing rules (with τ and σ being types):

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ \texttt{Var}$$

$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n : \texttt{Int}}\,\texttt{Cte(Int)}$$

$$\frac{\Gamma \vdash x : \texttt{Int} \quad \Gamma \vdash y : \texttt{Int}}{\Gamma \vdash x + y : \texttt{Int}}\,\texttt{Add}$$

$$\frac{\Gamma, x : \tau \vdash y : \sigma}{\Gamma \vdash (\lambda(x : \tau).y) : \tau \to \sigma}\,\texttt{Abs}$$

$$\frac{\Gamma \vdash f : \tau \to \sigma \quad \Gamma \vdash x : \tau}{\Gamma \vdash f x : \sigma}\,\texttt{App}$$

Rule `Var` states that if we know $x : \tau$ from the context, then we can conclude, trivially, that $x : \tau$. Rule `Cte(Int)` gives the type `Int` to any integer constant (as long as it really s an integer) and `Add` implements type safe addition on integers.

Rule `Abs` types abstractions by constructing the functional type $\tau \to \sigma$, that represents a function that takes an τ as input and outputs a σ, as long as the variable is typed τ and the body of the function is typed σ. The `App` rule enforces functions typed $\tau \to \sigma$ to only be applied if the input term is typed τ, typing the application σ as result.

The power of type rules rely on *derivation trees*. Derivation trees allow more complex terms to be derived from the simpler rules.

Example 3.2 (Derivation tree for STLC). The derivation tree

$$\dfrac{\dfrac{\dfrac{}{x : \texttt{Int} \vdash x : \texttt{Int}}\,\texttt{Var} \quad \dfrac{}{\vdash 5 : \texttt{Int}}\,\texttt{Cte(Int)}}{\dfrac{x : \texttt{Int} \vdash (5 + x) : \texttt{Int}}{\vdash (\lambda x : \texttt{Int}.5 + x) : \texttt{Int} \to \texttt{Int}}\,\texttt{Abs}}\,\texttt{Add} \qquad \dfrac{}{\vdash 4 : \texttt{Int}}\,\texttt{Cte(Int)}}{\vdash (\lambda x : \texttt{Int}.5 + x)4 : \texttt{Int}}\,\texttt{App}$$

proves that $(\lambda x.5 + x)4$ is of type `Int`, independently of the context.

3.1.3 Discussion on Type Systems

Type systems bring correctness in exchange for expressiveness. The STLC (without recursive types) is less expressive than the ULC. In the case of STLC, although its type system prevents unwanted terms (such as for example 1b and 1c) from being valid, it comes with the cost of considering some wanted terms invalid.

For example, an identity function $\lambda x : \tau.x$, that outputs its own input, is not particularly problematic as examples 1b and 1c. To be typed within the STLC, however, the type τ must be specified before an application. For example, if we set τ as `Int`, the identity function (λx : `Int`.x) will accept `Int`s but will not accept `Boolean`s although, semantically, there would be no issues with the function. For the identity function to work on any type, the type-system needs to allow *type polymorphism*.

Since type systems trade expressiveness for correctness, it is responsibility of the type system designer to choose how much and what kind of expressiveness they want to trade for correctness, and to define what does "correct"mean in the context of the target application.

For instance, if a polymorphic function like $\lambda x.x$ is important for the application, one could use a *System F* [12] type system instead of STLC, if types need to be more expressive to describe types such as a vector of length 5 (`Vector 5`) or a 3-by-3 matrix (`Matrix 3 3`) (or types even more complex), a *dependent type system* could be used [13]. Each type system comes with its own expressiveness/correctness ratios, differences and tradeoffs, which is why there are so many different type systems for the λ-calculus alone.

3.2 Hardware Models of Computation

When it comes to models of computation used in current HLS, a critique is that they are easy to construct but hard to verify and analyze. For most of the models, verification is challenging, and analysis is overlooked. In this sense, it becomes clear that a type system specially crafted for hardware design is a promising approach to solving many of the problems related to HLS.

A more detailed analysis of hardware models of computation will be provided in Chapter 6. In this chapter, we will only briefly characterize the general trends among models.

HLS is very commonly built around *Dataflow* models [5, 6, 7, 8, 9, 4, 10, 1, 2, 3, 4]. The Dataflow model [11] is a graphical-based concurrent model of computation. While expressive, Dataflow are generally difficult to verify and analyse, which is why different kinds of Dataflow introduce rules to make it more verifiable and analyzable, at expense of expressiveness, similar to the relationship between untyped and typed lambda calculi.

Although some Dataflow models — such as *Boolean Parametric Dataflow* (BPDF) [14], *Scenario-Aware Dataflow* (SADF) [15, 16], *Schedulable Parametric Dataflow* (SPDF) [17], and *Variable-Rate Dataflow* (VRDF) [18] — seriously attempt to solve the verification and analysis problem *Control Dataflow Graphs* [19], the most commonly used model in HLS, do not provide reliable mechanisms of both verification and analysis. As stated in the Chapter 1, we conjecture this is part of the reason why HLS has difficulty producing efficient results.

That being said, even the Dataflow models that solve verification and analysis have its caveats. From being a graph-based model, dataflow verification is computationally expensive for large systems and analysis is not as rich as we would want for hardware design space exploration. In face of these limitations, hardware-specific type systems still seem like a good idea.

Even if type systems seem to fit hardware design well, λ-calculi do not. The problem with λ-calculi is that they do not natively model concepts such as time, explicit parallelism, concurrency, communication, resource usage, channels, and others, crucial for efficient hardware modeling. Although it is possible to describe hardware with abstractions and applications, the long distance between functions and hardware would ultimately lead to inefficient modeling.

3.3 Session Types and Process Calculi

Among all candidate hardware representations, *Session Typed Process Calculi* stand out because they apply type system solutions to the context of concurrent processes.

The basic idea behind *Session Types* (ST) is to model concurrent processes using types, similar to how functional types are used, in Section 3.1, to model functions. They were first introduced in [20], to model the interaction between two communicating processes. Since then, as research evolved, different implementations were defined for different purposes, extending the original idea and expanding the use cases.

During the same time Session Types started to be researched, another research topic was the computational interpretation of *linear logic* [21, 22]. Notably, [23] described an *isomorphism* — a correspondence between linear logic and session types —, that connected both theories.

The particular ST implementation resulting from this particular isomorphism is exceptionally concise and expressive, using *Intuitionistic Linear Logic* (ILL) operators in a computational (e.g., $\otimes$, $\oplus$, $\&$, $\multimap$, and 1), instead of proof-theoretical, setting.

Now we will describe basic session types as described in [24, 25, 26] (Def. 15).

Definition 15 (Basic Session Types) The set of Session Types A, B, A_i is given by

$$A, B, A_i ::= A \otimes B \mid A \multimap B \mid \oplus\{A_i\} \mid \&\{A_i\} \mid 1.$$

In this computational interpretation of linear logic, linear logic operators represent a communication protocol. The multiplicative operators — the dual operations $\otimes$ and $\multimap$ — represent a higher-order message passing (sending or receiving) or parallel composition; the additive connectors — the duo $\oplus$ and $\&$ — represent internal and external choice respectively; and the *multiplicative truth* — 1 — is the end of the protocol.

This small syntax, plus recursion, is expressive enough to model complex communication protocols.

Example 3.3Basic Session Types Usage Examples. Session Type examples from [24, 25, 26]:

Case VI (Sequence of Bits). *An infinite sequence of bits could be modeled with internal choice and recursion*

$$bits = \oplus\{zero : bits, one : bits, end : 1\}$$

And if sequences are finite, an end choice can be added

$$bits = \oplus\{zero : bits, one : bits, end : 1\}$$

.

Case VII (List of Integers). *We can use a similar pattern to model a list of integers (or any other datatype):*

$$List = \oplus\{head : \mathtt{Int} \otimes List, end : 1\}$$

where the head of the list contains $\mathtt{Int} \otimes List$*, meaning a user will receive (*$\otimes$*) an integer and an updated* $List$*, with the next values.*

Case VIII (Sum Process). *Session types can model channels from concurrent processes very well. For instance, a* Sum *process adds every input it receives until a* $getVal$ *signal is received. The user sends integers using* $\multimap$ *and receives the updated* Sum *as a result.*

$$Sum = \oplus\{add : Int \multimap Sum, getVal : Int\}$$

Case IX (Stack). *We can also express protocols that require bidirectional flow of information. A stack, for example, would wait for an external choice (&) to either get or put a value from memory, but if get is chosen a value may or may not be available, which is modeled with an internal choice (⊕).*

$$Stack = \&\{put : \texttt{Int} \multimap Stack, get : \oplus\{empty : 1, val : \texttt{Int} \otimes Stack\}\}$$

If put is chosen, a user needs to send (⊸) an integer to receive the updated $Stack$ back, if get is chosen and the stack is not empty, an integer is sent (×) to the user together with the updated $Stack$.

3.4 Session Types for Hardware

Although classic STs seem to be closer to hardware designs than λ-calculi, they still lack accurate representations of concepts crucial for efficient hardware modeling, the most important of them being *time*.

Many concurrent models of computation model time [27, 28, 29, 30, 31, 32, 33, 34, 35, 24], but few of them combine it with ease of verification and analysis. Furthermore, most concurrent computational models, including STs, focus on *distributed systems*, meaning we need to tweak some details to enable accurate modeling of hardware architectures, which are local systems. For instance, system-wide synchronous time, resource sharing among components without trust issues, and determinism are notions that efficient hardware architectures rely on, but are hardly used in distributed environments.

The h-calculus combines temporal sequences, session types, process calculi, and synchronous time models to meet all of the requirements for efficient hardware modeling.

Capítulo 4

Semantics, Type Rules and Properties

In this section, we will show and explain formal definitions related to the h-calculus more formally. It will include definitions of its semantics, type system, and properties.

4.1 Operational Semantics

Before describing semantics it is important to define process equivalence. Processes constructed differently (according to 3) may be *equivalent* at the definition level. This equivalence definition describes that:

- time is fungible, so two `tick`s are the same as one longer `tick`; and that
- between time passage (`tick`s), the order of the actions do not matter since they occur, in fact, simultaneously.

Definition 16 (Process Definitional Equivalence) Process definitional equivalence is a relation $P \overset{def}{\approx} Q$ on process terms constructed from Def. 3. $P \overset{def}{\approx} Q$ holds in the following cases:

- $\texttt{tick}\ \tau_1;\texttt{tick}\ \tau_2;P' \overset{def}{\approx} \texttt{tick}\ (\tau_1+\tau_2);Q'$ *(Time is fungible)*

  if $P' \overset{def}{\approx} Q'$
- $a_1;a_2;\texttt{tick}\ \tau;P' \overset{def}{\approx} a_2;a_1;\texttt{tick}\ \tau;Q'$ *(Order between ticks does not matter)*

  if $P' \overset{def}{\approx} Q'$ *and* $(a_1, a_2 \neq \texttt{tick}\ t$ *for any* $t)$

Stating that $P \overset{def}{\approx} Q$ means that P and Q can be used interchangeably, within both type rules and operational semantic rules — P and Q will have the same type scheme and will reduce to the same operational semantic result (more about this later this chapter). It is also helpful to define an *equivalence set* that describes all processes that are equivalent to each other.

Definition 17 (Process Equivalence Set) A equivalence set $\{P_1, P_2, \cdots, P_n\}$ is a set of processes where for every i and $j \in [1, n]$, $P_i \overset{def}{\approx} P_j$. We refer as "the equivalence set of P"an equivalence set $\mathcal{P}(P)$ in which $P \in \mathcal{P}(P)$.

The operational semantics describes how the system, composed of multiple concurrent components, evolves through time and interacts with the outside environment. The h-calculus' operational semantics maps the current state to the next state, where a *configuration* represents the state.

Definition 18 (Configuration) A configuration is a structure that describes completely the state of the hardware system at a given moment. It is defined as

$$
\begin{array}{rll}
c & ::= \textit{Channel/Protocol variable} & \\
r & ::= \textit{Resource variable} & \\
\tau & ::= \textit{Real number} & \\
P & ::= \textit{Process term} & \\
\mathcal{C} & ::= \texttt{Main}(P) & \textit{(Main process definition)} \\
 & \mid\ \texttt{Closed}\{\,\mathcal{C}\,\} & \textit{(Closed configuration)} \\
 & \mid\ \mathcal{C}, \underline{\texttt{env}}(\tau,\tau,\tau,\tau) & \textit{(Environment values)} \\
 & \mid\ \mathcal{C}, \texttt{proc}\big(r \,\big|\, P \,\big|\, c\big) & \textit{(Resource executing process)} \\
 & \mid\ \mathcal{C}, \texttt{idle}(r) & \textit{(Idle resource)}
\end{array}
$$

Configurations model many hardware concepts such as time, clock, resource usage and sharing, execution of processes, communication through channels, and others. The semantic object:

- $\underline{\texttt{env}}(c, s, t, k)$ describes the temporal parameters of the configuration — c for clock period duration, s for the duration of the stable period, t for the time passed from the start of the current stable period until the present moment (from 0 to s), and k representing the number of complete clock periods already elapsed,
- $\texttt{idle}(r)$ represents a component not being used at the current moment

- $\texttt{proc}\big(r \mid P \mid c\big)$ describes a component r being executed as process P, providing channel c as output,
- $\texttt{Main}(P)$ represents the top-level process P, which is how all configurations start
- $\texttt{Closed}\{\,\mathcal{C}\,\}$ is a configuration that is closed in the sense that all internal channels are completely connected and the only channels available externally are, in fact, the inputs and output of the entire system.

Similar to untyped process syntax, not every configuration has semantic meaning or is correct (for instance, $\texttt{Closed}\{\,\texttt{Closed}\{\,\mathcal{C}\,\}\,\}$ makes no sense), but this is solved by the semantic and type rules that will not possibly construct such configurations.

A helper definition is configuration containment (Def. 19), indicating whether a semantic object is or is not inside a configuration.

Definition 19 (Configuration Containment) We say $o \in \mathcal{C}$, where o is a semantic object, if

- $\mathcal{C} = \mathcal{C}', o$ for any $\mathcal{C}'$, or
- $\mathcal{C} = \texttt{Closed}\{\,\mathcal{C}', o\,\}$ for any $\mathcal{C}'$.

Definition 20 (Operational Semantics) We define the operational semantics as rewriting rules from configuration to configuration. Every rule has the form

$$\texttt{Ant}\,[\vec{x}] \rightarrow \texttt{Conseq}\,[\vec{x}]$$

, where Ant is a pattern and Conseq is the resulting configuration in case Ant matches. Both Ant and Conseq are configurations defined using a set of parameters $\vec{x}$.

If the a current configuration $\mathcal{C}$ matches $\texttt{Ant}\,[\vec{x}]$, then the result of the match is a binding set of form $\vec{x} := \vec{a}$, where $\vec{a}$ are values inside $\mathcal{C}$. The result of the operational step is then $\texttt{Conseq}\,[\vec{a}/\vec{x}]$, representing the same Conseq but with the parameters replaced by current values from $\mathcal{C}$.

Since operational semantics based on multiset rewriting rules are well described by several publications [36], we will focus on the peculiar aspects of the h-calculus operational semantics. For instance, we allow $\texttt{Ant}\,[\vec{x}]$ and $\texttt{Conseq}\,[\vec{x}]$ to be, apart from all of the semantic objects ($\texttt{idle}(r)$, $\texttt{proc}\big(r \mid P \mid c\big)$, $\texttt{Main}(P)$ and $\texttt{Closed}\{\,\mathcal{C}\,\}$) that match themselves, the especial structure

$$[\mathcal{C}_i]_{\forall i \in [1,n]}$$

which is a pattern that matches multiple semantic objects at the same time. This pattern is crucial for the h-calculus because it allows for semantic rules to model hardware-like signals sent to more than one process simultaneously.

Furthermore it is important to note that pattern matching is valid up to process equivalence (Def. 16), meaning if $\texttt{proc}\big(r \mid P \mid c\big) \in \texttt{Ant}\,[\vec{x}]$, then any $\texttt{proc}\big(r \mid P' \mid c\big)$ with $P' \in \mathcal{P}(P)$ (Def. 17) would match correctly.

Because most of the h-calculus' semantic rules apply to Closed configurations, we define the helper notation $\xrightarrow{\texttt{Closed}}$ (Def. 21) for better visualization. All h-calculus semantic rules are defined in Def. 22

Definition 21 (Closed Step)

$$\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{C}' \overset{\text{def}}{=} \texttt{Closed}\{\,\mathcal{C}\,\} \rightarrow \texttt{Closed}\{\,\mathcal{C}'\,\}$$

Definition 22 (H-Calculus' Operational Semantic Rules)

(Main) $\texttt{Main}(P[\Sigma][\Delta][x]), \underline{\texttt{env}}(T_c, T_s, 0, 0)$
$\rightarrow \texttt{Closed}\Big\{\, \texttt{proc}\big(p \mid P \mid x\big), [\texttt{idle}(r)]_{\forall r \in \Sigma}, \underline{\texttt{env}}(T_c, T_s, 0, 0) \,\Big\}$ (fresh p)

(id) $\mathcal{C}, \texttt{proc}\big(r \mid x \leftarrow y \mid x\big) \xrightarrow{\texttt{Closed}} \mathcal{C}\,[y/x], \texttt{idle}(r)$

(cut) $\texttt{proc}\big(r \mid x \leftarrow P; Q \mid z\big) \xrightarrow{\texttt{Closed}} \texttt{proc}\big(r \mid Q\,[a/x] \mid z\big), \texttt{proc}\big(- \mid P\,[a/x] \mid a\big)$ (fresh a)

(1) $\texttt{proc}\big(r \mid \texttt{end}\; x \mid x\big) \xrightarrow{\texttt{Closed}} \texttt{idle}(r)$

(⊗★) $\texttt{proc}\big(p \mid P_1 \parallel P_2 \mid x\big), \Big[\texttt{proc}\big(q_i \mid (x_{i1}, x_{i2}) \leftarrow x; Q_i \mid z_i\big)\Big]_{\forall i \in [1,n]}$
$\xrightarrow{\texttt{Closed}} \texttt{proc}\big(p_1 \mid P_1\,[a_1/x] \mid a_1\big), \texttt{proc}\big(p_2 \mid P_2\,[a_2/x] \mid a_2\big)$
$, \Big[\texttt{proc}\big(q_i \mid Q_i\,[a_1/x_{i1}]\,[a_2/x_{i2}] \mid z_i\big)\Big]_{\forall i \in [1,n]}$ (fresh p_1,p_2,a_1,a_2)

(⊗) $\texttt{proc}\big(p \mid (x \rightarrow (x_1, x_2)).\big(P_1 \parallel P_2\big) \mid x\big), \Big[\texttt{proc}\big(q_i \mid (x_{i1}, x_{i2}) \leftarrow x; Q_i \mid z_i\big)\Big]_{\forall i \in [1,n]}$
$\xrightarrow{\texttt{Closed}} \texttt{proc}\big(p_1 \mid P_1\,[a_1/x_1]\,[a_2/x_2] \mid a_1\big), \texttt{proc}\big(p_2 \mid P_2\,[a_1/x_1]\,[a_2/x_2] \mid a_2\big)$
$, \Big[\texttt{proc}\big(q_i \mid Q_i\,[a_1/x_{i1}]\,[a_2/x_{i2}] \mid z_i\big)\Big]_{\forall i \in [1,n]}$ (fresh p_1,p_2,a_1,a_2)

(Loop) $\texttt{proc}\big(r \mid L : P\,[L] \mid x\big) \xrightarrow{\texttt{Closed}} \texttt{proc}\big(r \mid P\,[L : P\,[L]/L] \mid x\big)$

(→-1) $\texttt{proc}\big(r \mid x \leftarrow \texttt{put}\; y; P \mid x\big), \Big[\texttt{proc}\big(q_i \mid v_i \leftarrow \texttt{get}\; x; Q_i \mid z_i\big)\Big]_{\forall i \in [1,n]}$
$, \texttt{proc}\big(r_y \mid \texttt{Sig}(\tau, y \leftarrow w) \mid y\big)$ ($\tau \leq 0$)
$\xrightarrow{\texttt{Closed}} \texttt{proc}\big(r \mid P \mid x\big), \Big[\texttt{proc}\big(q_i \mid Q_i\,[y/v_i] \mid z_i\big)\Big]_{\forall i \in [1,n]}, \texttt{proc}\big(r_y \mid \texttt{Sig}(\tau, y \leftarrow w) \mid y\big)$

$$\textbf{(}\rightarrow\textbf{-2)}\ \text{proc}\Big(r \;\Big|\; x \leftarrow \text{put}\ y; P \;\Big|\; z\Big), \text{proc}\Big(r_x \;\Big|\; v_x \leftarrow \text{get}\ x; P_x \;\Big|\; x\Big)$$
$$,\Big[\text{proc}\Big(q_i \;\Big|\; v_i \leftarrow \text{get}\ x; Q_i \;\Big|\; z_i\Big)\Big]_{\forall i\in[1,n]} \text{proc}\Big(r_y \;\Big|\; \texttt{Sig}(\tau, y \leftarrow w) \;\Big|\; y\Big) \qquad (\tau \leq 0)$$
$$\xrightarrow{\texttt{Closed}} \text{proc}\Big(r \;\Big|\; P \;\Big|\; z\Big), \text{proc}\Big(r_x \;\Big|\; P_x \;\Big|\; x\Big)$$
$$\Big[\text{proc}\Big(q_i \;\Big|\; Q_i\,[y/v_i] \;\Big|\; z_i\Big)\Big]_{\forall i\in[1,n]}, \text{proc}\Big(r_y \;\Big|\; \texttt{Sig}(\tau, y \leftarrow w) \;\Big|\; y\Big)$$
$$\textbf{(}\oplus\textbf{-1)}\ \text{proc}\Big(r \;\Big|\; x.k; P \;\Big|\; x\Big), \Big[\text{proc}\Big(q_i \;\Big|\; \text{case}\ x\ \text{of}\ \{\ell \Rightarrow Q_{i\ell}\}_{\ell\in L} \;\Big|\; z_i\Big)\Big]_{\forall i\in[1,n]}$$
$$\xrightarrow{\texttt{Closed}} \text{proc}\Big(r \;\Big|\; P \;\Big|\; x\Big), \Big[\text{proc}\Big(q_i \;\Big|\; Q_{ik} \;\Big|\; z_i\Big)\Big]_{\forall i\in[1,n]}$$
$$\textbf{(}\oplus\textbf{-2)}\ \text{proc}\Big(r \;\Big|\; x.k; P \;\Big|\; z\Big), \text{proc}\Big(r_x \;\Big|\; \text{case}\ x\ \text{of}\ \{\ell \Rightarrow P_\ell\}_{\ell\in L} \;\Big|\; x\Big)$$
$$,\Big[\text{proc}\Big(q_i \;\Big|\; \text{case}\ x\ \text{of}\ \{\ell \Rightarrow Q_{i\ell}\}_{\ell inL} \;\Big|\; z_i\Big)\Big]_{\forall i\in[0,n]}$$
$$\xrightarrow{\texttt{Closed}} \text{proc}\Big(r \;\Big|\; P \;\Big|\; z\Big), \text{proc}\Big(r_x \;\Big|\; P_k \;\Big|\; x\Big), \Big[\text{proc}\Big(q_i \;\Big|\; Q_{ik} \;\Big|\; z_i\Big)\Big]_{\forall i\in[0,n]}$$
$$\textbf{(comb}\bigstar\textbf{)}\ \text{proc}\Big(r \;\Big|\; \texttt{Comb}(f, d, y \leftarrow x) \;\Big|\; y\Big), \text{proc}\Big(r_x \;\Big|\; \texttt{Sig}(\tau, x \leftarrow v) \;\Big|\; x\Big)$$
$$\xrightarrow{\texttt{Closed}} \text{proc}\Big(r \;\Big|\; \texttt{Sig}(\tau + d, y \leftarrow f(v)) \;\Big|\; y\Big)$$
$$\textbf{(comb)}\ \text{proc}\Big(r \;\Big|\; \texttt{Comb}(f, d, y \leftarrow (x_1, \cdots, x_n)) \;\Big|\; y\Big), \text{proc}\Big(r_x \;\Big|\; \texttt{Sig}(\tau, x \leftarrow v) \;\Big|\; x\Big)$$
$$,\text{proc}\Big(r_1 \;\Big|\; \texttt{Sig}(\tau_1, x_1 \leftarrow v_1) \;\Big|\; x_1\Big), \cdots, \text{proc}\Big(r_n \;\Big|\; \texttt{Sig}(\tau_n, x_n \leftarrow v_n) \;\Big|\; x_n\Big)$$
$$\xrightarrow{\texttt{Closed}} \text{proc}\Big(r \;\Big|\; \texttt{Sig}(\text{max}(\tau_1, \cdots, \tau_n) + d, y \leftarrow f(v_1, \cdots, v_n)) \;\Big|\; y\Big)$$
$$\textbf{(reg)}\ \text{proc}\Big(r \;\Big|\; \texttt{Reg}(y \leftarrow x) \;\Big|\; y\Big), \text{proc}\Big(r_x \;\Big|\; \texttt{Sig}(\tau, x \leftarrow v) \;\Big|\; x\Big), \underline{\text{env}}(c, s, k, t)$$
$$\xrightarrow{\texttt{Closed}} \underline{\text{env}}(c, s, k, t), \texttt{idle}(r)$$
$$,\text{proc}\Big(- \;\Big|\; \texttt{tick}\ s - \tau;\ \texttt{clock};\ a \leftarrow \texttt{Sig}(0, a' \leftarrow v);\ y \leftarrow a \;\Big|\; y\Big) \qquad (\text{fresh } a, a')$$
$$\textbf{(inst-1)}\ \texttt{idle}(p := P[\Sigma][\Delta][x]), \text{proc}\Big(r \;\Big|\; y \leftarrow p \leftarrow \{\overline{\Sigma}; \overline{\Delta}\}; Q[y] \;\Big|\; z\Big)$$
$$\xrightarrow{\texttt{Closed}} \text{proc}\Big(p \;\Big|\; P\Big[\overline{\Sigma}\Big/\Sigma\Big]\Big[\overline{\Delta}\Big/\Delta\Big][a/x] \;\Big|\; a\Big), \text{proc}\Big(r \;\Big|\; Q[a/y] \;\Big|\; z\Big) \qquad (\text{fresh } a)$$
$$\textbf{(inst-2)}\ \text{proc}\Big(p \;\Big|\; P[\Sigma][\Delta][x] \;\Big|\; x\Big), \text{proc}\Big(r \;\Big|\; y \leftarrow p \leftarrow \{\overline{\Sigma}; \overline{\Delta}\}; Q[y] \;\Big|\; z\Big)$$
$$\xrightarrow{\texttt{Closed}} \text{proc}\Big(p \;\Big|\; P\Big[\overline{\Sigma} \times \Sigma\Big/\Sigma\Big]\Big[\overline{\Delta} \times \Delta\Big/\Delta\Big][a/x] \;\Big|\; a\Big), \text{proc}\Big(r \;\Big|\; Q[a/y] \;\Big|\; z\Big) \qquad (\text{fresh } a)$$
$$\textbf{(tick)}\ \Big[\text{proc}\Big(r_i^p \;\Big|\; \texttt{tick}\ \tau; P_i \;\Big|\; x_i^p\Big)\Big]_{\forall i\in[1,l]}, \Big[\texttt{idle}\Big(r_i^r\Big)\Big]_{\forall i\in[1,m]}$$
$$,\Big[\text{proc}\Big(r_i^s \;\Big|\; \texttt{Sig}\Big(d_i^s, x_i^s \leftarrow v_i\Big) \;\Big|\; x_i^s\Big)\Big]_{\forall i\in[1,n]}, \underline{\text{env}}(c, s, k, t)$$
$$\xrightarrow{\texttt{Closed}} \Big[\text{proc}\Big(r_i^p \;\Big|\; P_i \;\Big|\; x_i^p\Big)\Big]_{\forall i\in[1,l]}, \Big[\texttt{idle}\Big(r_i^r\Big)\Big]_{\forall i\in[1,m]}$$
$$,\Big[\text{proc}\Big(r_i^s \;\Big|\; \texttt{Sig}\Big(d_i^s - \tau, x_i^s \leftarrow v_i\Big) \;\Big|\; x_i^s\Big)\Big]_{\forall i\in[1,n}, \underline{\text{env}}(c, s, k, t + \tau)$$
$$\textbf{(clock)}\ \Big[\text{proc}\Big(r_i^p \;\Big|\; \texttt{clock}; P_i \;\Big|\; x_i^p\Big)\Big]_{\forall i\in[1,l]}, \Big[\texttt{idle}\Big(r_i^r\Big)\Big]_{\forall i\in[1,m]}$$
$$,\Big[\text{proc}\Big(r_i^s \;\Big|\; \texttt{Sig}\Big(d_i^s, x_i^s \leftarrow v_i\Big) \;\Big|\; x_i^s\Big)\Big]_{\forall i\in[1,n]}, \underline{\text{env}}(c, s, k, s)$$
$$\xrightarrow{\texttt{Closed}} \Big[\text{proc}\Big(r_i^p \;\Big|\; P_i \;\Big|\; x_i^p\Big)\Big]_{\forall i\in[1,l]}, \Big[\texttt{idle}\Big(r_i^r\Big)\Big]_{\forall i\in[1,m]}$$
$$,\Big[\texttt{idle}\Big(r_i^s\Big)\Big]_{\forall i\in[1,n}, \underline{\text{env}}(c, s, k + 1, 0)$$

4.1.1 Main

$$\textbf{(Main)}\ \texttt{Main}(P[\Sigma][\Delta][x]), \underline{\text{env}}(T_c, T_s, 0, 0)$$
$$\rightarrow \texttt{Closed}\Big\{\ \text{proc}\Big(p \;\Big|\; P \;\Big|\; x\Big), [\texttt{idle}(r)]_{\forall r\in\Sigma}, \underline{\text{env}}(T_c, T_s, 0, 0)\ \Big\} \qquad (\text{fresh } p)$$

Every configuration starts with only one `Main` object, containing the top-level process definition, and an `env` object with the numerical values of the clock. The Main rule initializes every resource $r \in \Sigma$ as `idle` resources.

4.1.2 Id/Forwarding

$$\texttt{(id)}\ \mathcal{C}, \mathsf{proc}\Big(r \,\Big|\, x \leftarrow y \,\Big|\, x\Big) \xrightarrow{\texttt{Closed}} \mathcal{C}[y/x], \texttt{idle}(r)$$

When a process reaches a *forwarding* state, it means that it forwards any value from its input channel (y) to its output channel (x). The process then ends execution and becomes `idle` while the rest of the configuration replaces x by y.

4.1.3 Cut/Fork

$$\texttt{(cut)}\ \mathsf{proc}\Big(r \,\Big|\, x \leftarrow P; Q \,\Big|\, z\Big) \xrightarrow{\texttt{Closed}} \mathsf{proc}\Big(r \,\Big|\, Q[a/x] \,\Big|\, z\Big), \mathsf{proc}\Big(- \,\Big|\, P[a/x] \,\Big|\, a\Big) \quad \text{(fresh } a\text{)}$$

The logical **Cut** rule is interpreted as a parallel fork between two processes. It represents an *asymmetrical*, or dependent, parallelism because although P and Q run in parallel P feeds Q all of its results, not being a completely independent process. Since the process $\mathsf{proc}\Big(- \,\Big|\, P[a/x] \,\Big|\, a\Big)$ runs only once, it does not have a resource name assigned to it, thus the blank ($-$) field.

4.1.4 End of process

$$\texttt{(1)}\ \mathsf{proc}\Big(r \,\Big|\, \mathsf{end}\ x \,\Big|\, x\Big) \xrightarrow{\texttt{Closed}} \texttt{idle}(r)$$

When a process ends, the channel provided by it terminates and an `idle` object replaces the executing `proc` one. No resource is ever removed from the configuration in this semantics.

4.1.5 Parallelism

$$\begin{array}{l}
(\otimes)\ \mathsf{proc}\big(p \mid (x \to (x_1,x_2)).\big(P_1 \parallel P_2\big) \mid x\big), \big[\mathsf{proc}\big(q_i \mid (x_{i1},x_{i2}) \leftarrow x; Q_i \mid z_i\big)\big]_{\forall i \in [1,n]} \\
\quad \xrightarrow{\text{Closed}} \mathsf{proc}\big(p_1 \mid P_1\,[a_1/x_1]\,[a_2/x_2] \mid a_1\big), \mathsf{proc}\big(p_2 \mid P_2\,[a_1/x_1]\,[a_2/x_2] \mid a_2\big) \\
\quad , \big[\mathsf{proc}\big(q_i \mid Q_i\,[a_1/x_{i1}]\,[a_2/x_{i2}] \mid z_i\big)\big]_{\forall i \in [1,n]} \qquad \text{(fresh } p_1,p_2,a_1,a_2) \\
(\otimes\star)\ \mathsf{proc}\big(p \mid P_1 \parallel P_2 \mid x\big), \big[\mathsf{proc}\big(q_i \mid (x_{i1},x_{i2}) \leftarrow x; Q_i \mid z_i\big)\big]_{\forall i \in [1,n]} \\
\quad \xrightarrow{\text{Closed}} \mathsf{proc}\big(p_1 \mid P_1\,[a_1/x] \mid a_1\big), \mathsf{proc}\big(p_2 \mid P_2\,[a_2/x] \mid a_2\big) \\
\quad , \big[\mathsf{proc}\big(q_i \mid Q_i\,[a_1/x_{i1}]\,[a_2/x_{i2}] \mid z_i\big)\big]_{\forall i \in [1,n]} \qquad \text{(fresh } p_1,p_2,a_1,a_2)
\end{array}$$

Rule $(\otimes\star)$ matches a parallel composition and all of the processes that are listening to the parallel channel. After the matching, the rule spawns two independent processes, each with its output channel.

Rule $(\otimes)$ works the same way, with the difference that the parallel processes are not entirely independent. The rule allows parallel processes to "rename"their output channels, meaning they can react to each other's output.

4.1.6 Recursion (or loop)

$$(\textbf{Loop})\ \mathsf{proc}\big(r \mid L : P[L] \mid x\big) \xrightarrow{\text{Closed}} \mathsf{proc}\big(r \mid P[L : P[L]/L] \mid x\big)$$

Recursion is defined in this calculus using labels that mark a location. Once we reach a label, the rule replaces it with everything afterward. If $P = L : A_1;A_2;L$, then $A_1;A_2$ will be executed and after that, once L is reached, L will be replaced by $A_1;A_2;L$, and things will be repeated as $A_1;A_2;A_1;\cdots$.

4.1.7 Put/Get

$$\begin{array}{l}
(\to\text{-1})\ \mathsf{proc}\big(r \mid x \leftarrow \mathsf{put}\ y; P \mid x\big), \big[\mathsf{proc}\big(q_i \mid v_i \leftarrow \mathsf{get}\ x; Q_i \mid z_i\big)\big]_{\forall i \in [1,n]} \\
\quad , \mathsf{proc}\big(r_y \mid \mathtt{Sig}(\tau, y \leftarrow w) \mid y\big) \qquad (\tau \leq 0) \\
\quad \xrightarrow{\text{Closed}} \mathsf{proc}\big(r \mid P \mid x\big), \big[\mathsf{proc}\big(q_i \mid Q_i\,[y/v_i] \mid z_i\big)\big]_{\forall i \in [1,n]}, \mathsf{proc}\big(r_y \mid \mathtt{Sig}(\tau, y \leftarrow w) \mid y\big) \\
(\to\text{-2})\ \mathsf{proc}\big(r \mid x \leftarrow \mathsf{put}\ y; P \mid z\big), \mathsf{proc}\big(r_x \mid v_x \leftarrow \mathsf{get}\ x; P_x \mid x\big) \\
\quad , \big[\mathsf{proc}\big(q_i \mid v_i \leftarrow \mathsf{get}\ x; Q_i \mid z_i\big)\big]_{\forall i \in [1,n]} \mathsf{proc}\big(r_y \mid \mathtt{Sig}(\tau, y \leftarrow w) \mid y\big) \qquad (\tau \leq 0) \\
\quad \xrightarrow{\text{Closed}} \mathsf{proc}\big(r \mid P \mid z\big), \mathsf{proc}\big(r_x \mid P_x \mid x\big) \\
\quad \big[\mathsf{proc}\big(q_i \mid Q_i\,[y/v_i] \mid z_i\big)\big]_{\forall i \in [1,n]}, \mathsf{proc}\big(r_y \mid \mathtt{Sig}(\tau, y \leftarrow w) \mid y\big)
\end{array}$$

In both rules, the mechanism is the same. The value being put is transmitted, through bindings, to every process geting it. Additionally, the value being transmitted must come from a Signal process that holds it.

Rules **(→-1)** and **(→-2)** only differ in that in the first the value being put is the output channel, while in the second it is an input channel.

4.1.8 Internal/External Choice

$$\textbf{(}\oplus\textbf{-1)}\ \mathsf{proc}\big(r \mid x.k;P \mid x\big), \Big[\mathsf{proc}\big(q_i \mid \mathsf{case}\ x\ \mathsf{of}\ \{\ell \Rightarrow Q_{i\ell}\}_{\ell\in L} \mid z_i\big)\Big]_{\forall i\in[1,n]}$$
$$\xrightarrow{\mathsf{Closed}} \mathsf{proc}\big(r \mid P \mid x\big), \Big[\mathsf{proc}\big(q_i \mid Q_{ik} \mid z_i\big)\Big]_{\forall i\in[1,n]}$$
$$\textbf{(}\oplus\textbf{-2)}\ \mathsf{proc}\big(r \mid x.k;P \mid z\big), \mathsf{proc}\big(r_x \mid \mathsf{case}\ x\ \mathsf{of}\ \{\ell \Rightarrow P_\ell\}_{\ell\in L} \mid x\big)$$
$$, \Big[\mathsf{proc}\big(q_i \mid \mathsf{case}\ x\ \mathsf{of}\ \{\ell \Rightarrow Q_{i\ell}\}_{\ell inL} \mid z_i\big)\Big]_{\forall i\in[0,n]}$$
$$\xrightarrow{\mathsf{Closed}} \mathsf{proc}\big(r \mid P \mid z\big), \mathsf{proc}\big(r_x \mid P_k \mid x\big), \Big[\mathsf{proc}\big(q_i \mid Q_{ik} \mid z_i\big)\Big]_{\forall i\in[0,n]}$$

The choice operation works similarly to the get/put operations. The internal choice process sends a message containing the decision through the channel, and the listening processes react to it by selecting the corresponding continuation process.

4.1.9 Combinational process

$$\textbf{(comb)}\ \mathsf{proc}\big(r \mid \mathsf{Comb}(f,d,y \leftarrow (x_1,\cdots,x_n)) \mid y\big), \mathsf{proc}\big(r_x \mid \mathsf{Sig}(\tau,x \leftarrow v) \mid x\big)$$
$$, \mathsf{proc}\big(r_1 \mid \mathsf{Sig}(\tau_1,x_1 \leftarrow v_1) \mid x_1\big), \cdots, \mathsf{proc}\big(r_n \mid \mathsf{Sig}(\tau_n,x_n \leftarrow v_n) \mid x_n\big)$$
$$\xrightarrow{\mathsf{Closed}} \mathsf{proc}\big(r \mid \mathsf{Sig}(\mathsf{max}(\tau_1,\cdots,\tau_n)+d, y \leftarrow f(v_1,\cdots,v_n)) \mid y\big)$$
$$\textbf{(comb}\star\textbf{)}\ \mathsf{proc}\big(r \mid \mathsf{Comb}(f,d,y \leftarrow x) \mid y\big), \mathsf{proc}\big(r_x \mid \mathsf{Sig}(\tau,x \leftarrow v) \mid x\big)$$
$$\xrightarrow{\mathsf{Closed}} \mathsf{proc}\big(r \mid \mathsf{Sig}(\tau+d, y \leftarrow f(v)) \mid y\big)$$

A combinational process is executed from start to finish within one cycle period. It is fully defined by a pair (f,d) composed of function f and a *maximum process delay* d.

Rule **(comb)** applies function f to the current value v inside the input channel x and sets the current output y to the result of the application $f(v)$ with correct temporal delays. The rule also transforms the Comb into a Sig to prohibit multiple uses of the same combinational process in the same cycle.

The more general **(comb★)** rule does the same thing, but it allows functions with more than one input, that consumes multiple Signals in one cycle. The rule sets its delay as the maximum delay among all inputs max$(\tau_1,\cdots,\tau_n)$ plus the process delay d.

4.1.10 Register/Memory

$$
\begin{aligned}
&\textbf{(reg)}\ \mathtt{proc}\big(r \,\big|\, \mathtt{Reg}(y \leftarrow x) \,\big|\, y\big), \mathtt{proc}\big(r_x \,\big|\, \mathtt{Sig}(\tau, x \leftarrow v) \,\big|\, x\big), \underline{\mathtt{env}}(c,s,k,t) \\
&\quad \xrightarrow{\mathtt{Closed}} \underline{\mathtt{env}}(c,s,k,t), \mathtt{idle}(r) \\
&\quad , \mathtt{proc}\big(- \,\big|\, \mathtt{tick}\ s-\tau;\ \mathtt{clock};\ a \leftarrow \mathtt{Sig}(0, a' \leftarrow v);\ y \leftarrow a \,\big|\, y\big) \qquad \text{(fresh } a, a'\text{)}
\end{aligned}
$$

Registers are a special kind of process that carries values from one cycle to the next. Operationally it consumes a signal and becomes a proc that will become a *another* Signal, carrying the same value, in the next cycle.

Because of the special nature of computing between cycles, the semantics spawns an auxiliar process whose function is to provide the value signal at the start of the next cycle. This auxiliar process experiences the clock event and after that becomes a Sig process using the textitFork/Cut and Channel Forwarding ($a \leftarrow y$).

4.1.11 Resource instantiation

$$
\begin{aligned}
&\textbf{(inst-1)}\ \mathtt{idle}(p := P[\Sigma][\Delta][x]), \mathtt{proc}\big(r \,\big|\, y \leftarrow p \leftarrow \{\overline{\Sigma}; \overline{\Delta}\}; Q[y] \,\big|\, z\big) \\
&\quad \xrightarrow{\mathtt{Closed}} \mathtt{proc}\big(p \,\big|\, P\big[\overline{\Sigma}/\Sigma\big]\big[\overline{\Delta}/\Delta\big][a/x] \,\big|\, a\big), \mathtt{proc}\big(r \,\big|\, Q[a/y] \,\big|\, z\big) \qquad \text{(fresh } a\text{)} \\
&\textbf{(inst-2)}\ \mathtt{proc}\big(p \,\big|\, P[\Sigma][\Delta][x] \,\big|\, x\big), \mathtt{proc}\big(r \,\big|\, y \leftarrow p \leftarrow \{\overline{\Sigma}; \overline{\Delta}\}; Q[y] \,\big|\, z\big) \\
&\quad \xrightarrow{\mathtt{Closed}} \mathtt{proc}\big(p \,\big|\, P\big[\overline{\Sigma} \times \Sigma/\Sigma\big]\big[\overline{\Delta} \times \Delta/\Delta\big][a/x] \,\big|\, a\big), \mathtt{proc}\big(r \,\big|\, Q[a/y] \,\big|\, z\big) \qquad \text{(fresh } a\text{)}
\end{aligned}
$$

The semantics of resource instantiation are quite liberal, leaving the correctness verification for the type system. Rule **(inst-1)** models models how an *idle* resource starts executing and becomes a proc.

The **(inst-1)** rule, however, allows a process to provide only the channels and resources the instance needs **at the moment** (not forever). In other words, processes can instantiate resources incompletely, as long as the resources it needs at the current moment are available (the type system verifies this).

Incompletely instantiated resources need to be re-instantiated, which is why the rule **inst-2** exists. It "refuels"the contexts with new information.

Resource sharing is highly permissive in the h-calculus because of these two rules and resource types. As an extreme example, the h-calculus could allow a resource to receive, for n cycles, values coming from n different processes and would be able to verify that.

4.1.12 Tick/Clock

$$
\begin{array}{l}
\mathtt{(tick)}\ \left[\mathtt{proc}\left(r_i^p \mid \mathtt{tick}\ \tau; P_i \mid x_i^p\right)\right]_{\forall i\in[1,l]}, \left[\mathtt{idle}\left(r_i^r\right)\right]_{\forall i\in[1,m]} \\
\quad , \left[\mathtt{proc}\left(r_i^s \mid \mathtt{Sig}\left(d_i^s, x_i^s \leftarrow v_i\right) \mid x_i^s\right)\right]_{\forall i\in[1,n]}, \underline{\mathtt{env}}(c,s,k,t) \\
\quad \xrightarrow{\mathtt{Closed}} \left[\mathtt{proc}\left(r_i^p \mid P_i \mid x_i^p\right)\right]_{\forall i\in[1,l]}, \left[\mathtt{idle}\left(r_i^r\right)\right]_{\forall i\in[1,m]} \\
\quad , \left[\mathtt{proc}\left(r_i^s \mid \mathtt{Sig}\left(d_i^s - \tau, x_i^s \leftarrow v_i\right) \mid x_i^s\right)\right]_{\forall i\in[1,n}, \underline{\mathtt{env}}(c,s,k,t+\tau) \\
\mathtt{(clock)}\ \left[\mathtt{proc}\left(r_i^p \mid \mathtt{clock}; P_i \mid x_i^p\right)\right]_{\forall i\in[1,l]}, \left[\mathtt{idle}\left(r_i^r\right)\right]_{\forall i\in[1,m]} \\
\quad , \left[\mathtt{proc}\left(r_i^s \mid \mathtt{Sig}\left(d_i^s, x_i^s \leftarrow v_i\right) \mid x_i^s\right)\right]_{\forall i\in[1,n]}, \underline{\mathtt{env}}(c,s,k,s) \\
\quad \xrightarrow{\mathtt{Closed}} \left[\mathtt{proc}\left(r_i^p \mid P_i \mid x_i^p\right)\right]_{\forall i\in[1,l]}, \left[\mathtt{idle}\left(r_i^r\right)\right]_{\forall i\in[1,m]} \\
\quad , \left[\mathtt{idle}\left(r_i^s\right)\right]_{\forall i\in[1,n}, \underline{\mathtt{env}}(c,s,k+1,0)
\end{array}
$$

The **`(tick)`** rule estates what happens when time passes and the **`(clock)`** rule estates what happens when the clock cycle ends. Both of these rules only apply if the configuration matches **exactly all** of the objects in the pattern.

For the **`(tick)`** rule to match, all processes must be `tick`ing simultaneously, which emphasizes that the system experiences time synchronously — the exception being `Sig`s that carry intra-cycle values. After applied, the rule advances time inside `env` and advances the temporal sequences stored inside `Signal`s.

The system can never `tick` over to the next cycle; it can only `tick` until its end. For the **`(clock)`** rule to apply, all processes must synchronously acknowledge the end of the current cycle and start the next one by using the special `clock` operation. The clock operation destroys `Signal`s and updates the clock cycle count inside `env`.

4.2 Temporal Session Types

As seen in Chapter 2.7, Temporal Session Types are used, directly, to model communication channels among hardware components. These types are the foundation on which the h-calculus is built. Just like untyped processes, TSTs also have an equivalence relation (Def. 23). $S_1 \overset{def}{\approx} S_2$ means S_1 and S_2 can be used interchangeably without diffferent type rule implications.

Definition 23 (TST Equivalence) $S_1 \overset{def}{\approx} S_2$ holds in the following cases:

- $\bullet^{\tau_1} \bullet^{\tau_2} S \overset{def}{\approx} \bullet^{\tau_1+\tau_2} S'$ *(Time fungible)*

  if $S \overset{def}{\approx} S'$

- $S'_1 \otimes S'_2 \overset{def}{\approx} S'_2 \otimes S'_1$ *(Order of parallelism does not matter)*

Definition 24 (Type Equivalence Set) A equivalence set $\{S_1, S_2, \cdots, S_n\}$ is a set of processes where for every i and $j \in [1, n]$, $S_i \overset{def}{\approx} S_j$. We refer as "the equivalence set of S"a set $\mathcal{S}(S)$ in which $S \in \mathcal{S}(S)$.

4.3 Typing rules

4.3.1 Auxiliary Definitions

Before describing the h-calculus type rules, we show some important definitions that related to the definitions of process type (Def. 4) and resource types (Def. 5) shown in Chapter 2.7. These include typing judgements, contexts, context operations for both channels and resources.

Definition 25 (Channel Typing Judgement) A channel typing judgement is denoted $c : S$ where c is a type variable and S is a Temporal Session Type. $c : S$ means "c acts according to S".

Definition 26 (Channel Context) A channel context is an unordered set composed of multiple channel typing judgements. Is is defined as a list

$$\Delta := c : S, \Delta \mid -$$

where the operation $c : S, \Delta$ is called *appending* and $-$ denotes the empty context.

Definition 27 (Channel Containment) A channel c is contained within a context Δ, denoted $c \in \Delta$ if

$$\Delta = c : S, \Delta'$$

for any Δ' and S. We also say that c is not cointained within Δ, denoted $c \notin \Delta$, if $c \in \Delta$ does not hold.

Definition 28 (Channel Context Concatenation) The concatenation of channel contexts Δ_1 and Δ_2, denoted $\Delta_1\Delta_2$ is defined as

- $(c : S, \Delta_1')\Delta_2 = c : S, (\Delta_1'\Delta_2)$
  where $c \notin \Delta_2$
- $(-)\Delta_2 = \Delta_2$
- `undefined` otherwise

4.3.2 Typing Rules

H-Calculus typing rules ensure that if a hardware is well-typed communication errors, timing errors, and deadlocks do not happen (as we are going to see in Section 4.4). Furthermore it also encodes efficiency information within the types, enabling trivial performance analysis and comparisons. Definition 29 shows all of the type rules at once. Next we discuss them one-by-one.

Definition 29 (Type Rules) The set of all typing rules:

$$\frac{}{-; y : A \vdash^{k,t}_{s,c} \left(x \leftarrow y\right) :: \left(x : A\right)}\,\texttt{id}$$

$$\frac{\Sigma_1;\Delta_1 \vdash^{k,t}_{s,c} P :: \left(x : A\right) \quad \Sigma_2;\Delta_2, x : A \vdash^{k,t}_{s,c} Q :: \left(z : C\right)}{\Sigma_1 \times \Sigma_2;\Delta_1 \times \Delta_2 \vdash^{k,t}_{s,c} \left(x \leftarrow P; Q\right) :: \left(z : C\right)}\,\texttt{cut}$$

$$\frac{}{-;- \vdash^{k,t}_{s,c} \mathsf{end}\ x :: \left(x : 1\right)}\,\mathsf{1R} \qquad \frac{\Sigma;\Delta \vdash^{k,t}_{s,c} P :: \left(z : C\right)}{\Sigma;\Delta, x : 1 \vdash^{k,t}_{s,c} P :: \left(z : C\right)}\,\mathsf{1L}$$

$$\frac{\Sigma;\Delta, v : \alpha^{\tau}1, x : \bullet^{\tau}A \vdash^{k,t}_{s,c} P :: \left(z : C\right) \quad t + \tau = s}{\Sigma;\Delta, x : \overrightarrow{\alpha}^{\tau}A \vdash^{k,t}_{s,c} \left(v \leftarrow \mathsf{get}\ x; P\right) :: \left(z : C\right)}\,{\rightarrow}\ \mathsf{L}$$

$$\frac{\Sigma;\Delta \vdash^{k,t}_{s,c} P :: \left(z : \bullet^{\tau} A\right) \quad t + \tau = s}{\Sigma;\Delta, v : \alpha^{\tau} 1 \vdash^{k,t}_{s,c} \left(z \leftarrow \mathsf{put}\ v; P\right) :: \left(z : \overrightarrow{\alpha}^{\tau} A\right)} \longrightarrow \mathsf{R}$$

$$\frac{\Sigma;\Delta, x : \bullet^{\tau} A \vdash^{k,t}_{s,c} P :: \left(z : C\right) \quad t + \tau = s}{\Sigma;\Delta, v : \alpha^{\tau} 1, x : \overleftarrow{\alpha}^{\tau} A \vdash^{k,t}_{s,c} \left(x \leftarrow \mathsf{put}\ v; P\right) :: \left(z : C\right)} \longleftarrow \mathsf{L}$$

$$\frac{\Sigma;\Delta, v : \alpha^{\tau} 1 \vdash^{k,t}_{s,c} P :: \left(z : \bullet^{\tau} A\right) \quad t + \tau = s}{\Sigma;\Delta \vdash^{k,t}_{s,c} \left(v \leftarrow \mathsf{get}\ z; P\right) :: \left(z : \overleftarrow{\alpha}^{\tau} A\right)} \longleftarrow \mathsf{R}$$

$$\frac{\Sigma;\Delta, x : A\left[(\mu y.A)/y\right] \vdash^{k,t}_{s,c} P :: \left(z : C\right)}{\Sigma;\Delta, x : \mu y.A \vdash^{k,t}_{s,c} P :: \left(z : C\right)} \mu\ \mathsf{L}$$

$$\frac{\Sigma;\Delta \vdash^{k,t}_{s,c} P\left[(L : P)/L\right] :: \left(x : A\left[(\mu y.A)/y\right]\right)}{\Sigma;\Delta \vdash^{k,t}_{s,c} \left(L : P\right) :: \left(x : \mu y.A\right)} \mu\ \mathsf{R}$$

$$\frac{\begin{array}{c}\Sigma_1;\Gamma,\Delta_1, x_Q : B_I \vdash^{k,t}_{s,c} P :: \left(x_P : A_E \times A_I\right) \\ \Sigma_2;\Gamma,\Delta_2, x_P : A_I \vdash^{k,t}_{s,c} Q :: \left(x_Q : B_E \times B_I\right)\end{array}}{\Sigma_1 \times \Sigma_2;;\Delta_1 \times \Delta_2 \vdash^{k,t}_{s,c} \left(\left(x \rightarrow \left(x_P, x_Q\right)\right).\left(P \parallel Q\right)\right) :: \left(x : A_E \otimes B_E\right)} \otimes\ \mathsf{R}$$

$$\frac{\Sigma_1;\Gamma,\Delta_1 \vdash^{k,t}_{s,c} P :: \left(x : A\right) \quad \Sigma_2;\Gamma,\Delta_2 \vdash^{k,t}_{s,c} Q :: \left(x : B\right)}{\Sigma_1 \times \Sigma_2;;\Delta_1 \times \Delta_2 \vdash^{k,t}_{s,c} \left(P \parallel Q\right) :: \left(x : A \otimes B\right)} \otimes\ \mathsf{R}\ \star$$

$$\frac{\Sigma;\Delta, x_1 : A, x_2 : B \vdash^{k,t}_{s,c} Q :: \left(z : C\right)}{\Sigma;\Delta, x : A \otimes B \vdash^{k,t}_{s,c} \left((x_1, x_2) \leftarrow x; Q\right) :: \left(z : C\right)} \otimes\ \mathsf{L}$$

$$\frac{(k \in L) \quad \Sigma;\Delta \vdash^{k,t}_{s,c} P :: \left(z : A_k\right)}{\Sigma;\Delta \vdash^{k,t}_{s,c} \left(z.k; P\right) :: \left(z : \overrightarrow{\oplus}_z\{\ell : A_\ell\}_{\ell \in L}\right)} \overrightarrow{\oplus}\ \mathsf{R}$$

$$\frac{(\forall \ell \in L)\quad \Sigma_\ell;\Delta_\ell, x : A_\ell \vdash^{k,t}_{s,c} Q_\ell :: \big(z : C\big)}{\oplus_x\{\ell : \Sigma_\ell\}_{\ell\in L};\oplus_x\{\ell : \Delta_\ell\}_{\ell\in L}, x : \overrightarrow{\oplus}_x\{\ell : A_\ell\}_{\ell\in L} \vdash^{k,t}_{s,c} \big(\texttt{case}\ x\ \texttt{of}\ \{\ell \Rightarrow Q_\ell\}_{\ell\in L}\big) :: \big(z : C\big)}\ \overrightarrow{\oplus}\ \mathsf{L}$$

$$\frac{(\forall \ell \in L)\quad \Sigma_\ell;\Delta_\ell \vdash^{k,t}_{s,c} Q_\ell :: \big(z : A_\ell\big)}{\oplus_z\{\ell : \Sigma_\ell\}_{\ell\in L};\oplus_z\{\ell : \Delta_\ell\}_{\ell\in L} \vdash^{k,t}_{s,c} \big(\texttt{case}\ z\ \texttt{of}\ \{\ell \Rightarrow Q_\ell\}_{\ell\in L}\big) :: \big(z : \overleftarrow{\oplus}_z\{\ell : A_\ell\}_{\ell\in L}\big)}\ \overleftarrow{\oplus}\ \mathsf{R}$$

$$\frac{(k \in L)\quad \Sigma;\Delta, x : A_k \vdash^{k,t}_{s,c} P :: \big(z : C\big)}{\Sigma;\Delta, x : \overleftarrow{\oplus}_x\{\ell : A_\ell\}_{\ell\in L} \vdash^{k,t}_{s,c} \big(x.k;P\big) :: \big(z : C\big)}\ \overleftarrow{\oplus}\ \mathsf{L}$$

$$\frac{\Sigma;\Delta \vdash^{k,t+\tau}_{s,c} P :: \big(z : C\big)\quad t+\tau \geq s}{\bullet^\tau\Sigma;\bullet^\tau\Delta \vdash^{k,t}_{s,c} \big(\texttt{tick}\ \tau;P\big) :: \big(z : \bullet^\tau C\big)}\ \texttt{tick}$$

$$\frac{\Sigma;\Delta \vdash^{k+1,0}_{s,c} P :: \big(z : C\big)}{\Sigma;\Delta \vdash^{k,s}_{s,c} \big(\texttt{clock};P\big) :: \big(z : C\big)}\ \texttt{clock}$$

$$\frac{\Sigma^Q, r : (\Sigma_2^r;\Delta_2^r;A_2^r);\Delta^Q, x : A_1^r \vdash^{k,t}_{s,c} Q :: \big(z : C\big)}{\Sigma_1^r \times \Sigma^Q, r : (\Sigma_1^r;\Delta_1^r;A_1^r)\times(\Sigma_2^r;\Delta_2^r;A_2^r);\Delta_1^r \times \Delta^Q \vdash^{k,t}_{s,c} \big(x \leftarrow r \leftarrow \{\Sigma_1^r;\Delta_1^r\};Q\big) :: \big(z : C\big)}\ \mathsf{Use}$$

$$\frac{(\forall(((\sigma \coloneqq P_\sigma) : R_\sigma) \in \Sigma).\texttt{ext}(P_\sigma, R_\sigma))\quad \Sigma;\Delta \vdash^{0,0}_{s,c} P :: \big(z : C\big)}{\Sigma;\Delta \vdash^{0,0}_{s,c} \texttt{Main}(P) :: \big(z : C\big)}\ \texttt{Main}$$

$$\frac{\vdash^{F} e : T\quad (s > t+\tau)\quad (\tau \geq 0)}{-;- \vdash^{k,t}_{s,c} \big(\texttt{Sig}(\tau, y \leftarrow e)\big) :: \big(y : \bullet^\tau T^{s-(t+\tau)} 1\big)}\ \texttt{Signal-1}$$

$$\frac{\vdash^{F} e : T\quad (s > t+\tau)\quad (\tau < 0)}{-;- \vdash^{k,t}_{s,c} \big(\texttt{Sig}(\tau, y \leftarrow e)\big) :: \big(y : T^{s-t} 1\big)}\ \texttt{Signal-2}$$

$$\dfrac{\stackrel{F}{\vdash} f : T_1 \to \cdots \to T_n \to T_{out} \quad (s > t + \max(d_1,\cdots,d_n) + p) \quad (p > 0)}{\begin{array}{c} -; x_1 : \bullet^{d_1} T_1^{s-(t+d_1)} 1, \cdots, x_n : \bullet^{d_n} T_n^{s-(t+d_n)} 1 \mathrel{\vdash^{k,t}_{s,c}} \mathsf{Comb}(f, p, y \leftarrow (x_1,\cdots,x_n)) \\ :: \left(y : \bullet^{\max(d_1,\cdots,d_n)} \bullet^{p} T_{out}^{s-(t+\max(d_1,\cdots,d_n)+p)} 1\right) \end{array}}\ \mathsf{Comb}$$

$$\dfrac{(s > t + \tau)}{-; x : \bullet^{\tau} T^{s-(t+\tau)} 1 \mathrel{\vdash^{k,t}_{s,c}} \left(\mathsf{Reg}(y \leftarrow x)\right) :: \left(y : \bullet^{s-t} T^{s} 1\right)}\ \mathsf{Reg}$$

4.3.3 General Insights

The type system is based on *Intuitionistic Session Types* (IST), which are isomorphic to *Intuitionistic Linear Logic* (ILL) [23]. Some TST rules are similar to IST, but there are some modifications, additions and removals that make the calculus suitable for hardware modelling.

A recurring distinction in TST type rules compared to IST ones is that, instead of using *set partitioning* for context splitting, where intersections (and therefore sharing) are not permitted, TST uses definitions of resource and channel merge (×), allowing general sharing of resources and channels whenever a new process is spawned. This change appears in any type rule that manages two or more parallel processes, including the foundational cut rule. Apart from this, other distinctions will be discussed in detail individually.

Most type operations have a left rule, which tells us how an operation is used by processes, and a right rule which tells us how a process performs an operation. That said, some special operations such as id, cut and tick are not divided into left and right rules for reasons which will be explained individually later.

When explaining the rules, it will sometimes be useful to explain their logical interpretation, in addition to their hardware interpretation, for a broader understanding of why some rules are defined the way they are. Every type rule will also have, apart from its *proof-tree* definition, a graphical interpretation using the simplified depiction of process shown in Fig. 2.23b.

4.3.4 Identity or Channel Forwarding

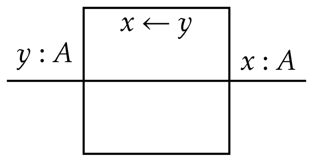


Figura 4.1: Graphical representation of the identity rule

$$\frac{}{-; y : A \mathrel{\Big|\!\!\frac{k,t}{s,c}} \big(x \leftarrow y\big) :: \big(x : A\big)}\ \texttt{id}$$

The channel forwarding or identity rule indicates that an input channel can be forwarded as an output. In terms of linear logic, this is one of the most essential rules, that enables an assumption to be used as a conclusion. This rule and the cut rule form the bridge that connects left (L) and right (R) rules.

4.3.5 Forking process

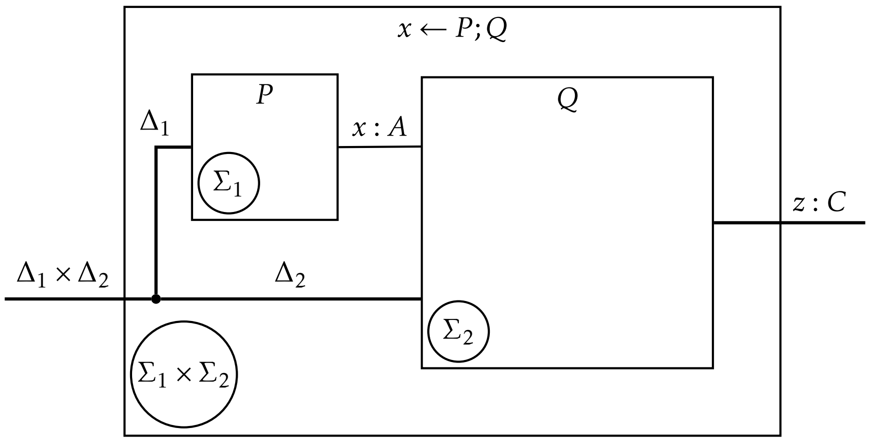


Figura 4.2: Graphical representation of the cut rule

$$\frac{\Sigma_1; \Delta_1 \mathrel{\Big|\!\!\frac{k,t}{s,c}} P :: \big(x : A\big) \quad \Sigma_2; \Delta_2, x : A \mathrel{\Big|\!\!\frac{k,t}{s,c}} Q :: \big(z : C\big)}{\Sigma_1 \times \Sigma_2; \Delta_1 \times \Delta_2 \mathrel{\Big|\!\!\frac{k,t}{s,c}} \big(x \leftarrow P; Q\big) :: \big(z : C\big)}\ \texttt{cut}$$

The cut rule defines how processes can fork subprocesses and use their result as input. Part of the main process contexts, Σ_1 and Δ_1, are assigned to the subprocess P according to the merge definition ($\times$), meaning resources can be shared between P and Q. The cut rule and the identity rule bridge (L) and right (R) rules logically by defining how hypothesis can be used to reach more complex conclusions.

4.3.6 End of computation - 1

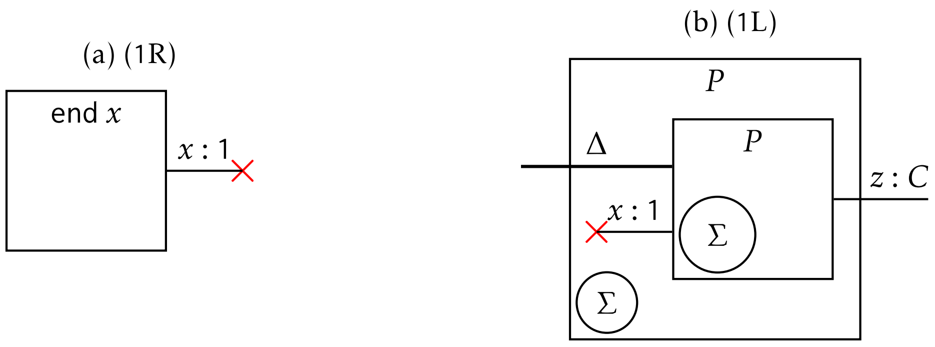


Figura 4.3: Graphical representation of 1 rules

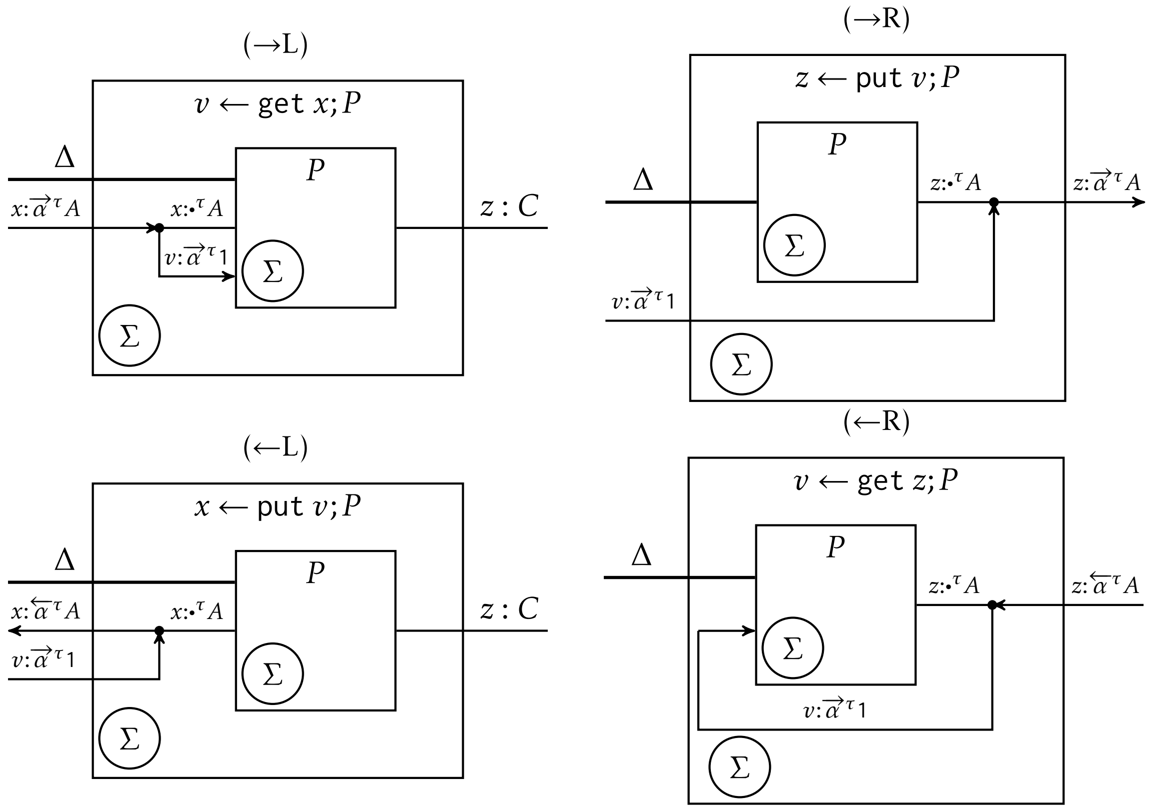
$$\frac{}{-;-\,\vdash^{k,t}_{s,c} \mathsf{end}\ x :: \left(x : 1\right)}\ 1R \qquad \frac{\Sigma;\Delta \vdash^{k,t}_{s,c} P :: \left(z : C\right)}{\Sigma;\Delta, x : 1 \vdash^{k,t}_{s,c} P :: \left(z : C\right)}\ 1L$$

The *end of computation* type 1 means the channel will not carry useful information anymore. Rule 1R constructs a channel typed 1 using the process term end x while rule 1L removes the useless channel from inside of its channel context.

The left rule does not have an explicit process action because all processes interacting with x have protocol knowledge, meaning they do not need additional information to know when x closes.

For the end x process, this rule also means the end of computation. In operational semantics, this means the instance will go `idle` after that.

4.3.7 Getting/Putting values from channels



(→L)
$v \leftarrow \mathsf{get}\ x; P$
P
Δ
$x{:}\overrightarrow{\alpha}^{\tau}A$
$x{:}\bullet^{\tau}A$
$v{:}\overrightarrow{\alpha}^{\tau}1$
$z : C$
Σ
(→R)
$z \leftarrow \mathsf{put}\ v; P$
P
Δ
$z{:}\bullet^{\tau}A$
$z{:}\overrightarrow{\alpha}^{\tau}A$
$v{:}\overrightarrow{\alpha}^{\tau}1$
Σ
(←L)
$x \leftarrow \mathsf{put}\ v; P$
P
Δ
$x{:}\overleftarrow{\alpha}^{\tau}A$
$x{:}\bullet^{\tau}A$
$v{:}\overrightarrow{\alpha}^{\tau}1$
$z : C$
Σ
(←R)
$v \leftarrow \mathsf{get}\ z; P$
P
Δ
$z{:}\bullet^{\tau}A$
$z{:}\overleftarrow{\alpha}^{\tau}A$
$v{:}\overrightarrow{\alpha}^{\tau}1$
Σ

Figura 4.4: Graphical representation of messaging rules

$$\frac{\Sigma;\Delta, v : \alpha^{\tau}1, x : \bullet^{\tau}A \vdash^{k,t}_{s,c} P :: \left(z : C\right) \quad t + \tau = s}{\Sigma;\Delta, x : \overrightarrow{\alpha}^{\tau}A \vdash^{k,t}_{s,c} \left(v \leftarrow \mathsf{get}\ x; P\right) :: \left(z : C\right)} \rightarrow \mathsf{L}$$

$$\frac{\Sigma;\Delta \vdash^{k,t}_{s,c} P :: \left(z : \bullet^{\tau}A\right) \quad t + \tau = s}{\Sigma;\Delta, v : \alpha^{\tau}1 \vdash^{k,t}_{s,c} \left(z \leftarrow \mathsf{put}\ v; P\right) :: \left(z : \overrightarrow{\alpha}^{\tau}A\right)} \rightarrow \mathsf{R}$$

$$\frac{\Sigma;\Delta, x : \bullet^{\tau}A \vdash^{k,t}_{s,c} P :: \left(z : C\right) \quad t + \tau = s}{\Sigma;\Delta, v : \alpha^{\tau}1, x : \overleftarrow{\alpha}^{\tau}A \vdash^{k,t}_{s,c} \left(x \leftarrow \mathsf{put}\ v; P\right) :: \left(z : C\right)} \leftarrow \mathsf{L}$$

$$\frac{\Sigma;\Delta, v : \alpha^{\tau}1 \vdash^{k,t}_{s,c} P :: \left(z : \bullet^{\tau}A\right) \quad t + \tau = s}{\Sigma;\Delta \vdash^{k,t}_{s,c} \left(v \leftarrow \mathsf{get}\ z; P\right) :: \left(z : \overleftarrow{\alpha}^{\tau}A\right)} \leftarrow \mathsf{R}$$

Getting a value from an input channel (→L) is, in this calculus, equivalent to splitting the channel typed $x : \overrightarrow{\alpha}^{\tau}A$ into two: a short internal value $v : \alpha^{\tau}1$ lasting until the end of the current cycle and a continuation channel $x : \bullet^{\tau} A$ with no current value, but carrying values for future cycles.

The 4 rules represent all possibilities among either with input (x) or output (z) channels and either $\rightarrow$ or $\leftarrow$, all of them using the same split/merge mechanism between v and x or z. Fig. 4.4 shows graphically the flow of data according to all rules.

4.3.8 Recursion/Loop - μ

$$\frac{\Sigma; \Delta, x : A\left[(\mu y.A)/y\right] \vdash^{k,t}_{s,c} P :: \left(z : C\right)}{\Sigma; \Delta, x : \mu y.A \vdash^{k,t}_{s,c} P :: \left(z : C\right)} \mu \text{ L}$$

$$\frac{\Sigma; \Delta \vdash^{k,t}_{s,c} P\left[(L : P)/L\right] :: \left(x : A\left[(\mu y.A)/y\right]\right)}{\Sigma; \Delta \vdash^{k,t}_{s,c} \left(L : P\right) :: \left(x : \mu y.A\right)} \mu \text{ R}$$

Recursive types can be only constructed from recursive processes (μR). Recursive types get unrolled without the need of an action (μL). Since recursion is not a structural rule, graphical depiction is not useful here.

Recursion in types and recursion in process terms are implemented using a similar term substitution mechanism: for types $\mu y.A$ is replaced by $A\left[(\mu y.A)/y\right]$ and for terms $L : P$ is replaced by $P\left[(L : P)/L\right]$. This means when the type variable y is reached, the overall type becomes A again and when the process label L is reached the overall process becomes P again, generating a loop.

As an example of recursive type, $x : \mu y.\overrightarrow{Int}^c\overleftarrow{Int}^c y$ becomes $\overrightarrow{Int}^c\overleftarrow{Int}^c(\mu y.\overrightarrow{Int}^c\overleftarrow{Int}^c y)$, which is equivalent to the infinite type $\overrightarrow{Int}^c\overleftarrow{Int}^c\overrightarrow{Int}^c\overleftarrow{Int}^c\cdots$. As an example of recursive process, L: $v \leftarrow$ get x; $z \leftarrow$ put v; clock; L becomes $v \leftarrow$ get x; $z \leftarrow$ put v; clock; (L: $v \leftarrow$ get x; $z \leftarrow$ put v; clock; L), which is the infinite process

$$v \leftarrow \texttt{get}\ x;\ z \leftarrow \texttt{put}\ v;\ \texttt{clock};\ v \leftarrow \texttt{get}\ x;\ z \leftarrow \texttt{put}\ v;\ \texttt{clock};\cdots.$$

4.3.9 Parallel Composition - ⊗

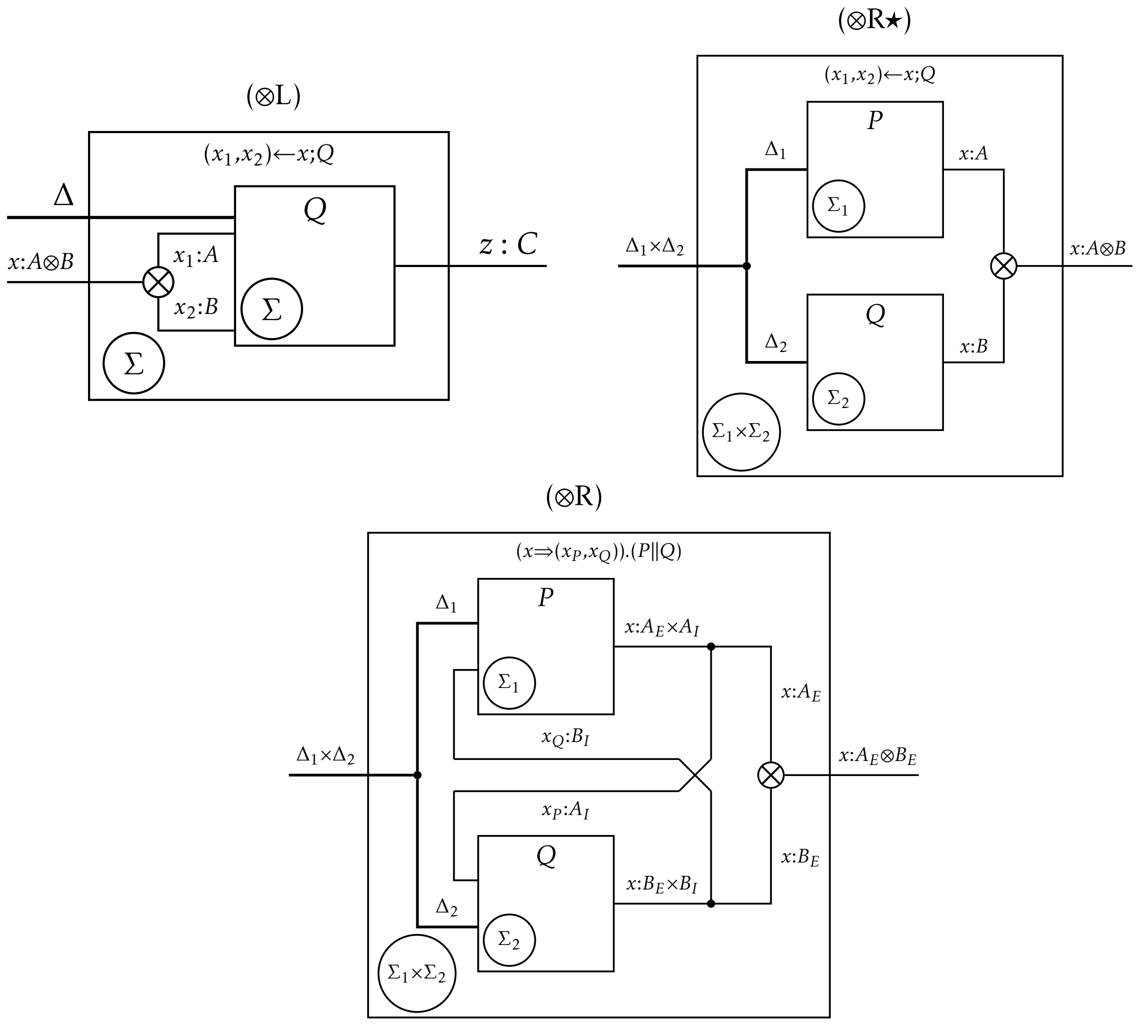


Figura 4.5: Graphical representation of ⊗ rules

$$\frac{\begin{array}{c}\Sigma_1;\Gamma,\Delta_1,x_Q:B_I \vdash^{k,t}_{s,c} P :: \left(x_P : A_E \times A_I\right)\\ \Sigma_2;\Gamma,\Delta_2,x_P:A_I \vdash^{k,t}_{s,c} Q :: \left(x_Q : B_E \times B_I\right)\end{array}}{\Sigma_1\times\Sigma_2;;\Delta_1\times\Delta_2 \vdash^{k,t}_{s,c} \left(\left(x\to\left(x_P,x_Q\right)\right).\left(P \parallel Q\right)\right) :: \left(x : A_E\otimes B_E\right)}\otimes\ \mathsf{R}$$

$$\frac{\Sigma_1;\Gamma,\Delta_1 \vdash^{k,t}_{s,c} P :: \left(x:A\right) \quad \Sigma_2;\Gamma,\Delta_2 \vdash^{k,t}_{s,c} Q :: \left(x:B\right)}{\Sigma_1\times\Sigma_2;;\Delta_1\times\Delta_2 \vdash^{k,t}_{s,c} \left(P \parallel Q\right) :: \left(x : A\otimes B\right)}\otimes\ \mathsf{R}\ \star$$

The linear logic operator ⊗ is used in TST to define arbitrary parallel composition between processes. Compared to ⊗ in IST, which is defined only for processes that are independent of each other, TST modifies the rule to also allow internal communication among the processes (using the merge operator $\times$) making the $\otimes$ rule general enough to represent both independent and dependent parallelism.

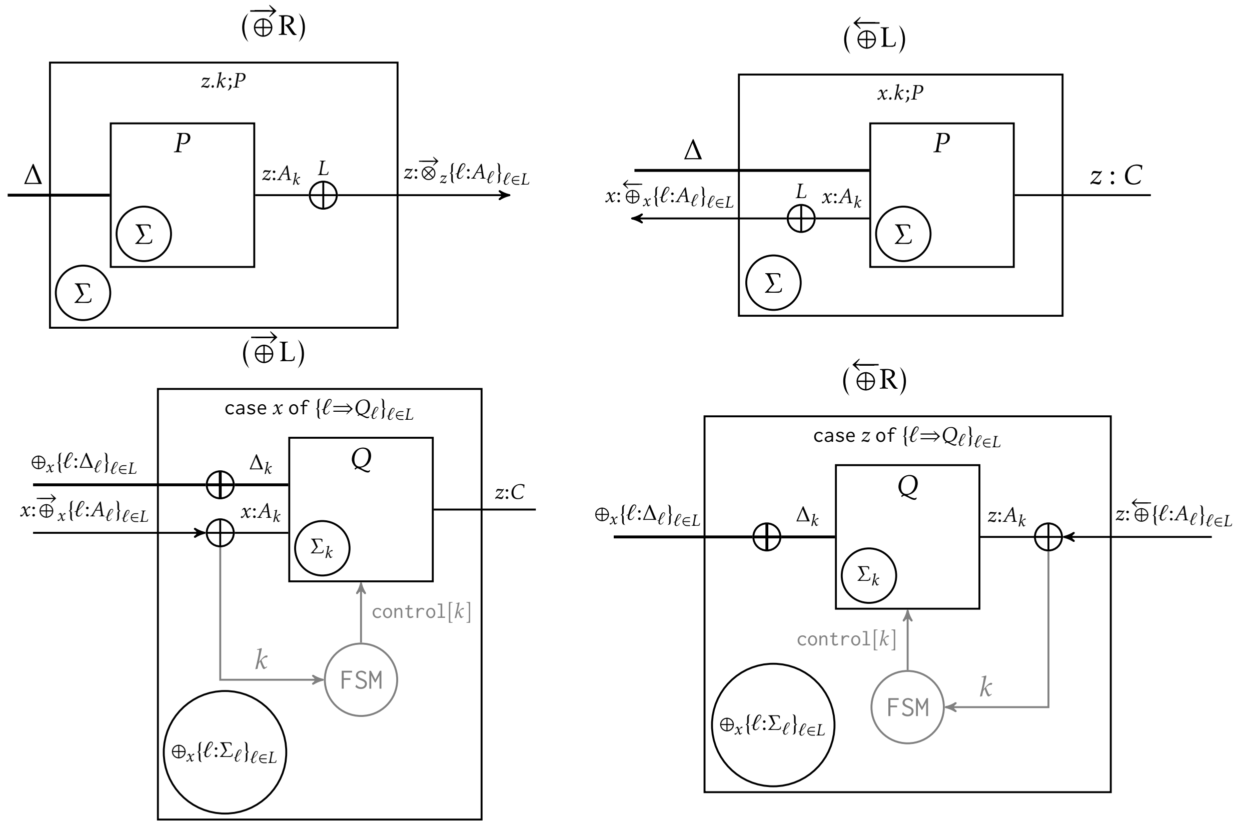
Logically speaking, the rule is a mixture of the IST's $\otimes$ rule (independent parallelism) with the *cut* rule (communication, or dependent parallelism). Although this gives the rule $\otimes R$ an unbalanced expressive power, it allows the calculus to compose processes more liberally, allowing for more powerful optimizations. Furthermore, parallelism in hardware design and any other concurrent system is so important that it makes sense for the rule to be "overpowered". Nevertheless, type preservation and global progress will still hold with this change.

The rule $\otimes R\star$ is a derivation of $\otimes R$, representing parallelism without internal communication, depicted by Fig. 4.5b. In this case, the outputs of P and Q, A and B respectively, become part of the output $A \otimes B$ completely. Note that both contexts are merged, implying two parallel processes can still share channels and resources even though there is no internal communication.

The more general $\otimes R$ rule (see Fig. 4.5a for better understanding) is more elaborate. The outputs of P ($A_E \times A_I$) and Q ($B_E \times B_I$) are, instead of being completely forwarded as output as in $\otimes R\star$, split into internal and external channels according to the definition of channel merge (Def. 2). The internal output of P (A_I) becomes input of Q and the internal output of Q (B_I) becomes input of P, while the external output of both become the output of the composition ($A_E \otimes B_E$).

The left rule $\otimes L$, in Fig. 4.5c, does not care if the input is generated by dependent or independent parallelism. It decomposes the channel into two separate channels, assigning them new names.

4.3.10 Choice operators



$(\overrightarrow{\oplus}\mathrm{R})$
$z.k;P$
P
Δ
$z{:}A_k$
L
$z{:}\overrightarrow{\otimes}_z\{\ell{:}A_\ell\}_{\ell\in L}$
Σ
Σ
$(\overleftarrow{\oplus}\mathrm{L})$
$x.k;P$
Δ
P
$z : C$
$x{:}\overleftarrow{\oplus}_x\{\ell{:}A_\ell\}_{\ell\in L}$
L
$x{:}A_k$
Σ
Σ
$(\overrightarrow{\oplus}\mathrm{L})$
case x of $\{\ell{\Rightarrow}Q_\ell\}_{\ell\in L}$
$\oplus_x\{\ell{:}\Delta_\ell\}_{\ell\in L}$
Δ_k
Q
$z{:}C$
$x{:}\overrightarrow{\oplus}_x\{\ell{:}A_\ell\}_{\ell\in L}$
$x{:}A_k$
Σ_k
control$[k]$
k
FSM
$\oplus_x\{\ell{:}\Sigma_\ell\}_{\ell\in L}$
$(\overleftarrow{\oplus}\mathrm{R})$
case z of $\{\ell{\Rightarrow}Q_\ell\}_{\ell\in L}$
Q
$\oplus_x\{\ell{:}\Delta_\ell\}_{\ell\in L}$
Δ_k
$z{:}A_k$
$z{:}\overleftarrow{\oplus}\{\ell{:}A_\ell\}_{\ell\in L}$
Σ_k
control$[k]$
k
FSM
$\oplus_x\{\ell{:}\Sigma_\ell\}_{\ell\in L}$

Figura 4.6: Graphical representations of $\oplus$ rules

$$\frac{(k\in L)\quad \Sigma;\Delta \vdash^{k,t}_{s,c} P :: \big(z : A_k\big)}{\Sigma;\Delta \vdash^{k,t}_{s,c} \big(z.k;P\big) :: \big(z : \overrightarrow{\oplus}_z\{\ell : A_\ell\}_{\ell\in L}\big)}\ \overrightarrow{\oplus}\ \mathsf{R}$$

$$\frac{(\forall\ell\in L)\quad \Sigma_\ell;\Delta_\ell, x : A_\ell \vdash^{k,t}_{s,c} Q_\ell :: \big(z : C\big)}{\oplus_x\{\ell : \Sigma_\ell\}_{\ell\in L};\oplus_x\{\ell : \Delta_\ell\}_{\ell\in L}, x : \overrightarrow{\oplus}_x\{\ell : A_\ell\}_{\ell\in L} \vdash^{k,t}_{s,c} \big(\mathsf{case}\ x\ \mathsf{of}\ \{\ell \Rightarrow Q_\ell\}_{\ell\in L}\big) :: \big(z : C\big)}\ \overrightarrow{\oplus}\ \mathsf{L}$$

$$\frac{(\forall\ell\in L)\quad \Sigma_\ell;\Delta_\ell \vdash^{k,t}_{s,c} Q_\ell :: \big(z : A_\ell\big)}{\oplus_z\{\ell : \Sigma_\ell\}_{\ell\in L};\oplus_z\{\ell : \Delta_\ell\}_{\ell\in L} \vdash^{k,t}_{s,c} \big(\mathsf{case}\ z\ \mathsf{of}\ \{\ell \Rightarrow Q_\ell\}_{\ell\in L}\big) :: \big(z : \overleftarrow{\oplus}_z\{\ell : A_\ell\}_{\ell\in L}\big)}\ \overleftarrow{\oplus}\ \mathsf{R}$$

$$\frac{(k\in L)\quad \Sigma;\Delta, x : A_k \vdash^{k,t}_{s,c} P :: \big(z : C\big)}{\Sigma;\Delta, x : \overleftarrow{\oplus}_x\{\ell : A_\ell\}_{\ell\in L} \vdash^{k,t}_{s,c} \big(x.k;P\big) :: \big(z : C\big)}\ \overleftarrow{\oplus}\ \mathsf{L}$$

The type of the choice operations, similar to get and put operations, have an arrow representing the direction flow of information, which can be from outside to inside or from inside to outside, depending on whether the channel is used or provided by the process and if the choice is internal (made by the process itself) or external (made by another process). The operator $\overrightarrow{\oplus}$ describes external choice if it is an input channel and internal choice if it is an output channel, while $\overleftarrow{\oplus}$ describes the opposite: internal choice if it is an input channel and external choice if it is an output channel.

Since the choice is interpreted as a message, the operators carry, as an index, the name of the channel on which the decision was made (the c in $\overleftrightarrow{\otimes}_c$) so if, at the same time, two choice types carry the same index, say $x : A \overleftrightarrow{\otimes}_c B$ and $y : C \overleftrightarrow{\otimes}_c D$, this means the decisions are synchronized, meaning $x : A$ implies $y : C$ and $y : D$ implies $x : B$, without $x : A$ and $y : D$ or $x : B$ and $y : C$ being possible.

The rules $\overrightarrow{\otimes}R$ (Fig. 4.6a) and $\overleftarrow{\otimes}L$ (Fig. 4.6b) represent processes making a decision internally and sending the decision as a message through the channel. As processes decide internally, they do not need to prepare for all the possible choices, instead it needs to prepare itself only for the chosen type A_k.

The rules $\overrightarrow{\otimes}L$ (Fig. 4.6c) and $\overleftarrow{\otimes}R$ (Fig. 4.6d) represent a process receiving a decision from an external process. In this case, the process receiving the decision must be ready for each one of the possible choices. In both of these rules, the input contexts are expressed as $\oplus_x\{\ell : \Sigma_\ell\}_{\ell \in L}$ and $\oplus_x\{\ell : \Delta_\ell\}_{\ell \in I}$, a notation that expresses the fact some input channels and resources may interact according to the same decision carried by x (because of the nature of multicasting), in which case they also must change accordingly.

The case operation is depicted (in Fig. 4.6c and 4.6d) uses a *finite state machine (FSM)* module which was not depicted previously (more about that in Chapter 5). The FSM takes as input the decision and updates control signals which make process Q operate as Q_ℓ for any decision ℓ (in this case, Q_k). Using FSMs to store and update the state of processes is extremely common in hardware design, but is abstracted away in H-Calculus. The consequences of hiding state control will be explained in detail in Chapter 5.

4.3.11 Intra-Cycle Signal

$$\frac{\vdash^{F} e : T \quad (s > t + \tau) \quad (\tau \geq 0)}{-;- \vdash^{k,t}_{s,c} \left(\mathtt{Sig}(\tau, y \leftarrow e)\right) :: \left(y : \bullet^{\tau} T^{s-(t+\tau)} 1\right)}\ \mathtt{Signal\text{-}1}$$

$$\frac{\overset{F}{\vdash} e : T \quad (s > t + \tau) \quad (\tau < 0)}{-;- \vdash^{k,t}_{s,c} \left(\mathtt{Sig}(\tau, y \leftarrow e)\right) :: \left(y : T^{s-t} 1\right)}\ \texttt{Signal-2}$$

`Sig`s are unique processes that live inside one cycle and carry functional values $e : T$ with them — for example, $5 : \mathtt{Int}$, $\mathtt{false} : \mathtt{Bool}$, or any other finite data structure. We use a functional sequent $\overset{F}{\vdash} e : T$ to check that the value e is well typed according to some simply-typed function type scheme similar to Def. 12, capable of type checking simple values and and functions (of functional type $\tau \to \sigma$).

Rule `Signal-1` models intra-cycle values that become stable after τ units of time from the start of the cycle, while rule `Signal-2` a value stabilized some time ago, meaning τ is negative.

4.3.12 Combinational Circuit

$$\frac{\overset{F}{\vdash} f : T_1 \to \cdots \to T_n \to T_{out} \quad (s > t + \mathsf{max}(d_1, \cdots, d_n) + p) \quad (p > 0)}{\begin{array}{c} -; x_1 : \bullet^{d_1} T_1^{s-(t+d_1)} 1, \cdots, x_n : \bullet^{d_n} T_n^{s-(t+d_n)} 1 \vdash^{k,t}_{s,c} \mathsf{Comb}(f, p, y \leftarrow (x_1, \cdots, x_n)) \\ :: \left(y : \bullet^{\mathsf{max}(d_1,\cdots,d_n)} \bullet^{p} T_{out}^{s-(t+\mathsf{max}(d_1,\cdots,d_n)+p)} 1\right) \end{array}}\ \mathsf{Comb}$$

Combinational processes compute within one cycle. They both consume and produce intra-cycle values. They represent a pure functional application being elevated to the realm of hardware processes.

Input signals arrive at different instants ($x_i : \bullet^{d_i} T_i^{s-(t+d_i)} 1$), so the output signal must take into account the combinational machine only starts to react to correct values after all input values are stable $\mathsf{max}(d_1, \cdots, d_n)$, after that a maximum possible delay is added and the output type is formed ($\bullet^{\mathsf{max}(d_1,\cdots,d_n)} \bullet^{p} T_{out}^{s-(t+\mathsf{max}(d_1,\cdots,d_n)+p)} 1$). As long as the function f is functionally well typed and the output becomes stable before the end of the cycle, the combinational the process is well typed.

4.3.13 Register

$$\frac{(s > t + \tau)}{-; x : \bullet^{\tau} T^{s-(t+\tau)} 1 \vdash^{k,t}_{s,c} \left(\mathsf{Reg}(y \leftarrow x)\right) :: \left(y : \bullet^{s-t} T^{s} 1\right)}\ \mathsf{Reg}$$

The `Reg` rule models how an intra-cycle signal interacts correctly with a register. The register takes a signal from the current cycle $x : \bullet^{\tau} T^{s-(t+\tau)}$, and forwards it to the next cycle $y : \bullet^{s-t} T^{s} 1$ as an output. As long as the input value gets stable before the end of the cycle, the register is well-typed.

4.3.14 Tick/Delay

$$\frac{\Sigma;\Delta \vdash_{s,c}^{k,t+\tau} P :: (z : C) \quad t+\tau \geq s}{\bullet^{\tau}\Sigma;\bullet^{\tau}\Delta \vdash_{s,c}^{k,t} (\texttt{tick}\ \tau; P) :: (z : \bullet^{\tau} C)}\ \texttt{tick}$$

The `tick` operation means that the process recognizes that time has passed for itself and for all other processes. Even though this rule represents an isolated process ticking, every process in the system must tick together, which is why input channels and resources also need to advance in time. After the `tick`, the time is updated from t to $t+\tau$ where $t+\tau$ cannot surpass the clock cycle itself. Logically speaking this rule is both left and right and proofs of preservation and global progress will highly depend on the way it is defined.

4.3.15 Clock synchronization

$$\frac{\Sigma;\Delta \vdash_{s,c}^{k+1,0} P :: (z : C)}{\Sigma;\Delta \vdash_{s,c}^{k,s} (\texttt{clock}; P) :: (z : C)}\ \texttt{clock}$$

Similar to the way `tick` is defined, the `clock` event is also experienced by all processes at the same time, synchronizing every action and state. Operationally, the `clock` rule resets the intra-period timer from s to 0 and increments the clock count k. Notice how the `clock` rule resets the timer but does not advance time.

4.3.16 Resource instantiation

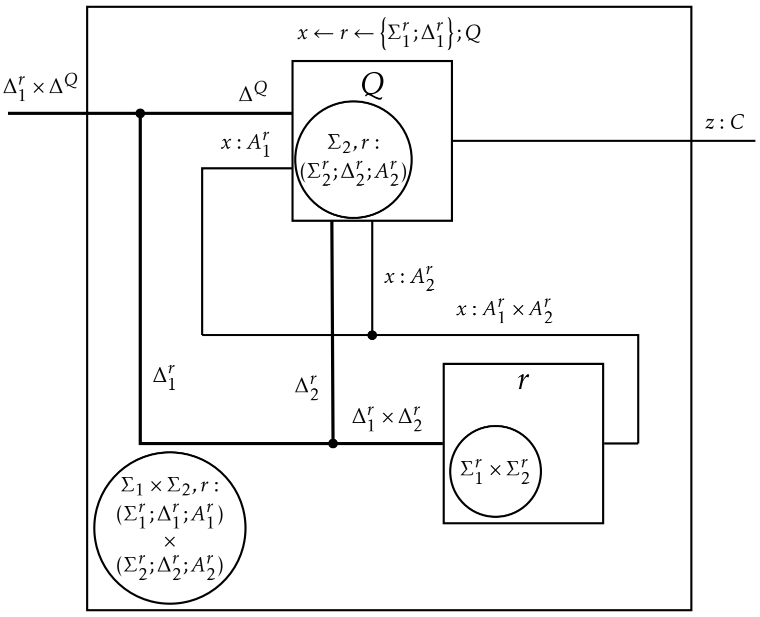

$x \leftarrow r \leftarrow \{\Sigma_1^r;\Delta_1^r\};Q$
$\Delta_1^r \times \Delta^Q$
Δ^Q
Q
$z:C$
$x:A_1^r$
$\Sigma_2, r:$
$(\Sigma_2^r;\Delta_2^r;A_2^r)$
$x:A_2^r$
$x:A_1^r \times A_2^r$
Δ_1^r
Δ_2^r
r
$\Delta_1^r \times \Delta_2^r$
$\Sigma_1^r \times \Sigma_2^r$
$\Sigma_1 \times \Sigma_2, r:$
$(\Sigma_1^r;\Delta_1^r;A_1^r)$
$\times$
$(\Sigma_2^r;\Delta_2^r;A_2^r)$

Figura 4.7: Graphical representation of Use

$$\frac{\Sigma^Q, r:(\Sigma_2^r;\Delta_2^r;A_2^r);\Delta^Q, x:A_1^r \mid\!\frac{k,t}{s,c}\, Q :: \left(z:C\right)}{\Sigma_1^r \times \Sigma^Q, r:(\Sigma_1^r;\Delta_1^r;A_1^r)\times(\Sigma_2^r;\Delta_2^r;A_2^r);\Delta_1^r \times \Delta^Q \mid\!\frac{k,t}{s,c} \left(x \leftarrow r \leftarrow \{\Sigma_1^r;\Delta_1^r\};Q\right) :: \left(z:C\right)}\ \texttt{Use}$$

The resource instantiation rule is similar to the *cut* rule, the difference being that the *cut* rule spawns a *process term P* while this rule initializes an `idle` resource r. Similar to *cut*, the input contexts are split, to allow for channel and resource sharing.

This rule allows for partial use of resources, which means that Q may or may not fully interact with r. Operationally, this means, the resource type of r is split into two, the first one $(\Sigma_1^r;\Delta_1^r;A_1^r)$ representing how Q will use it, and the second one $(\Sigma_2^r;\Delta_2^r;A_2^r)$ representing the "rest of interaction" needed for r to be completely satisfied. In the case of Q interacting completely with r, $(\Sigma_2^r;\Delta_2^r;A_2^r)$ would be $(-;-;1)$.

4.3.17 Main instantiation

$$\frac{(\forall(((\sigma := P_\sigma):R_\sigma)\in\Sigma).\texttt{ext}(P_\sigma,R_\sigma)) \quad \Sigma;\Delta \mid\!\frac{0,0}{s,c}\, P :: \left(z:C\right)}{\Sigma;\Delta \mid\!\frac{0,0}{s,c}\, \texttt{Main}(P) :: \left(z:C\right)}\ \texttt{Main}$$

The `Main` rule determines that the main process, the highest one in the hierarchy tree, which contains all the channels and all the resources of the system, in addition to being well-typed $(\Sigma;\Delta \mid\!\frac{0,0}{s,c}\, P :: (z:C))$, must ensure that every resource $\sigma \in \Sigma$ is a complete and correct instantiation of their respective process definitions $(\forall((\sigma := P_\sigma):R_\sigma)\in\Sigma).\texttt{ext}(P_\sigma,R_\sigma))$.

The Main rule is necessary because, even though incomplete interactions with resources is permitted to enable expressive resource sharing, all resources must be completely instantiated in the end.

4.4 Properties of the Type System

Both semantic and typing rules need to be harmoniously related to each other for the entire system to be useful. Two properties are crucial:

- Every well typed system — constructed from the typing rules defined in Def. 29 — must, always, be able to evolve through time — i.e., must match one of the semantic rule patterns described in Def. 22. This property is called **global progress**, and it ensures that no well-typed system is ever going to reach a deadlock state.
- Every time a semantic rule is applied to a well-typed system, the resulting system — after the rule is applied — must, not only be well-typed, but also have exactly the same type as before. This property is called **type preservation** and it ensures that we can trust that our types are not going to "change"throughout the execution of the system (in other words, we can *trust* our types).

To define these two properties formally we first need to understand what does it mean for a system to be well-typed. We know how to type individual processes, but semantic rules work with configurations (Def. 18) instead of individual processes. This is where *configuration typing rules* comes in. The type of a configuration informs us the type of the system during an snapshot of its execution. We begin by defining the *configuration type sequent* the mathematical structure that contains the type of the configuration.

Definition 30 (Configuration Type Sequent) The configuration type sequent is the object

$$\Sigma^I;\Delta^I \models \mathcal{C} :: \left(\Sigma^O;\Delta^O\right)$$

where Σ^I and Δ^I are *input contexts,* that must be provided for configuration $\mathcal{C}$ to execute correctly and Σ^O and Δ^O are *output contexts* that are provided by the configuration $\mathcal{C}$ during correct execution. Σ^I and Σ^O are resource contexts and Δ^I and Δ^O are channel contexts.

The configuration type sequent, different from the process type sequent, contains multiple output channels, Δ^O, and also contains output resources, Σ^O, since a configuration contains multiple processes, and resources, computing simultaneously. The typing rules for configuration are defined in Def. 31, with an accompanying set of illustrations.

Definition 31 (Configuration Typing) The set of all configuration typing rules:

$$\frac{\Sigma_1^I;\Delta_1^I \models \mathcal{C}_1, \underline{\mathtt{env}}(s,c,k,t) :: \left(\Sigma_1^O;\Delta_1^O\right) \quad \Sigma_2^I;\Delta_2^I \models \mathcal{C}_2, \underline{\mathtt{env}}(s,c,k,t) :: \left(\Sigma_2^O;\Delta_2^O\right)}{\Sigma_1^I \times \Sigma_2^I;\Delta_1^I \times \Delta_2^I \models \mathcal{C}_1\mathcal{C}_2, \underline{\mathtt{env}}(s,c,k,t) :: \left(\Sigma_1^O\Sigma_2^O;\Delta_1^O\Delta_2^O\right)}\,\mathtt{Compose}$$

$$\frac{\Sigma_C;\Delta\Delta_C \models \mathcal{C} :: (\Sigma_C, r:R;\Delta_C, x:A)}{-;\Delta \models \mathtt{Closed}\{\,\mathcal{C}\,\} :: (-;x:A)}\,\mathtt{Closed}$$

$$\frac{\Sigma;\Delta \vdash_{s,c}^{k,0} \mathtt{Main}(P) :: \left(x:A\right)}{-;\Delta \models \mathtt{Main}(P), \underline{\mathtt{env}}(s,c,0,0) :: (-;x:A)}\,\mathtt{Main}$$

$$\frac{[\Sigma]^{+T};[\Delta]^{+T} \vdash_{s,c}^{k,t} P :: \left(x:[A]^{+T}\right) \quad \mathtt{inst}((\Sigma;\Delta;A),R) \quad (T = k\times s+t)}{\Sigma;\Delta \models \mathtt{proc}\left(r \,\middle|\, P \,\middle|\, x\right), \underline{\mathtt{env}}(s,c,k,t) :: (r:R;x:A)}\,\mathtt{proc}$$

$$\frac{\mathtt{inst}(\mathtt{DEF}(P),R)}{-;- \models \mathtt{idle}(r := P), \underline{\mathtt{env}}(s,c,k,t) :: (r:R;-)}\,\mathtt{idle}$$

Some interesting aspects to note about these configuration typing rules:

- The rule Compose does not connect channels between two configurations, all of the connections happen within the Closed rule. The reason why we separate gathering from connecting channels and resources is that, since the h-calculus permits channels to connect to multiple components, we are never sure whether a given channel is fully connected or not. The Closed object is used to inform that, within the given configuration, every channel must be fully connected, meaning its type must be completely satisfied.
- The proc rule does take into account the passing of time. This is crucial for type preservation because, although time passes, the configuration type will stay the same, considering the beginning of time $(k,t) = (0,0)$.

Now we can define type preservation and global progress using configuration types.

Theorem 1 (Preservation). *If* $\Sigma^I;\Delta^I \models \mathcal{C} :: \left(\Sigma^O;\Delta^O\right)$ *and* $\mathcal{C} \to \mathcal{D}$ *then* $\Sigma^I;\Delta^I \models \mathcal{D} :: \left(\Sigma^O;\Delta^O\right)$.

The complete proof of preservation is in the Appendix A. This proof consists of finding the type of $\Sigma^I;\Delta^I \models \mathcal{C} :: \left(\Sigma^O;\Delta^O\right)$ and the type of $\Sigma^{I\prime};\Delta^{I\prime} \models \mathcal{D} :: \left(\Sigma^{O\prime};\Delta^{O\prime}\right)$ for each one of the possible semantic rule cases $\mathcal{C} \to \mathcal{D}$ defined in Def. 20) and check if $\Sigma^I = \Sigma^{I\prime}$, $\Delta^I = \Delta^{I\prime}$, $\Sigma^O = \Sigma^{O\prime}$ and $\Delta^O = \Delta^{O\prime}$.

Theorem 2 (Global Progress). *If* $-;\Delta \models$ `Closed`$\{\,\mathcal{C}\,\} :: (r : R; x : A)$, *then*

1. $\mathcal{C} \to \mathcal{D}$, *for some* $\mathcal{D}$, *or*
2. *is communicating through* $c \in \Delta$ *or* x, *or*
3. *does not have* `proc` *objects (computation is over).*

The complete proof of progress is also in the Appendix A. This is proof is more intricate than the preservation one. The main idea is to show that because `Closed`$\{\,\mathcal{C}\,\}$ is a well-typed configuration either computation is over (case 3) or it is interacting with channels or resources. Because the configuration is closed, interacting with resources means the resource is internal, in which case an operational step should occur (case 1). If the configuration interacts with channels, the channel is either internal or external. If the channel is external case 2 applies. If the channel is internal, the type system is designed in a way that every action is met by its correct reaction, resulting in an operational step (case 1).

Capítulo 5

H-Calculus for High-Level Synthesis

This chapter discusses the consequences of using the h-calculus as an Intermediate Representation (IR) in a High-Level Synthesis (HLS) system. We will also compare the use of h-calculus to Control Dataflow Graphs (CDFGs), the most commonly used IR in HLS, pointing out both advantages and disadvantages.

We will discuss how the h-calculus impacts the three steps of HLS (Fig. 5.1, shown in Chapter 1) separately.

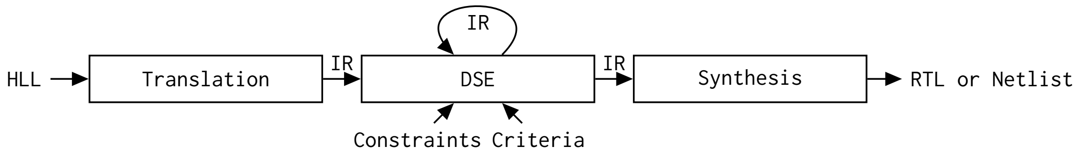


Figura 5.1: Simplified High-Level Synthesis flow

5.1 Translation

Translation is the first step of HLS. Translation infers an IR from a high-level specification, written in a High-Level Language (HLL), without efficiency or optimization concerns, since these are the responsibility of the Design Space Exploration step.

The effectiveness and complexity of translation techniques depend on the HLL and IR choices. This thesis proposes using a functional typed programming language as input and the h-calculus as the output of the translation step. We shall compare it to C-like languages as input and CDFGs as output, which are typical choices in traditional HLS.

High-Level Language Choice

Ideally, any HLL could be transformed into IR. In practice, however, the language choice can either complicate or facilitate the translation step. There are some compelling arguments in favor of *functional languages* over *sequential languages* as input for HLS. The most compelling being related to *concurrency inference*. Both functional and sequential code do not natively understand the notion of concurrency, meaning it needs to be inferred by translation techniques.

Inferring concurrency from sequential specifications is a challenging problem without an efficient solution. This leads to specifications that are, in general, not as parallel as hardware designers would want. After transforming sequential code into CDFGs, it is still difficult to infer concurrency from CDFGs, meaning concurrency inefficiency will be carried all the way from the high-level specification up to the HLS result. It is common in traditional HLS systems to extend their sequential languages with explicit concurrency constructs (such as `par`), but the result is not ideal since now the designers needs to worry about concurrency.

Inferring concurrency from functional specifications is as simple as it gets: given a functional application $f(x,y,z)$, we know that the terms x, y and z can be evaluated in any sequence, or in parallel, without correctness issues. Since the entire specification is composed of abstractions and applications, this is enough to infer system-wide concurrency. Functional specifications suffer from the opposite problem: they often need to be sequentialized, which is a simpler problem.

Intermediate Representation

Compared to extracting CDFGs from C-like specifications, extracting h-calculus specifications from functional languages is not as simple and does not have well-established solutions. The h-calculus, being closer to hardware, contains low-level details that require a great deal of inferring, which is alleviated by the fact that Translation does not need to output optimized results. For example, a translation step could consider every function application as a new resource running parallel and every recursion as sequential computation. This is a practical way to translate since we expect the DSE step to optimize any inefficiencies introduced by Translation.

Although translating C-like code to CDFGs does not require much inferring, the resulting CDFGs do not provide efficient mechanisms for verification and analysis, so it is a tradeoff between ease of analysis and ease of Translation.

Translation Scheme

This thesis does not provide a specific translation scheme because techniques still need to be understood better. However, we provide a plan, a set of ideas, that should efficiently translate from functional language to h-calculus despite concerns. These are:

The Functionality Operator: The Functionality Operator: an operator, denoted $\mathcal{F}$, that takes a well-typed process as input and returns a function as output

$$\mathcal{F}\left(\Sigma;\Delta \,\Big|\frac{k,t}{s,c}\, P :: \left(x : A\right)\right) \rightarrow f.$$

This operator should effectively transform temporal session types into algebraic datatypes and erase the notions of concurrency, sequence, time, resources, and channels from processes until only a function, a mapping from algebraic types to algebraic types, is left.

A translation procedure: opposite of the functionality operator, denoted $\mathcal{T}$, the translation should take a well-typed function as input and return a well-typed h-calculus process

$$\mathcal{T}\left(f\right) \rightarrow \Sigma;\Delta \,\Big|\frac{k,t}{s,c}\, P :: \left(x : A\right).$$

Types are transformed into Temporal Session Types and functions into h-calculus processes by inferring every characteristic not modeled by functions: concurrency, time, clock, resources. There are multiple possible processes for one function, but the translation procedure can choose any, even not optimized, as long as it is correct. Previous work [37, 38, 39] successfully transformed typed functions, using a Haskell-like language, into hardware: we need to translate the ideas to output h-calculus processes instead.

A correctness proof: a proof that that translation preserves functionality, $\mathcal{F}(\mathcal{T}(f)) = f$.

The main issue with this plan regards the proof of correctness. Ideally, we should define functionality and translation operations with this proof as an objective; otherwise, it may be difficult to prove since it is reduced to a *function extensionality* problem (proving two functions are equal), which is a challenging problem. Furthermore, $\mathcal{F}$'s outputs must be as simple as possible to allow DSE to perform automatic reasoning at the functional level efficiently; a functionality operator that outputs functions that look like a functional hardware description would not be helpful in this case. In summary, bridging the h-calculus with functions is challenging, but solving the problem can significantly improve the level of abstraction provided by HLS.

5.2 Design Space Exploration

Design Space Exploration is the step that analyses and manipulates IR descriptions until it finds a design that meets all the project's constraints. The exploration step is more general than performing a fixed sequence of transformations. DSE (Fig. 5.2) receives feedback information from multiple IR definitions and decides, based on it, the next transformations it will perform. This feedback information contains efficiency parameters and is crucial for the effectiveness of DSE.

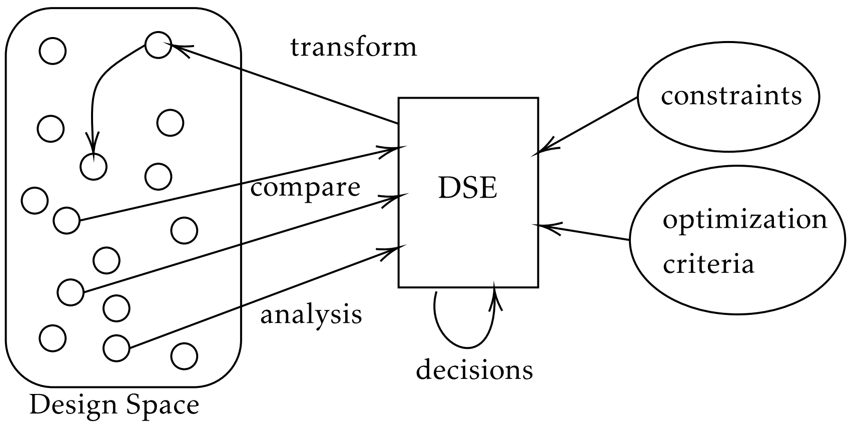


Figura 5.2: Design Space Exploration Schematics

For hardware DSE to be effective, feedback information must be easily fetchable — i.e., must be computationally light and fast — and must accurately model hardware at a low level, or else the information is not helpful for hardware analysis.

The h-calculus type system solves this by providing hardware-aware information easily fetchable from its types. In the case of Control Dataflow Graphs, they are interpreted as hardware but do not natively model hardware, meaning not all of the information extracted from it will be accurate. Furthermore, extraction of analytical information from CDFGs requires graph-crawling algorithms, which can be computationally heavy for large systems.

We do not provide one particular DSE system because we can implement them in so many ways. It can be as simple as merging resources to minimize area usage, multiplying resources to maximize throughput, or pipelining a sequential computation segment, or as advanced as using analytical data as input for a machine learning or genetic programming decision scheme. As long as the IR is easy to analyze, multiple schemes are possible.

An example is the best way to understand how the h-calculus aids DSE in practice. In the following example, we will emulate a DSE system. Our objective is to optimize an IR description according to specific design constraints and optimization criteria using h-calculus' types. We will also understand how DSE can be performed using CDFGs instead of h-calculus and compare both results.

Example 5.1 (Dot Product Design Exploration). We will consider the hardware analysis of a *dot product* implementation as an example. We will assume a designer wrote the definition of dot product in a high-level language. The *translation* step then transformed it into an h-calculus process, resulting in the Dotp#1 definition (#1 for first version) described by Figure 5.3.

```
$\texttt{Dotp\#1} \overset{\text{def}}{=}$
$x_1 \leftarrow \text{get } in_1;$
$x_2 \leftarrow \text{get } in_2;$
$m_x \leftarrow \texttt{mul}_1 \leftarrow \{x_1, x_2\};$
$r_1 \leftarrow \texttt{reg}_1 \leftarrow m_x;$
$\texttt{tick } s; \texttt{ clock};$
$y_1 \leftarrow \text{get } in_1;$
$y_2 \leftarrow \text{get } in_2;$
$m_y \leftarrow \texttt{mul}_2 \leftarrow \{y_1, y_2\};$
$a_1 \leftarrow \texttt{add}_1 \leftarrow \{m_y, r_1\};$
$r_2 \leftarrow \texttt{reg}_2 \leftarrow a_1;$
$\texttt{tick } s; \texttt{ clock};$
$z_1 \leftarrow \text{get } in_1;$
$z_2 \leftarrow \text{get } in_2;$
$m_z \leftarrow \texttt{mul}_3 \leftarrow \{z_1, z_2\};$
$a_2 \leftarrow \texttt{add}_2 \leftarrow \{m_z, r_2\};$
$\texttt{tick } (\delta_{\texttt{mul}} + \delta_{\texttt{add}});$
$out \leftarrow \text{put } a_2; \text{ end } out$
```

First version

```
$\texttt{Dotp\#2} \overset{\text{def}}{=}$
$x_1 \leftarrow \text{get } in_1;$
$x_2 \leftarrow \text{get } in_2;$
$m_x \leftarrow \texttt{mul}_{123} \leftarrow \{x_1, x_2\};$
$r_1 \leftarrow \texttt{reg}_{12} \leftarrow m_x;$
$\texttt{tick } s; \texttt{ clock};$
$y_1 \leftarrow \text{get } in_1;$
$y_2 \leftarrow \text{get } in_2;$
$m_y \leftarrow \texttt{mul}_{123} \leftarrow \{y_1, y_2\};$
$a_1 \leftarrow \texttt{add}_{12} \leftarrow \{m_y, r_1\};$
$r_2 \leftarrow \texttt{reg}_{12} \leftarrow a_1;$
$\texttt{tick } s; \texttt{ clock};$
$z_1 \leftarrow \text{get } in_1;$
$z_2 \leftarrow \text{get } in_2;$
$m_z \leftarrow \texttt{mul}_{123} \leftarrow \{z_1, z_2\};$
$a_2 \leftarrow \texttt{add}_{12} \leftarrow \{m_z, r_2\};$
$\texttt{tick } (\delta_{\texttt{mul}} + \delta_{\texttt{add}});$
$out \leftarrow \text{put } a_2; \text{ end } out$
```

Second version

```
$\texttt{Dotp\#3} \overset{\text{def}}{=}$
$(x_1, y_1, z_1) \leftarrow in_1;$
$(x_2, y_2, z_2) \leftarrow in_2;$
$m_x \leftarrow \texttt{mul}_1 \leftarrow \{x_1, x_2\};$
$m_y \leftarrow \texttt{mul}_2 \leftarrow \{y_1, y_2\};$
$m_z \leftarrow \texttt{mul}_3 \leftarrow \{z_1, z_2\};$
$a_1 \leftarrow \texttt{add}_1 \leftarrow \{m_x, m_y\};$
$a_2 \leftarrow \texttt{add}_2 \leftarrow \{m_z, a_1\};$
$\texttt{tick } (\delta_{\texttt{mul}} + 2\delta_{\texttt{add}});$
$out \leftarrow \text{put } a_2; \text{ end } out$
```

Third version

```
$\texttt{Dotp\#4} \overset{\text{def}}{=}$
$m \leftarrow \texttt{mul}_{seq} \leftarrow \{in_1, in_2\};$
$\texttt{tick } \delta_{\texttt{seq}}; m_1 \leftarrow \text{get } m;$
$r_1 \leftarrow \texttt{reg} \leftarrow m_1;$
$\texttt{tick } s; \texttt{clock}; \texttt{tick } \delta_{\texttt{seq}};$
$m_2 \leftarrow \text{get } m;$
$a_1 \leftarrow \texttt{add} \leftarrow \{m_2, r_1\};$
$r_2 \leftarrow \texttt{reg} \leftarrow a_1;$
$\texttt{tick } s; \texttt{clock}; \texttt{tick } \delta_{\texttt{seq}};$
$m_3 \leftarrow \text{get } m;$
$a_2 \leftarrow \texttt{add} \leftarrow \{m_3, r_2\};$
$\texttt{tick } \delta_{\texttt{add}};$
$out \leftarrow \text{put } a_2; \text{end } out$
```

Fourth version

Figura 5.3: Different h-calculus processes for the Dotp function

$$
\begin{aligned}
&-; in_1 : \bullet^{\tau_1}\mathtt{Int}^{s-\tau_1}1, in_2 : \bullet^{\tau_2}\mathtt{Int}^{s-\tau_2}1\\
&\quad \vdash^{k,0}_{s,c} \mathsf{Add} :: \Big(out : \bullet^{\max(\tau_1,\tau_2)}\bullet^{\delta_{\mathsf{add}}}\overrightarrow{\mathtt{Int}}^{s-(\max(\tau_1,\tau_2)+\delta_{\mathsf{add}}}1\Big)\\
&-; in_1 : \bullet^{\tau_1}\mathtt{Int}^{s-\tau_1}1, in_2 : \bullet^{\tau_2}\mathtt{Int}^{s-\tau_2}1\\
&\quad \vdash^{k,0}_{s,c} \mathtt{Mul} :: \Big(out : \bullet^{\max(\tau_1,\tau_2)}\bullet^{\delta_{\mathtt{mul}}}\overrightarrow{\mathtt{Int}}^{s-(\max(\tau_1,\tau_2)+\delta_{\mathtt{mul}}}1\Big)\\
&-; in_1 : \mathtt{Int}^{s}\mathtt{Int}^{s}\mathtt{Int}^{s}1, in_2 : \mathtt{Int}^{s}\mathtt{Int}^{s}\mathtt{Int}^{s}1\\
&\quad \vdash^{k,0}_{s,c} \mathtt{Mul}_{\mathsf{seq}} :: \Big(out : \bullet^{s}\bullet^{s}\bullet^{\delta_{\mathsf{seq}}}\mathtt{Int}^{s-\delta_{\mathsf{seq}}}\bullet^{\delta_{\mathsf{seq}}}\mathtt{Int}^{s-\delta_{\mathsf{seq}}}\bullet^{\delta_{\mathsf{seq}}}\mathtt{Int}^{s-\delta_{\mathsf{seq}}}1\Big)
\end{aligned}
$$

$$
\begin{aligned}
&\Sigma_{\#1}; in_1 : \overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}1, in_2 : \overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}1\\
&\quad \vdash^{k,0}_{s,c} \mathtt{Dotp\#1} :: \Big(out : \bullet^{2s}\bullet^{\delta_{\mathtt{mul}}+\delta_{\mathsf{add}}}\overrightarrow{\mathtt{Int}}^{s-(\delta_{\mathtt{mul}}+\delta_{\mathsf{add}})}1\Big)\\
&\Sigma_{\#2}; in_1 : \overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}1, in_2 : \overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}1\\
&\quad \vdash^{k,0}_{s,c} \mathtt{Dotp\#2} :: \Big(out : \bullet^{2s}\bullet^{\delta_{\mathtt{mul}}+\delta_{\mathsf{add}}}\overrightarrow{\mathtt{Int}}^{s-(\delta_{\mathtt{mul}}+\delta_{\mathsf{add}})}1\Big)\\
&\Sigma_{\#3}; in_1 : \Big(\overrightarrow{\mathtt{Int}}^{s}1\Big)\otimes\Big(\overrightarrow{\mathtt{Int}}^{s}1\Big)\otimes\Big(\overrightarrow{\mathtt{Int}}^{s}1\Big)\\
&, in_2 : \Big(\overrightarrow{\mathtt{Int}}^{s}1\Big)\otimes\Big(\overrightarrow{\mathtt{Int}}^{s}1\Big)\otimes\Big(\overrightarrow{\mathtt{Int}}^{s}1\Big)\\
&\quad \vdash^{k,0}_{s,c} \mathtt{Dotp\#3} :: \Big(out : \bullet^{(\delta_{\mathtt{mul}}+2\delta_{\mathsf{add}})}\overrightarrow{\mathtt{Int}}^{s-(\delta_{\mathtt{mul}}+2\delta_{\mathsf{add}})}1\Big)\\
&\Sigma_{\#4}; in_1 : \overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}1, in_2 : \overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}\overrightarrow{\mathtt{Int}}^{s}1\\
&\quad \vdash^{k,0}_{s,c} \mathtt{Dotp\#4} :: \Big(out : \bullet^{4s}\bullet^{\delta_{\mathsf{seq}}+\delta_{\mathsf{add}}}\overrightarrow{\mathtt{Int}}^{s-(\delta_{\mathsf{seq}}+\delta_{\mathsf{add}})}1\Big)
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_{\#1} = {} & \mathtt{mul}_1 : (-; in_1 : \mathtt{Int}^{s}\bullet^{s}\bullet^{s}1, in_2 : \mathtt{Int}^{s}\bullet^{s}\bullet^{s}1; \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}\bullet^{s}\bullet^{s}1)\\
&, \mathtt{mul}_2 : (-; in_1 : \bullet^{s}\mathtt{Int}^{s}\bullet^{s}1, in_2 : \bullet^{s}\mathtt{Int}^{s}\bullet^{s}1; \bullet^{s}\bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}\bullet^{s}1)\\
&, \mathtt{mul}_3 : (-; in_1 : \bullet^{s}\bullet^{s}\mathtt{Int}^{s}1, in_2 : \bullet^{s}\bullet^{s}\mathtt{Int}^{s}1; \bullet^{s}\bullet^{s}\bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}1)\\
&, \mathtt{reg}_1 : (-; in : \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}\bullet^{s}\bullet^{s}1; \bullet^{s}\mathtt{Int}^{s}\bullet^{s}1)\\
&, \mathtt{reg}_2 : (-; in : \bullet^{s}\bullet^{\delta_{\mathtt{mul}}+\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathtt{mul}}+\delta_{\mathsf{add}})}\bullet^{s}1; \bullet^{s}\bullet^{s}\mathtt{Int}^{s}1)\\
&, \mathsf{add}_1 : (-; in_1 : \bullet^{s}\bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}\bullet^{s}1, in_2 : \bullet^{s}\mathtt{Int}^{s}\bullet^{s}1\\
&\qquad ; \bullet^{s}\bullet^{\delta_{\mathtt{mul}}+\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathtt{mul}}+\delta_{\mathsf{add}})}\bullet^{s}1)\\
&, \mathsf{add}_2 : \Big(-; in_1 : \bullet^{s}\bullet^{s}\bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}1, in_2 : \bullet^{s}\bullet^{s}\mathtt{Int}^{s}1\\
&\qquad ; \bullet^{s}\bullet^{s}\bullet^{\delta_{\mathtt{mul}}+\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathtt{mul}}+\delta_{\mathsf{add}})}1\Big)
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_{\#2} = {} & \mathtt{mul}_{123} : \Big(-; in_1 : \mathtt{Int}^{s}\mathtt{Int}^{s}\mathtt{Int}^{s}1, in_2 : \mathtt{Int}^{s}\mathtt{Int}^{s}\mathtt{Int}^{s}1\\
&\qquad ; \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}\bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}\bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}1\Big)\\
&, \mathtt{reg}_{12} : (-; in : \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}\bullet^{\delta_{\mathtt{mul}}+\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathtt{mul}}+\delta_{\mathsf{add}})}\bullet^{s}1; \bullet^{s}\mathtt{Int}^{s}\mathtt{Int}^{s}1)\\
&, \mathsf{add}_{12} : \Big(-; in_1 : \bullet^{s}\bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}\bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}1, in_2 : \bullet^{s}\mathtt{Int}^{s}\mathtt{Int}^{s}1\\
&\qquad ; \bullet^{s}\bullet^{\delta_{\mathtt{mul}}+\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathtt{mul}}+\delta_{\mathsf{add}})}\bullet^{\delta_{\mathtt{mul}}+\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathtt{mul}}+\delta_{\mathsf{add}})}1\Big)
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_{\#3} = {} & \mathtt{mul}_1 : (-; in_1 : \mathtt{Int}^{s}1, in_2 : \mathtt{Int}^{s}1; \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}1)\\
&, \mathtt{mul}_2 : (-; in_1 : \mathtt{Int}^{s}1, in_2 : \mathtt{Int}^{s}1; \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}1)\\
&, \mathtt{mul}_3 : (-; in_1 : \mathtt{Int}^{s}1, in_2 : \mathtt{Int}^{s}1; \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}1)\\
&, \mathsf{add}_1 : \Big(-; in_1 : \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}1, in_2 : \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}1\\
&\qquad ; \bullet^{\delta_{\mathtt{mul}}+\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathtt{mul}}+\delta_{\mathsf{add}})}1\Big)\\
&, \mathsf{add}_2 : \Big(-; in_1 : \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}}1, in_2 : \bullet^{\delta_{\mathtt{mul}}+\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathtt{mul}}+\delta_{\mathsf{add}})}1\\
&\qquad ; \bullet^{\delta_{\mathtt{mul}}+2\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathtt{mul}}+2\delta_{\mathsf{add}})}1\Big)
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_{\#4} = {} & \mathtt{mul}_{seq} : \Big(-; in_1 : \mathtt{Int}^{s}\mathtt{Int}^{s}\mathtt{Int}^{s}1, in_2 : \mathtt{Int}^{s}\mathtt{Int}^{s}\mathtt{Int}^{s}1\\
&\qquad ; \bullet^{2s}\bullet^{\delta_{\mathsf{seq}}}\mathtt{Int}^{s-\delta_{\mathsf{seq}}}\bullet^{\delta_{\mathsf{seq}}}\mathtt{Int}^{s-\delta_{\mathsf{seq}}}\bullet^{\delta_{\mathsf{seq}}}\mathtt{Int}^{s-\delta_{\mathsf{seq}}}1\Big)\\
&, \mathtt{reg} : \Big(-; in : \bullet^{2s}\bullet^{\delta_{\mathsf{seq}}}\mathtt{Int}^{s-\delta_{\mathsf{seq}}}\bullet^{\delta_{\mathsf{seq}}+\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathsf{seq}}+\delta_{\mathsf{add}})}\bullet^{s}1\\
&\qquad ; \bullet^{2s}\bullet^{s}\mathtt{Int}^{s}\mathtt{Int}^{s}1\Big)\\
&, \mathsf{add} : \Big(-; in_1 : \bullet^{2s}\bullet^{s}\bullet^{\delta_{\mathsf{seq}}}\mathtt{Int}^{s-\delta_{\mathsf{seq}}}\bullet^{\delta_{\mathsf{seq}}}\mathtt{Int}^{s-\delta_{\mathsf{seq}}}1, in_2 : \bullet^{2s}\bullet^{s}\mathtt{Int}^{s}\mathtt{Int}^{s}1\\
&\qquad ; \bullet^{2s}\bullet^{s}\bullet^{\delta_{\mathsf{seq}}+\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathsf{seq}}+\delta_{\mathsf{add}})}\bullet^{\delta_{\mathsf{seq}}+\delta_{\mathsf{add}}}\mathtt{Int}^{s-(\delta_{\mathsf{seq}}+\delta_{\mathsf{add}})}1\Big)
\end{aligned}
$$

Figura 5.4: Typing judgements for all `Dotp` versions and auxiliary definitions

The process definitions discussed in this example are in Figures 5.3, and auxiliary type definitions are in Figure 5.4. Our goals with this example are to:

- analyze and explore the design space for correct dot-product hardware designs, analyze them and discuss their advantages and disadvantages,
- show that even for simple descriptions, such as a dot-product, hardware analysis can get quite complicated, which is why h-calculus is helpful. There are many correct but drastically different architectures that need to be explored before reaching optimal results, and
- show how a semi or fully automatic system can use the information provided by types to make sense of the complexity, extract relevant information and, as a consequence, be able to choose optimizations more intelligently

.

Analysis and Design Space Exploration As previously stated, we assume the result in Figure 5.3 comes from the *translation* stage. The translation stage's objective is to transform the high-level input into *any* h-calculus definition as long as it type-checks and preserves the input's functionality. It does not matter, for this stage, if results are optimized or not, which is good because the algorithm can focus on being correct and as simple as possible.

Since we expect `Dotp#1` to be inefficient, we start by looking at its types to collect analytical information. By merging the types of $\mathtt{Mul}_1$, $\mathtt{Mul}_2$, and $\mathtt{Mul}_3$, we can tell they are compatible. Their merged type is

$$(-; in_1 : \mathtt{Int}^s\mathtt{Int}^s\mathtt{Int}^s 1, in_2 : \mathtt{Int}^s\mathtt{Int}^s\mathtt{Int}^s 1; \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}} \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}} \bullet^{\delta_{\mathtt{mul}}}\mathtt{Int}^{s-\delta_{\mathtt{mul}}} 1),$$

which is enough information to replace them all by only one multiplier, called $\mathtt{Mul}_{123}$. Similarly, we can replace $\mathtt{Reg}_1$ and $\mathtt{Reg}_2$ by $\mathtt{Reg}_{12}$, and $\mathtt{Add}_1$ and $\mathtt{Add}_2$ by $\mathtt{Add}_{12}$. After these optimizations, `Dotp#1` becomes `Dotp#2`. These optimizations do not change the structure of `Dotp#1`, it just makes the hardware module leaner, decreasing its area and cost

Suppose that, as designers, we want higher throughput. There are many ways of achieving this, but we will try parallelization, also known as unrolling, which is, in terms of TSTs, changing input types from

$$\overrightarrow{Int}^s\overrightarrow{Int}^s\overrightarrow{Int}^s 1$$

to

$$\left(\overrightarrow{Int}^s 1\right)\otimes\left(\overrightarrow{Int}^s 1\right)\otimes\left(\overrightarrow{Int}^s 1\right),$$

meaning all three values will arrive at the same time instead of in sequence. This optimization yields `Dotp#3`. Different from `Dotp#1`, all resources in $\Sigma_{\#3}$ are merge-incompatible, which is a side-effect of computing in parallel.

All three versions of `Dotp` so far assume that the processing time of multiplications $\delta_{\mathtt{mul}}$ to be less than the cycle duration $\delta_{\mathtt{mul}} < s$, but that might not be true. The value of s might be constrained, or designers might prefer faster clock frequencies (in which case s should be as small as possible). Assuming $s \leq \delta_{\mathtt{mul}}$, 1-cycle multipliers would not type check. A solution is to replace them with sequential pipelined multipliers $\mathtt{Mul}_{\mathtt{seq}}$ (Figure 5.4) instead, which, in our case, take 3 stages to finish one multiplication. The timings are such that $\delta_{\mathtt{seq}}$ is smaller than $\delta_{\mathtt{mul}}$ since it represents the computation time for the third stage of multiplication only. The result is a sequential `Dotp#4` that uses this new sequential multiplier.

Comparison

A summary of information collected from the types of all Dotp versions is in Table 5.1.

Resource Usage The sequential versions #2 and #4 use fewer resources than #1 and #3. The first version uses many resources because it is the version before resource sharing optimization, but it has no logical reason to use this amount of resources.

Version #2 provides the same advantages as #1 while using fewer resources, which is a case of objective improvement, independent of designers' optimization constraints. Version #3 uses more resources because it uses parallel computation: it performs computations in fewer cycles, but it uses more resources in turn.

Regarding resource usage, #2 and #4 are the best options. However, it is hard to measure which one is the best since we would have to estimate the resource cost of Mul compared to $\texttt{Mul}_{\text{seq}}$, which is unclear since Mul should use more combinational circuitry and $\texttt{Mul}_{\text{seq}}$ should use registers to carry information through the pipeline stages.

Throughput Considering output per cycle rate, the winner is #3 because it runs all the computation in one cycle, while others take 3 and 5. However, output per time — the output per cycle divided by the minimum period — requires a more detailed analysis.

#3 would indeed perform the computation in one s. However, the minimum possible value for s, in this case, would be $\delta_{\texttt{mul}} + 2\delta_{\text{add}}$, which could be a relatively large number, mainly because $\delta_{\texttt{mul}}$ should be much bigger than δ_{add} due to the complexity of multiplication.

To know which architecture would have the highest throughput, we would have to set relative values for $\delta_{\texttt{mul}}$, δ_{seq}F andδ_{add} G and calculate the throughputs. In Table 5.2, we show different possible cases, A and B, and their throughput. In case A, $\delta_{\texttt{mul}}$H being too big, version #4 will have a better throughput by allowing a faster clock speed. Version #1 and #2 would be especially slow in this case since they have to perform multiplication three times in sequence. In case B, δ_{seq}J is only slightly bigger than $\delta_{\texttt{mul}}$K, so versions #2 and #4 have very similar throughput, and version #3 would offer the best throughput.

Tabela 5.1: `Dotp` versions comparison

Version	`Mul`	$\texttt{Mul}_{seq}$	`Add`	`Reg`	Total	Latency	Output/Cycle	Min. period	Output/Time
#1	3	0	2	2	7	$2s + \delta_{\texttt{mul}} + \delta_{\texttt{add}}$	1/3	$s = \delta_{\texttt{mul}} + \delta_{\texttt{add}}$	$1/(3(\delta_{\texttt{mul}} + \delta_{\texttt{add}}))$
#2	1	0	1	1	3	$2s + \delta_{\texttt{mul}} + \delta_{\texttt{add}}$	1/3	$s = \delta_{\texttt{mul}} + \delta_{\texttt{add}}$	$1/(3(\delta_{\texttt{mul}} + \delta_{\texttt{add}}))$
#3	3	0	2	0	5	$\delta_{\texttt{mul}} + 2\delta_{\texttt{add}}$	1	$s = \delta_{\texttt{mul}} + 2\delta_{\texttt{add}}$	$1/(\delta_{\texttt{mul}} + 2\delta_{\texttt{add}})$
#4	0	1	1	1	3	$4s + \delta_{\texttt{seq}} + \delta_{\texttt{add}}$	1/5	$s = \delta_{\texttt{seq}} + \delta_{\texttt{add}}$	$1/(5(\delta_{\texttt{seq}} + \delta_{\texttt{add}}))$

Tabela 5.2: Different possible cases of `Dotp`

Case	$\delta_{\texttt{add}}$	$\delta_{\texttt{mul}}$	$\delta_{\texttt{seq}}$
A	1	64	8
B	1	8	3

Version	A	B
#1	1/195	1/21
#2	1/195	1/21
#3	1/66	1/8
#4	1/45	1/20

Discussion

As we can see, all the versions have advantages and disadvantages, and choosing the right one depends on the project's constraints and optimization criteria. If we wanted better throughput, we could either choose higher clock frequency and go with architecture #4 or choose parallelism and go with architecture #3. If we wanted low-cost hardware, versions #2 and #4 would be better suited. If optimization parameters consider cost and throughput equally important, version #4 would probably be the better fit by providing both high throughput and low resources to a certain extent. This entire analysis is only possible by the information provided by the type system.

Comparison with Control Dataflow Graphs

Although the `Dotp` example shows how h-calculus helps analysis and DSE, how does it compare to Control Dataflow Graphs (CDFGs)? CDFGs combine control and data flow within the same graph and are easily extracted from source code, especially from imperative descriptions, but it is not easy to verify and analyze. To demonstrate this distinction, let us suppose the Translation step inferred a CDFG equivalent to `Dot#1` (shown in Fig 5.5, where rectangles represent conditional path selectors).

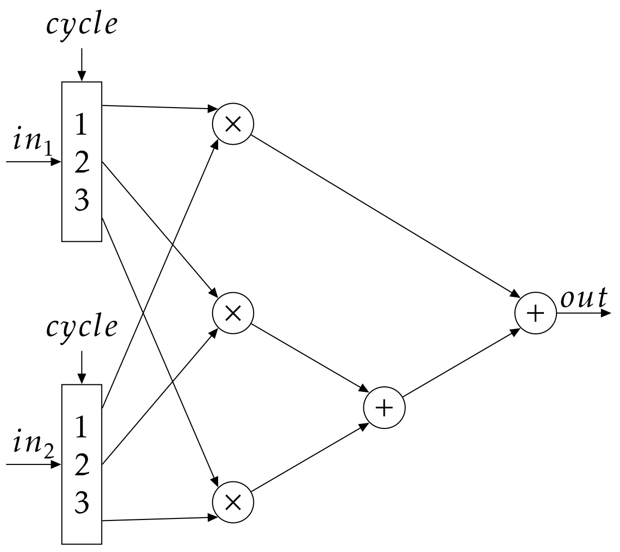


Figura 5.5: `Dotp#1` as a Control Dataflow Graph (CDFG)

Since CFDGs do not model temporal information, *First In First Out (FIFO)* buffers become intrinsic to the design. We do not know during static analysis where buffers need to be synthesized and how much data they need to store at a given time. Several other dataflow models (such as SADF [15, 16], BPDF [14], SPDF [17], VRDF [18]) try to make the buffer analysis problem statically decidable, but most solutions involve exhaustive algorithms. The h-calculus' type system does not natively use buffers, and if one needs to be implemented as a process, it needs to have a finite storage capacity.

Another issue with the dataflow representation is that it is not trivial to analyze when control is involved. For instance, we can perform a resource sharing analysis analogous to using the merge operation in h-calculus (such as the one that generated `Dotp#2` from `Dotp#1`) using dataflow representations; however, the technique would be iterative, verifying every possible path provided by the selector separately. The h-calculus analysis is easy to automate for any other example. In contrast, the kind of graph analysis required by dataflow models does not scale when applied to larger systems.

In summary, compared to dataflow models, analyzing h-calculus descriptions is more computationally straightforward since structured analytical information is readily available within well-typed processes.

HLS systems using h-calculus as IR can perform design exploration more effectively at the cost of a slightly more complex translation stage. Compared to dataflow, the synthesis from IR to RTL should be more straightforward because the h-calculus represents hardware at a lower level (registers, combinational circuits, clock). In contrast, dataflow models still need to infer components (especially buffers), generating a mismatch between static analysis and the actual results.

5.3 Synthesis

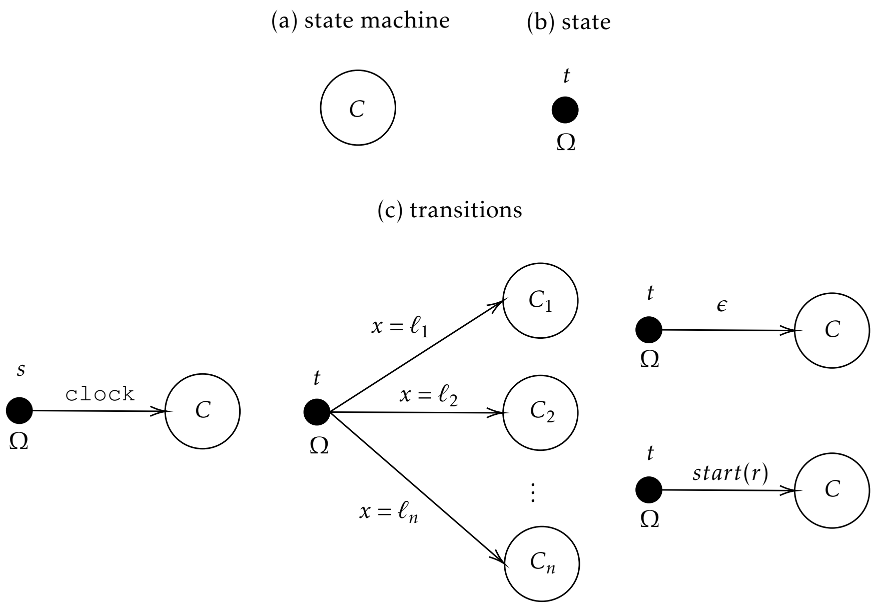
Since h-calculus' level of detail is very close to RTL, the synthesis step — inferring an RTL architecture from an h-calculus specification — is not a complex procedure.

Similar to the RTL model, the h-calculus describes an architecture composed of connections and resources. Unlike RTL, however, h-calculus hides control information and multiplexers (components that route values through different wires depending on a control signal).

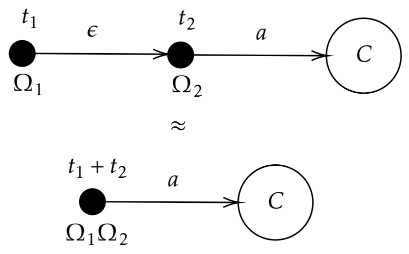
The objective of Synthesis is then to infer all of the things that are implicit in the h-calculus and construct the RTL architecture. Next, we define an RTL architecture as two separate connected parts: the control part, a Finite State Machine that outputs control signals, and the operative part, containing the components, registers, multiplexers, and connections. Then we define *Control Merging* and, finally, *Synthesis*.

5.3.1 Control - Finite State Machines

Definition 32 (Control Part) The control part of an architecture is a finite state machine (a), represented by a circle, where states (b) are assigned a set of control signals, denoted by Ω, and an index t between 0 and s that describes when, within one cycle, the state is situated. Signals in Ω can be: control flow signals $conn(x) = y$, start signals $start(r)$, or branching signals $x = \ell$. The transitions (c) react to a start, clock, branching, or *empty* events.

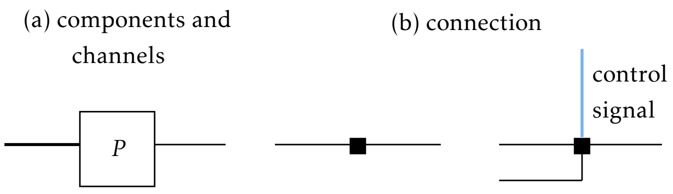


We use the empty event to construct finite state machines recursively. *Control* *equivalence* optimizes away the empty events by considering two states connected by an empty event the same.

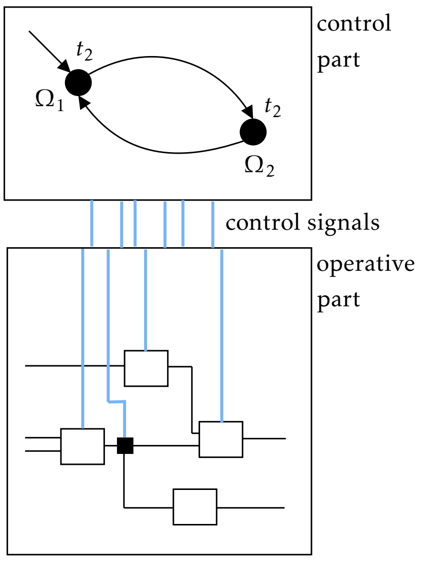


Where $\Omega_1\Omega_2$ is the *union* between all control signals in Ω_1 and Ω_2.

Definition 33 (Operative Part) The operative part of an architecture is a network of components connected through channels. We represent components as rectangles and channels as lines (a). A little dark square (b) graphically represents connections between two endpoints. When more than two endpoints connect, the connection must receive a control signal that will route data correctly — this connection may become a multiplexer at later stages of synthesis depending on the direction of the data.

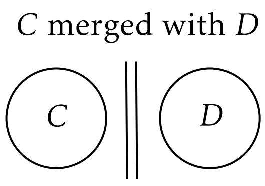


Definition 34 (Hardware Architecture) Hardware architecture represents an RTL description. It is composed of an *operative* and a *control* parts. The control part sends signals to the operative part managing the dynamic connections, and the operative part sends branching information (coming from external choices) as input to the control part.

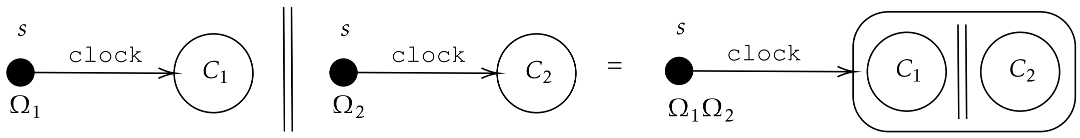


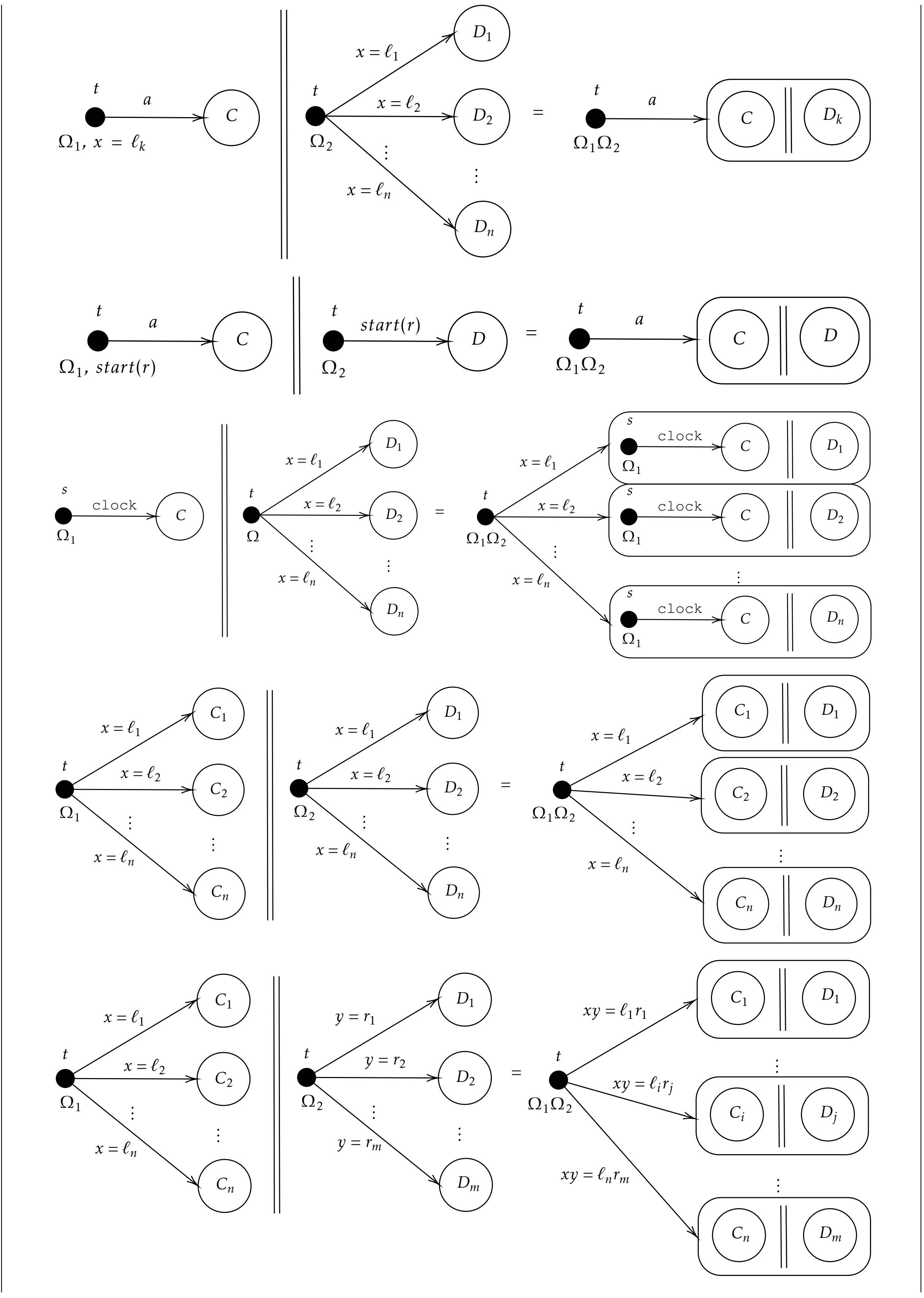
A vital definition for Synthesis is Control Merge, which optimizes two control parts running in parallel, resulting in an equivalent one. Control merge will allow the entire system to have only one control part, made from all its components' finite state machines merged.

Definition 35 (Control Merge) The control merge operation merges two control parts running in parallel and constructs one resulting control part. Its definition is recursive and graphical:

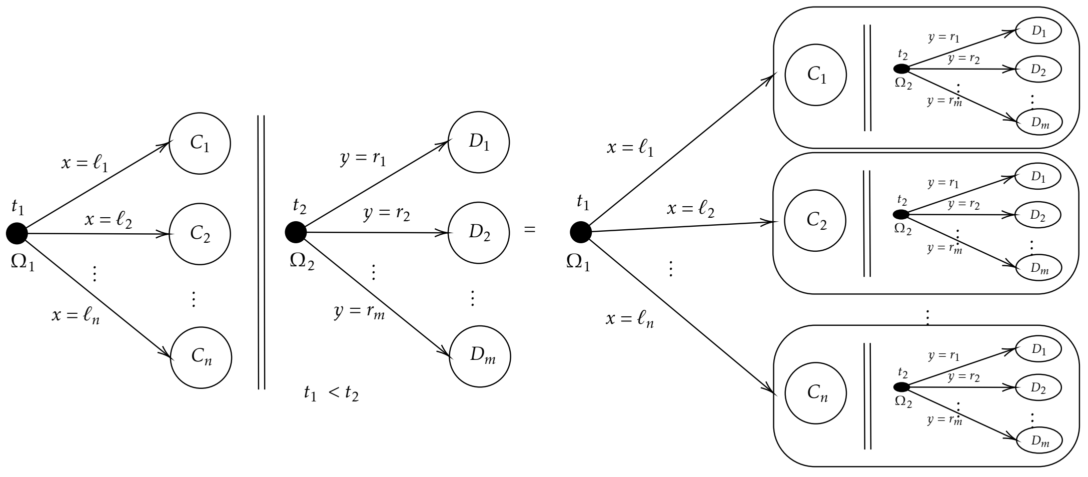


Cases:



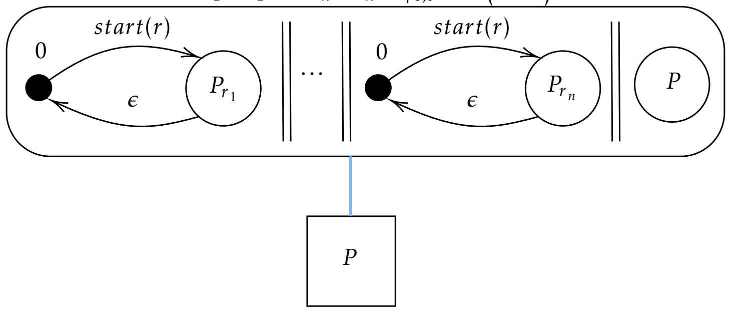


t
a
C
Ω₁, x = ℓ_k
x = ℓ₁
x = ℓ₂
x = ℓ_n
D₁
D₂
D_n
Ω₂
=
Ω₁Ω₂
D_k
start(r)
Ω₁, start(r)
D
s
clock
Ω₁
Ω
C₁
C₂
C_n
y = r₁
y = r₂
y = r_m
D_m
xy = ℓ₁r₁
xy = ℓ_ir_j
xy = ℓ_nr_m
C_i
D_j



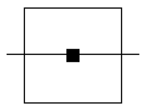
Definition 36 (Synthesis) Synthesis transforms a process into a hardware architecture composed of control and operative (components) parts. Synthesis is defined recursively and graphically:
We start by synthesizing and merging the control parts from all resources within the system together with the main process control:

$$r_1 : R_1, \cdots, r_n : R_n; \Delta \vdash^{0,0}_{s,c} P :: (x : A)$$

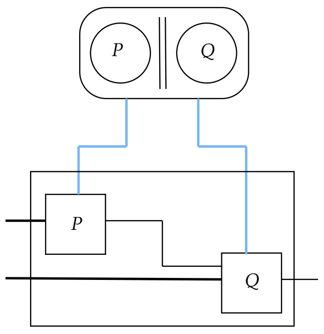


Next rules show how P is transformed:

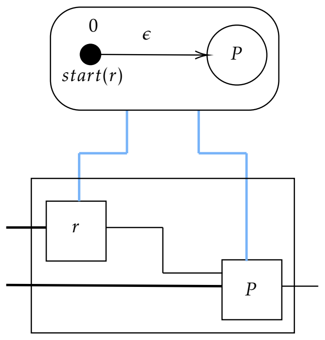
$x \leftarrow y$

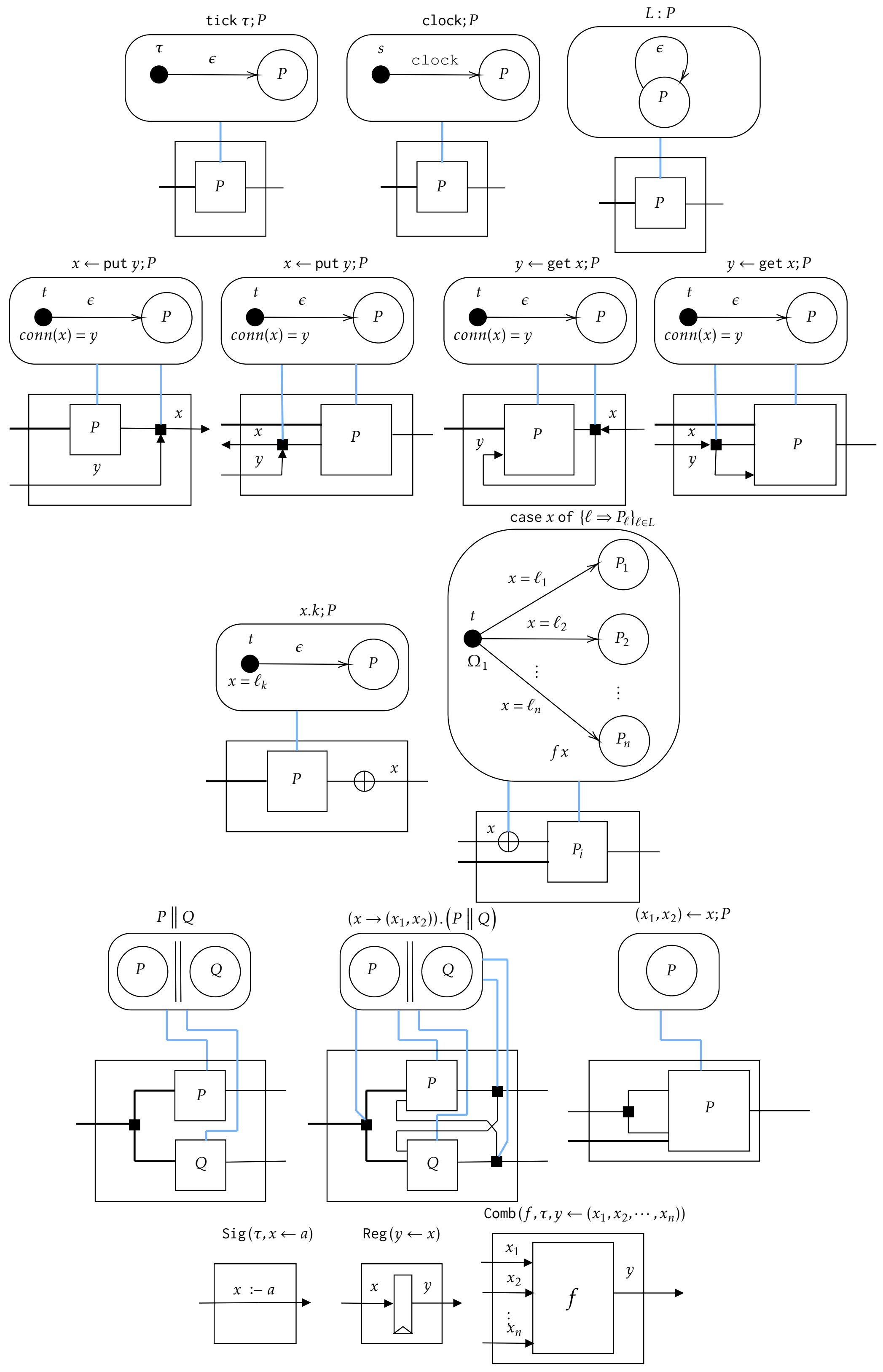


$x \leftarrow P; Q$



$x \leftarrow r \leftarrow \{\Sigma; \Delta\}; Q$





tick τ; P
clock; P
L : P
x ← put y; P
x ← put y; P
y ← get x; P
y ← get x; P
conn(x) = y
case x of {ℓ ⇒ P_ℓ}_{ℓ∈L}
x.k; P
x = ℓ_k
x = ℓ_1
x = ℓ_2
x = ℓ_n
Ω_1
f x
P ∥ Q
(x → (x_1, x_2)).(P ∥ Q)
(x_1, x_2) ← x; P
Sig(τ, x ← a)
x := a
Reg(y ← x)
Comb(f, τ, y ← (x_1, x_2, ⋯, x_n))
f

5.3.2 Practicalities

The control part, hidden by the h-calculus, introduces practical issues that need to be solved for synthesis to produce correct hardware.

The synthesis step also includes transforming the control part into a circuit. It translates the Finite State Machine first into a set of boolean equations, then into a digital circuit composed of flip-flops and logic gates (e.g., `AND`, `OR`, `NOT`, and `NAND`). Like any hardware component, control needs time to transition from one state to the other. We denote $\delta_{\texttt{control}}$ as the maximum amount of time elapsed for an input change to result in a stable state (maximum transition time).

Setup Time We explained earlier, in Chapter 2, that the *setup time* $st = c - s$ is the time that it takes for every register within the system to stabilize its value. It turns out st must also be greater or equal than $\delta_{\texttt{control}}$, or else the state would not be stable during the stable period, resulting in the following additional constraint: $st \geq \texttt{max}(\delta_{\texttt{register}}\delta_{\texttt{control}})$.

Synthesizable Choice Another consequence of having to consider $\delta_{\texttt{control}}$ regards the choice events. Different from reacting to clock events, reacting to choice events of type $\overrightarrow{\oplus}_x\{\ell : S_\ell\}_{\ell\in L}$ can occur anywhere within the stable period. Since the maximum transition time is at least greater than zero, an additional constraint is that types such as $\overrightarrow{\oplus}_x\{\ell : S_\ell\}_{\ell\in L}$ need to be replaced by their delayed versions $\overrightarrow{\oplus}_x\{\ell : \bullet^{\delta_{\texttt{control}}} S_\ell\}_{\ell\in L}$ to synthesize correctly.

Fortunately, this constraint needs to be enforced only to choices produced **outside** of the system since those produced within the system have their control optimized away by the *control merge* operation (Def. 35). For a choice external to the system to be synthesizable, it needs to follow the pattern $\overrightarrow{\oplus}_x\{\ell : \bullet^\tau S_\ell\}_{\ell\in L}$ with $\tau \geq \delta_{\texttt{control}}$. For example, if c is a channel coming from the external environment, type

$$c : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \overrightarrow{Int}^s L, \\ neg : \overrightarrow{Int}^s L, \\ fwd : \overrightarrow{Int}^s L \end{cases}$$

is incorrect, while type

$$c : \mu L.\overrightarrow{\oplus}_c \begin{cases} inc : \bullet^\tau \overrightarrow{Int}^{s-\tau} L, \\ neg : \bullet^\tau \overrightarrow{Int}^{s-\tau} L, \\ fwd : \bullet^\tau \overrightarrow{Int}^{s-\tau} L \end{cases}$$

with $\tau \geq \delta_{\texttt{control}}$ is correct.

Capítulo 6

Related Work

This chapter will compare the h-calculus to other models of computation (MoCs) used for hardware modeling or concurrent systems modeling. To aid the comparison, we provide a summary of the comparisons in Table 6.1. The table analyses all models of computation according to the following criteria:

Well-Formedness Does the model allow for any well-formedness technique (such as type-checking or iterative model checking) that detects erratic hardware?

Analysis Does the MoC provide methods for analysis that are not computationally intensive? Where analysis, in this case, is the ability to retrieve, from the model, efficiency parameters that will guide design exploration.

Expressive Is the MoC expressive enough to model complex computations, or do their limitations make it difficult for designers to represent complex architectures?

Resource Does the model understand resource usage and resource sharing? Efficient use and sharing of resources are perhaps one of the most critical aspects of hardware design optimization.

A model without resource modeling cannot validate or analyze the way resources are used, which leads to inefficient results. An approach to avoid this problem is considering every instance of processes to be unique. Although this is acceptable for deeply parallel architectures, this approach cannot output "lean"results with fewer components, which is undesirable.

Concurrency Is it a concurrent model of computation? Since hardware is a naturally concurrent system, using a model that does not understand concurrency implies the HLS system must **infer** concurrency before trying to optimize any design.

Because inferring concurrency is an undecidable, complex problem with unsatisfactory solutions [40, 41], non-concurrent models should be avoided for hardware modeling.

Time Does the MoC model time? Although many models represent sequences of events, such as *action A then action B*, and some understand the notion of cycles, few MoCs model time inside one clock cycle. Modeling time with more precision allows the model to analyze and transform architectures more efficiently [40].

HLL → X (Translation) Is it easy to convert High-Level Language (HLL) into the target model?

X → RTL (Synthesis) Is it easy to transform the model definition into an RTL or other low-level hardware representation?

Hardware Does the MoC understand hardware design concepts such as clock, registers, combinational circuits, and others?

6.1 Comparisons

Next, we discuss the most relevant aspects of all models of computation presented in Table 6.1.

(HC) H-Calculus The h-calculus uses session type checking/inference to detect ill-formed hardware descriptions.

Analysis is straightforward because efficiency parameters can be fetched from the type definitions of processes, including resource usage and communication information. The h-calculus understands time, concurrency, and hardware concepts, and it is expressive enough for hardware-design needs, although not as expressive as general MoCs such as λ or π-calculus.

Although it is simple to transform the h-calculus into HLS (Def. 36), it is not as trivial to convert HLLs into h-calculus when compared to other models. The h-calculus trades off ease to define for ease to analyze. The separation of HLS in steps — that removes any responsibility of optimization from the translation step — alleviates this downside, but the transformation is still relatively challenging.

(DF) Dataflow Dataflow [42, 43, 44, 45, 46, 47] is a directed graph where nodes are concurrent actors, and the vertices are communication channels. Actors perform computation when certain **trigger** conditions on their inputs are met. Although it is an expressive concurrent model of computation, it is generally hard to analyze.

In order to suit better the needs of specific applications, including better analysis, different kinds dataflow models were introduced. Different trigger conditions define different kinds of Dataflow. This chapter will briefly discuss *Control Dataflow Graphs* (CDFGs) [19], *Boolean Parametric Dataflow* (BPDF) [14], *Scenario-Aware Dataflow* (SADF) [15, 16], *Schedulable Parametric Dataflow* (SPDF) [17], and *Variable-Rate Dataflow* (VRDF) [18]. Figure 6.1 shows a map of different dataflow models and their relationship to each other.

$$\begin{tikzcd}
 & & \text{DF/KN} \arrow[dl] \arrow[dr] & & \\
 & \text{CDFG} & & \text{SDF} \arrow[d] & \\
 & & & \text{ESDF} \arrow[dll] \arrow[dr] \arrow[ddl] \arrow[dd] & \\
 & \text{BPDF} & & & \text{VRDF} \\
 & & \text{SADF} & \text{SPDF} &
\end{tikzcd}$$

Figura 6.1: Dataflow Relationship Schematics

(KN) Kahn Networks A Kahn Network [48] is a network of concurrent components represented as mathematical functions that manipulate sequences of values instead of plain values. [11] shows that Kahn Networks and Dataflow are equivalent. Similar to DF, KN is expressive but hard to analyze.

(CDFG) Control Dataflow Graphs Control Dataflow Graphs [19] are the most commonly used intermediate representation in HLS systems [2, 3, 1]. CDFGs are a kind of Dataflow representing both the control and operative parts of the hardware within the same graph. In other words, the actors' computations can change state variables, thus controlling the operative part.

The advantage of CDFGs is that it is straightforward to infer it from high-level code, especially sequential code. Thus it is commonly used to extract control information from C programs (the most common example). However, CDFGs do not solve the general Dataflow problem of being difficult to analyze.

Since analyzing CDFGs is an arduous task, HLS systems based on CDFGs rarely rely upon gathering information from definitions [4, 9]. Instead, they apply a fixed, statically defined sequence of optimizations/algorithms that statistically gives better results most of the time. While an inflexible is not efficient enough for general hardware designs, it often works for specific applications [2, 3, 1].

(SDF) Synchronous Dataflow Synchronous Dataflow [49] (see Figure 6.1) is a dataflow with triggering rules that require actors to consume and produce values at **fixed**, statically determined rates (for example, two values per cycle).

SDF's restrictive firing rules result in a loss of expressiveness but allow for better analysis compared to general DF and CDGF models. It is straightforward to check deadlock freedom, optimally schedule tasks, and choose buffer sizes. If an application needs little to no control flow, SDF can be a powerful model to use.

It is important to note that SDF's analysis have several differences from those performed within h-calculus. H-Calculus performs analysis during construction through the type checking/inference algorithm, while algorithms for analysis used in SDF and other dataflows rely on graph crawling. Graph crawling gets computationally expensive as systems get big, which is the case with modern *System on Chips* (SoCs).

Moreover, h-calculus' analysis relates almost 1 to 1 to hardware analysis, while SDF's analysis needs to be interpreted as hardware analysis, resulting in inaccuracies.

The major downside of SDF is that it is not very expressive. For example, since it allows fixed rates only, any conditional or dynamic flow cannot be represented. Several kinds of Dataflow try to combine the SDF's ease of analysis with the expressiveness of dynamic DF. We are going to talk about some of them next.

(ESDF) Extended Synchronous Dataflow Extended SDF is the name we use to classify Dataflows that try to extend the SDF model, adding more expressiveness while keeping the ease of analysis (see Fig 6.1).

Some examples of ESDF are *Boolean Parametric Dataflow* (BPDF) [14], *Scenario-Aware Dataflow* (SADF) [15, 16], *Schedulable Parametric Dataflow* (SPDF) [17], and *Variable-Rate Dataflow* (VRDF) [18].

Details aside, these Dataflow models attempt to extend the SDF with mechanisms that change the rate of consumption/production of values dynamically. The result is a slightly more complicated model that keeps SDF's properties (e.g., deadlock freedom, buffer size decidability) but is better suited for applications with more control flow.

Although these models look promising for hardware modeling, especially compared to the standard CDFGs used in HLS, their major drawback is their lack of time representation. It should be simple to declare actors related to hardware components (such as registers, multiplexers, and logical gates), but extending Dataflow models with temporal information while keeping it analyzable is not trivial.

(FSMD) Finite State Machine with Dataflow The FSMD model [4, 50] combines a finite state machine representing control with a hardware architecture datapath. Compared to other models, this one understands hardware concepts such as control signals, clock cycle, registers, and multiplexers, so it is trivial to transform a description into an RTL one. Furthermore, it understands resource usage and resource sharing.

Although the FSMD represents hardware somewhat accurately, it does not have adequate analysis mechanisms. Since FSMD is more low-level and harder to manipulate, it is common for current HLS systems to transform CDFGs into FSMD during the later stages of high-level synthesis [4, 2, 9, 3, 1].

(LC) Lambda Calculi The untyped lambda calculus (seen in Chapter 3.4) is a *Turing complete* model of computation with simple syntax and semantic rules. Its simplicity and expressiveness are the reason several programming languages are based on it. However, λ-calculus' abstractions and applications are not enough for hardware modeling as they cannot trivially model concurrency, resources, time, and hardware components.

(TLC) Typed Lambda Calculi While types still do not trivially model concurrency and time, using a type system brings ease of well-formedness, which is a crucial part of hardware design.

Expressive types allow hardware-like types to be constructed — such as `Signal`, `Component` types, for example —, which is the approach taken by some research [51, 52, 53, 54, 55]. The problem with this approach is that while it might be a better *Hardware Description Language* with well-formedness advantages, it is still difficult to analyze.

(PC) Process Calculi. The process calculus approach models concurrent processes with algebraic simplicity compared to lambda-calculus. The pi-calculus [56, 57] and its variations stand out for being Turing-complete (able to encode the lambda-calculus within themselves) among all process calculi. The pi-calculus, for instance, models channels and concurrent processes and uses operational semantic rules to describe how the processes evolve through interactions and time.

The pi-calculus is more suitable for hardware design than lambda-calculi since it is built around concurrency, but it is still not a perfect fit. Several modeling details separate pi-calculus from an ideal hardware representation, the most prevalent one being the lack of efficient analysis mechanisms.

(STPC) Session-Typed Process Calculi. Session Types [23, 21, 20] solve the well-formedness issue of process calculi by providing a type system that understands concurrency and ensures well-formation — no deadlock or communication errors possible for well-typed processes for example. Compared to Dataflow models, session types provide a great balance between efficient well-formedness checking and expressiveness.

Although session types look promising candidates for hardware modeling, it still is not the perfect fit. Considering basic session types described in Section 3.4 [23], the system lacks temporal durations and is not hardware-aware enough to provide high-quality hardware analysis. Furthermore, some of the expressiveness needs to be capped for hardware design's sake. For example, *dynamic creation of channels*, a distinct characteristic of the π-calculus, does not translate well into (efficient) hardware as well as some linear-logic inspired interpretations.

The upside about Session Types is that it is possible to extend/modify the type system to make it as appropriate for the target application as possible (e.g., [58, 59, 24, 32, 33, 60, 61, 62, 25, 26]), which was the path taken by the h-calculus research.

X	Well-Formedness	Analysis	Expressive	Resource	Concurrency	Time	HLL → X	X → RTL	Hardware
HC	✓	✓	✓	✓	✓	✓	✗	✓	✓
DF	✗	✗	✓	✗	✓	✗	✓	✗	✗
KN	✗	✗	✓	✗	✓	✗	✓	✗	✗
CDFG	✗	✗	✓	✗	✓	✗	✓	✗	✗
SDF	✓	✗	✗	✗	✓	✗	✓	✗	✗
ESDF	✓	✗	✓	✗	✓	✗	✓	✗	✗
FSMD	✗	✗	✗	✓	✓	✗	✗	✓	✓
LC	✗	✗	✓	✗	✗	✗	✓	✗	✗
TLC	✓	✗	✓	✗	✗	✗	✓	✗	✗
PC	✗	✗	✓	✗	✓	✗*	✗	✗	✗
STPC	✓	✓	✓	✗*	✓	✗	✗	✗	✗

Tabela 6.1: Comparison of different models of computation

Capítulo 7

Discussion and Future Work

,

In this thesis, we introduced a process calculus for hardware design called *H-Calculus*, which uses a novel type system of *Temporal Session Types (TSTs)*, based on the works on *Session Types*. The H-Calculus provides an intermediate representation well-suited for *Design Space Exploration (DSE)* of hardware by focusing on correctness-by-construction and ease of analysis. We provided several examples demonstrating the level of expressiveness and detail of the h-calculus to describe, analyze and transform low-level interconnected hardware modules.

However, there are still pending issues with the h-calculus and research to be done. Translation from high-level languages needs to be solved entirely with an efficient algorithm and a functionality conservation proof. Furthermore, an easier way to prove functionality preservation for transformations/optimizations would be handy. [37] partially solves translation's challenges for functional languages, but it is incomplete how to fit all of the details within the context of the h-calculus.

The h-calculus' type system is expressive enough to model low-level hardware, but it comes with a price. The type-checking algorithm for the h-calculus is somewhat complex and needs a great deal of type inference to work. A decidable type inference algorithm that implements the h-calculus' type system has not yet been thought out. This algorithm should be a priority for future work since a future implementation requires it.

Future work is directed towards implementation of an entire HLS flow, including a translation scheme (the frontend of a compiler), a Design Space Exploration system, and a synthesis scheme into RTL (backend of the compiler), and assess the efficiency of the system for real-world hardware design examples.

Further research on the properties of the h-calculus type system could have exciting consequences for hardware design. For example, the possibility of type inhabitation could imply hardware generation from their types alone.

One of our main motivations for introducing this calculus was to offer a powerful intermediate representation for High-Level Synthesis, making automatic hardware DSE practical. Although the thesis does not describe a particular DSE scheme, which could be the theme for future research, the h-calculus is the perfect environment to implement such a complex scheme, which depends on extensive detailed analysis and transformations.

The implementation of other parts of the HLS system, including a compiler frontend and backend and a hardware DSE scheme, seem to be the next logical steps for future work. Also, a study on the description of real-world hardware design examples using h-calculus would be useful to better under and its advantages, disadvantages, and limitations.

References

[1] Campbell, Keith, Wei Zuo, and Deming Chen: *New advances of high-level synthesis for efficient and reliable hardware design*. Integration, 58:189–214, 2017, ISSN 0167-9260. https://www.sciencedirect.com/science/article/pii/S0167926016301432. 1, 2, 3, 40, 93, 94, 95

[2] Nane, R., V. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels: *A Survey and Evaluation of FPGA High-Level Synthesis Tools*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 35(10):1591–1604, 2016. 1, 2, 3, 40, 93, 94, 95

[3] Meeus, Wim, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt: *An overview of today's high-level synthesis tools*. Design Autom. for Emb. Sys., 16:31–51, September 2012. 1, 2, 3, 40, 93, 94, 95

[4] Coussy, P., D. D. Gajski, M. Meredith, and A. Takach: *An Introduction to High-Level Synthesis*. IEEE Design Test of Computers, 26(4):8–17, 2009. 1, 2, 3, 40, 93, 95

[5] Martin, G. and G. Smith: *High-Level Synthesis: Past, Present, and Future*. IEEE Design Test of Computers, 26(4):18–25, 2009. 2, 3, 40

[6] Canis, Andrew, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson: *LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems*. ACM Trans. Embed. Comput. Syst., 13(2), September 2013, ISSN 1539-9087. https://doi.org/10.1145/2514740. 2, 40

[7] Bourgeat, Thomas, Clément Pit-Claudel, Adam Chlipala, and Arvind: *The Essence of Bluespec: A Core Language for Rule-Based Hardware Design*. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 243–257, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450376136. https://doi.org/10.1145/3385412.3385965. 2, 40

[8] Nikhil, Rishiyur S.: *Bluespec: A General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions*, pages 129–146. Springer Netherlands, Dordrecht, 2008, ISBN 978-1-4020-8588-8. https://doi.org/10.1007/978-1-4020-8588-8_8. 2, 40

[9] Fingeroff, Michael: *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010, ISBN 1450097243. 2, 40, 93, 95

[10] Fine Licht, Johannes de, Maciej Besta, Simon Meierhans, and Torsten Hoefler: *Transformations of High-Level Synthesis Codes for High-Performance Computing*. CoRR, abs/1805.08288, 2018. `http://arxiv.org/abs/1805.08288`. 2, 40

[11] Lee, Edward A and Eleftherios Matsikoudis: *The semantics of dataflow with firing*. G. Huet, G. Plotkin, J.-J. Lévy, and Y. Bertot, editors, From Semantics to Computer Science: Essays in Honour of Gilles Kahn, pages 71–94, 2009. 9, 40, 93

[12] Girard, Jean Yves: *The system F of variable types, fifteen years later*. Theoretical computer science, 45:159–192, 1986. 40

[13] Aspinall, David and Martin Hofmann: *Dependent Types*, pages 45–86. MIT Press, December 2004, ISBN 9780262162289. 40

[14] Bebelis, Vagelis, Pascal Fradet, Alain Girault, and Bruno Lavigueur: *BPDF: A statically analyzable dataflow model with integer and boolean parameters*. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2013. 41, 82, 93, 94

[15] Stuijk, Sander, Marc Geilen, Bart Theelen, and Twan Basten: *Scenario-Aware Dataflow: Modeling, Analysis and Implementation of Dynamic Applications*. pages 404 – 411, August 2011. 41, 82, 93, 94

[16] Theelen, Bart, M.C.W. Geilen, T. Basten, Jeroen Voeten, S.V. Gheorghita, and Sander Stuijk: *A Scenario-Aware Data Flow model for combined long-run average and worst-case performance analysis*. pages 185 – 194, August 2006. 41, 82, 93, 94

[17] Fradet, Pascal, Alain Girault, and Peter Poplavko: *SPDF: A Schedulable Parametric Data-Flow MoC (Extended Version)*. Research Report RR-7828, INRIA, December 2011. `https://hal.inria.fr/hal-00666284`. 41, 82, 93, 94

[18] Wiggers, Maarten H., Marco J.G. Bekooij, and Gerard J.M. Smit: *Buffer Capacity Computation for Throughput Constrained Streaming Applications with Data-Dependent Inter-Task Communication*. In *2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 183–194, 2008. 41, 82, 93, 94

[19] Orailoglu, Alex and Daniel Gajski: *Flow Graph Representation*. pages 503–509, January 1986. 41, 93

[20] Honda, Kohei: *Types for dyadic interaction*. In *International Conference on Concurrency Theory*, pages 509–523. Springer, 1993. 41, 96

[21] Kobayashi, Naoki, Benjamin Pierce, and David Turner: *Linearity and the Pi-Calculus*. ACM Transactions on Programming Languages and Systems (TOPLAS), 21:914–947, December 1999. 41, 96

[22] Abramsky, Samson: *Computational interpretations of linear logic*. Theoretical Computer Science, 111(1):3–57, 1993, ISSN 0304-3975. https://www.sciencedirect.com/science/article/pii/030439759390181R. 41

[23] Caires, Luís and Frank Pfenning: *Session Types as Intuitionistic Linear Propositions*. In Gastin, Paul and François Laroussinie (editors): *CONCUR 2010 - Concurrency Theory*, pages 222–236, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg, ISBN 978-3-642-15375-4. 41, 58, 96

[24] Das, Ankush, Jan Hoffmann, and Frank Pfenning: *Parallel Complexity Analysis with Temporal Session Types*. Proc. ACM Program. Lang., 2(ICFP), July 2018. https://doi.org/10.1145/3236786. 42, 43, 96

[25] Das, Ankush, Jan Hoffmann, and Frank Pfenning: *Work Analysis with Resource-Aware Session Types*. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 305–314, New York, NY, USA, 2018. Association for Computing Machinery, ISBN 9781450355834. https://doi.org/10.1145/3209108.3209146. 42, 96

[26] Balzer, Stephanie, Frank Pfenning, and Bernardo Toninho: *A universal session type for untyped asynchronous communication*. In *29th International Conference on Concurrency Theory (CONCUR 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. 42, 96

[27] Yi, Wang: *CCS + time = an interleaving model for real time systems*. In Albert, Javier Leach, Burkhard Monien, and Mario Rodríguez Artalejo (editors): *Automata, Languages and Programming*, pages 217–228, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg, ISBN 978-3-540-47516-3. 43

[28] *Timed Process Calculi, a LOTOS Perspective*, pages 261–286. Springer London, London, 2006, ISBN 978-1-84628-336-9. https://doi.org/10.1007/1-84628-336-1_9. 43

[29] Bernardo, Marco, Flavio Corradini, and Luca Tesei: *Timed process calculi with deterministic or stochastic delays: Commuting between durational and durationless actions*. Theoretical Computer Science, 629:2–39, 2016, ISSN 0304-3975. https://www.sciencedirect.com/science/article/pii/S0304397516001444, Theoretical Computer Science in Italy. 43

[30] Cerans, Karlis, Jens Godskesen, and Kim Larsen: *Timed Modal Specification - Theory and Tools*. Volume 697, pages 253–267, June 1993, ISBN 978-3-540-56922-0. 43

[31] Moller, Faron and Chris Tofts: *A temporal calculus of communicating systems*. In Baeten, J. C. M. and J. W. Klop (editors): *CONCUR '90 Theories of Concurrency: Unification and Extension*, pages 401–415, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg, ISBN 978-3-540-46395-5. 43

[32] Bocchi, Laura, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida: *Asynchronous Timed Session Types*. In Caires, Luís (editor): *Programming Languages and Systems*, pages 583–610, Cham, 2019. Springer International Publishing, ISBN 978-3-030-17184-1. 43, 96

[33] Bartoletti, Massimo, Tiziana Cimoli, and Maurizio Murgia: *Timed Session Types*. arXiv e-prints, page arXiv:1710.05388, October 2017. 43, 96

[34] Halbwachs, N., P. Caspi, P. Raymond, and D. Pilaud: *The synchronous data flow programming language LUSTRE*. Proceedings of the IEEE, 79(9):1305–1320, 1991. 43

[35] Caspi, P., D. Pilaud, N. Halbwachs, and J. A. Plaice: *LUSTRE: A Declarative Language for Real-Time Programming*. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, page 178–188, New York, NY, USA, 1987. Association for Computing Machinery, ISBN 0897912152. https://doi.org/10.1145/41625.41641. 43

[36] Durgin, Nancy, Patrick Lincoln, and John Mitchell: *Multiset Rewriting and the Complexity of Bounded Security Protocols*. Journal of Computer Security, 12:247–311, February 2004. 46

[37] Sá, Luiz: *Síntese de arquiteturas dedicadas a partir de linguagens funcionais*. Bachelor's Thesis, Universidade de Brasília, DF, Brazil, 2017. https://bdm.unb.br/handle/10483/19378. 75, 98

[38] Townsend, Richard, Martha A. Kim, and Stephen A. Edwards: *From Functional Programs to Pipelined Dataflow Circuits*. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 76–86, New York, NY, USA, 2017. Association for Computing Machinery, ISBN 9781450352338. https://doi.org/10.1145/3033019.3033027. 75

[39] Zhai, Kuangya, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards: *Hardware synthesis from a recursive functional language*. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 83–93, 2015. 75

[40] Edwards, S.A.: *The Challenges of Synthesizing Hardware from C-Like Languages*. IEEE Design Test of Computers, 23(5):375–386, 2006. 92

[41] Masud, Abu, Björn Lisper, and Federico Ciccozzi: *Automatic Inference of Task Parallelism in Task-Graph-Based Actor Models*. IEEE Access, PP:1–1, December 2018. 92

[42] Whiting, P.G. and R.S.V. Pascoe: *A history of data-flow languages*. IEEE Annals of the History of Computing, 16(4):38–59, 1994. 92

[43] Ackerman, W.: *Data flow languages*. 1979 International Workshop on Managing Requirements Knowledge (MARK), pages 1087–1095, 1979. 92

[44] Adams, Duane Albert: *A Computation Model with Data Flow Sequencing*. PhD thesis, Stanford, CA, USA, 1969. AAI6913919. 92

[45] Karp, Richard M and Rayamond E Miller: *Properties of a model for parallel computations: Determinacy, termination, queueing*. SIAM Journal on Applied Mathematics, 14(6):1390–1411, 1966. 92

[46] Lee, Edward A. and Thomas M. Parks: *Dataflow Process Networks*, page 59–85. Kluwer Academic Publishers, USA, 2001, ISBN 1558607021. 92

[47] Rodrigues, J. E. and Jorge E Rodriguez Bezos: *A Graph Model for Parallel Computations*. Technical report, USA, 1969. 92

[48] Kahn, Gilles: *The Semantics of a Simple Language for Parallel Programming*. In Rosenfeld, Jack L. (editor): *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 471–475. North-Holland, 1974. 93

[49] Lee, Edward A and David G Messerschmitt: *Synchronous data flow*. Proceedings of the IEEE, 75(9):1235–1245, 1987. 94

[50] Davis, Justin and Robert Reese: *Finite State Machine Datapath Design, Optimization, and Implementation*. Synthesis Lectures on Digital Circuits and Systems, 2(1):1–113, 2007. 95

[51] Grundy, J., T. Melham, and J. O'Leary: *A reflective functional language for hardware design and theorem proving*. Journal of Functional Programming, 16:157 – 196, 2005. 95

[52] Brady, Edwin, James McKinna, and Kevin Hammond: *Constructing Correct Circuits: Verification of Functional Aspects of Hardware Specifications with Dependent Types*. In *Proceedings of the Eighth Symposium on Trends in Functional Programming, TFP 2007, New York City, New York, USA, April 2-4. 2007*, pages 159–176, 2007. 95

[53] Baaij, Christiaan, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards: *CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell*. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, pages 714–721, 2010. 95

[54] Bjesse, Per, Koen Claessen, Mary Sheeran, and Satnam Singh: *Lava: Hardware Design in Haskell*. ACM SIGPLAN Notices, 34, May 2001. 95

[55] Grov, Gudmund, Andrew Ireland, Greg Michaelson, and Kevin Hammond: *Verifying Temporal Properties in HW-Hume*. January 2006. 95

[56] Oquendo, Flavio: *-ARL: An Architecture Refinement Language for Formally Modelling the Stepwise Refinement of Software Architectures*. SIGSOFT Softw. Eng. Notes, 29(5):1–20, September 2004, ISSN 0163-5948. `https://doi.org/10.1145/1022494.1022517`. 95

[57] Milner, Robin: *Communicating and Mobile Systems: The -Calculus*. Cambridge University Press, USA, 1999, ISBN 0521658691. 95

[58] Pruiksma, Klaas and Frank Pfenning: *A Message-Passing Interpretation of Adjoint Logic*. In Martins, Francisco and Dominic Orchard (editors): *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 60–79, 2019. https://doi.org/10.4204/EPTCS.291.6. 96

[59] Balzer, Stephanie, Bernardo Toninho, and Frank Pfenning: *Manifest deadlock-freedom for shared session types*. In *European Symposium on Programming*, pages 611–639. Springer, 2019. 96

[60] Das, Ankush and Frank Pfenning: *Session Types with Arithmetic Refinements and Their Application to Work Analysis*. CoRR, abs/2001.04439, 2020. https://arxiv.org/abs/2001.04439. 96

[61] Honda, Kohei, Nobuko Yoshida, and Marco Carbone: *Multiparty Asynchronous Session Types*. J. ACM, 63(1), March 2016, ISSN 0004-5411. https://doi.org/10.1145/2827695. 96

[62] Das, Ankush and Frank Pfenning: *Rast: A Language for Resource-Aware Session Types*. CoRR, abs/2012.13129, 2020. https://arxiv.org/abs/2012.13129. 96

Apêndice A

Definitions, Theorems and Proofs

A.1 Definitions

Definition 37 (Communicating through channel) $\mathcal{C}$ is communicating through channel c if $\mathsf{proc}\big(r \mid P \mid c\big) \in \mathcal{C}$, for any r, or $\mathsf{proc}\big(r \mid P \mid d\big) \in \mathcal{C}$, for any r and $d \neq c$, where P is equal to:

I. $c \leftarrow \mathtt{put}\ y; P'$

II. $y \leftarrow \mathtt{put}\ c; P'$

III. $y \leftarrow \mathtt{get}\ c; P'$

IV. $c \leftarrow \mathtt{get}\ y; P'$

V. $c.k; P'$

VI. $\mathsf{case}\ c\ \mathsf{of}\ \{\ell \Rightarrow Q_\ell\}_{\ell \in L}$

VII. $(c \rightarrow (c_1, c_2)).\big(P_1 \parallel P_2\big)$

VIII. $P_1 \parallel P_2$

IX. $(c_1, c_2) \leftarrow c; P'$

X. $\mathtt{Sig}(\tau, c \leftarrow a)$

XI. $\mathtt{Reg}(c \leftarrow x)$

XII. $\mathtt{Reg}(y \leftarrow c)$

XIII. $\texttt{Comb}(f,\tau,c \leftarrow (d_1,d_2,\cdots,d_n))$

XIV. $\texttt{Comb}(f,\tau,d \leftarrow (c_1,c_2,\cdots,c_n))$ such that $\exists j \in [1,n].c_j = c$

Definition 38 (Requesting resource) $\mathcal{C}$ is requesting resource r if it has the form $\mathcal{C}',\texttt{proc}\big(p \,\big|\, x \leftarrow r \leftarrow \{\Sigma;\Delta\};Q \,\big|\, d\big)$, for any p, x, Σ, Δ, Q, d.

Definition 39 (Configuration Well-formedness) $\mathcal{C}$ is well formed if:

- $\mathcal{C} = \texttt{Main}(P),\underline{\texttt{env}}(c,s,k,t)$, with $c \geq s \geq t$, or
- $\mathcal{C} = \texttt{Closed}\{\,\mathcal{C}',\underline{\texttt{env}}(c,s,k,t)\,\}$, with $c \geq s \geq t$, where inside $\mathcal{C}'$, for every channel c there is exactly one $\texttt{proc}\big(r \,\big|\, P \,\big|\, c\big)$ object and for every resource r there is exactly one $\texttt{idle}(r)$ or $\texttt{proc}\big(r \,\big|\, P \,\big|\, c\big)$ object.

A.2 Lemmas and Corollaries

Lemma 1 (Right Inversion). *If* $\Sigma^I;\Delta^I \models \mathcal{C},\underline{\texttt{env}}(T_c,T_s,k,t) :: \big(\Sigma^O;\Delta^O,c:S\big)$ *then* $\texttt{proc}\big(r \,\big|\, P \,\big|\, c\big) \in \mathcal{C}$ *where* $\Sigma;\Delta \vdash^{k,t}_{T_s,T_c} P :: \big(c : [S]^{+(kT_s+t)}\big)$.

Demonstração. Inversion on proc and compose ∎

Lemma 2 (Left Inversion). *If* $\Sigma^I;\Delta^I,c:S \models \mathcal{C},\underline{\texttt{env}}(T_c,T_s,k,t) :: \big(\Sigma^O;\Delta^O\big)$ *then* $\big[\texttt{proc}\big(r_i \,\big|\, P_i \,\big|\, x_i\big)\big]_{\forall i\in[1,n]} \in \mathcal{C}$ *where* $\Sigma_i;\Delta_i,c:S_i \vdash^{k,t}_{T_s,T_c} P_i :: \big(x_i : S_i\big)$ *and* $\prod_{i=1}^{n} S_i = [S]^{+(kT_s+t)}$.

Demonstração. Inversions on proc and compose ∎

Lemma 3 (Resource Inversion). *If* $\Sigma^I;\Delta^I \models \mathcal{C},\underline{\texttt{env}}(T_c,T_s,k,t) :: \big(\Sigma^O,r:R;\Delta^O\big)$ *then either*

I. $\texttt{idle}(r) \in \mathcal{C}$, *or*

II. $\texttt{proc}\big(r \,\big|\, P \,\big|\, x\big) \in \mathcal{C}$

Demonstração. Inversions on proc, idle, and compose ∎

Lemma 4 (Comb reduction). *If* $-;\Delta \models \texttt{Closed}\{\,\mathcal{C}\,\} :: (r:R;x:A)$ *with* $\texttt{proc}\big(q \,\big|\, \texttt{Comb}(f,d,z \leftarrow (y_1,\cdots,y_n)) \,\big|\, z\big) \in \mathcal{C}$, *for any* q, f, d, $y_1,\cdots,y_n$ *and* $z \neq x$, *then either*

1. $\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{D}$, *for some* $\mathcal{D}$, *or*

2. *$\mathcal{C}$ is communicating through $c \in \Delta$ or x.*

Demonstração. This lemma is proved using a recursive proof that only terminates because the h-calculus does not permit cyclical references.

$\Sigma_C;\Delta\Delta_C \models \mathcal{C} :: (\Sigma_C, r : R; \Delta_C, x : A)$ *(by inversion on (Closed) and substitution)*
for some Σ_C and Δ_C

(by inversion on $\mathtt{Comb}$*)*

$-; y_1 : \bullet^{d_1} T_1^{s-(t+d_1)} 1, \cdots, y_n : \bullet^{d_n} T_n^{s-(t+d_n)} 1 \mid\!\frac{k,t}{s,c}\, \mathtt{Comb}(f, p, z \leftarrow (y_1, \cdots, y_n))$
$:: \left(z : \bullet^{\max(d_1,\cdots,d_n)} \bullet^{p} T_{out}^{s-(t+\max(d_1,\cdots,d_n)+p)} 1\right)$
where $s > t + \max(d_1, \cdots, d_n) + p$ and $p > 0$

$\forall i \in [1, n]. y_i \in \Delta\Delta_C$ *(By (proc))*

$\forall i \in [1, n].$ *either $y_i \in \Delta$ or $y_i \in \Delta_C$* *(By Lemma 2)*

We analyse two subcases: If there is a $k \in [1, n]$ such that $y_k \in \Delta$, and if for all $i \in [1, n].y_i \in \Delta_C$.

Case I ($\exists k \in [1, n].y_k \in \Delta$)**.** *By definition of communication, $\mathcal{C}$ is communicating through $y_k \in \Delta$ so case 2 applies.*

Case II ($\forall i \in [1, n].y_i \in \Delta_C$)**.**

$\mathcal{C} = \mathcal{C}'', \left[\mathtt{proc}\left(q_i^y \mid P_i^y \mid y_i\right)\right]_{\forall i \in [1,n]}$ *(by applications of Lemma 1)*
where $\Sigma_i; \Delta_i \mid\!\frac{k,t}{s,c}\, P_i^y :: \left(y_i : \bullet^{d_i} T_i^{s-(t+d_i)} 1\right)$
for some $\mathcal{C}''$, q_i^y, Σ_i and Δ_i

(by inversions on (Comb) and (Signal-1))

$P_i^y = \mathtt{Comb}(g_i, p_i, y_i \leftarrow (w_1, \cdots, w_m))$ *or* $P_i^y = \mathtt{Sig}(d_i, y_i \leftarrow e_i)$
for some $m, p_i, g_i, w_1, \cdots, w_m$ and e_i

We analyse two subcases: If there is a $k \in [1, n]$ such that $P_k^y = \mathtt{Comb}(g_k, p_k, y_k \leftarrow (w_1, \cdots, w_m))$, and if for all $i \in [1, n].P_i^y = \mathtt{Sig}(d_i, y_i \leftarrow e_i)$.

Case II.I ($\exists k \in [1, n].P_k^y = \mathtt{Comb}(g_k, p_k, y_k \leftarrow (w_1, \cdots, w_m))$)**.**

$\mathcal{C} = \mathcal{C}'', \mathtt{proc}\left(q_k \mid \mathtt{Comb}(g_k, p_k, y_i \leftarrow (w_1, \cdots, w_m)) \mid y_k\right)$ *(by substitution and generalization)*
for some $\mathcal{C}''$

By applying Lemma 4 recursively we know that either $\mathcal{C} \xrightarrow{\mathtt{Closed}} \mathcal{D}$, for some $\mathcal{D}$ (case 1), or $\mathcal{C}$ is communicating through $c \in \Delta$ or x (case 2).

Case II.II ($\forall i \in [1, n].P_i^y = \mathtt{Sig}(d_i, y_i \leftarrow e_i)$)**.**

(by substitution and generalization)

$\mathcal{C} = \mathcal{C}''', \mathtt{proc}\left(q \mid \mathtt{Comb}(f, p, z \leftarrow (y_1, \cdots, y_n)) \mid z\right), \left[\mathtt{proc}\left(q_i^y \mid \mathtt{Sig}(d_i, y_i \leftarrow e_i) \mid y_i\right)\right]_{\forall i \in [1,n]}$
for some $\mathcal{C}'''$

$\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{C}''', \texttt{proc}\Big(q \,\Big|\, \texttt{Sig}(\texttt{max}(d_1, \cdots, d_n) + p, z \leftarrow f(e_1, \cdots, e_n)) \,\Big|\, z\Big)$ *(by Comb)*
Case 1 applies.

■

Lemma 5 (Reg reduction). *If* $-;\Delta \models \texttt{Closed}\{\,\mathcal{C}\,\} :: (r : R; x : A)$ *with* $\texttt{proc}\Big(q \,\Big|\, \texttt{Reg}(z \leftarrow y) \,\Big|\, z\Big) \in \mathcal{C}$*, for any* $\mathcal{C}'$*,* q*,* y *and* $z \neq x$*, then either*

1. $\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{D}$*, for some* $\mathcal{D}$*, or*

2. $\mathcal{C}$ *is communicating through* $c \in \Delta$ *or* x*.*

Demonstração.

$\Sigma_C; \Delta\Delta_C \models \mathcal{C} :: (\Sigma_C, r : R; \Delta_C, x : A)$ *(by inversion on (Closed) and substitution)*
for some Σ_C *and* Δ_C
$-; y : \bullet^d T^{s-(t+d)} 1 \,\Big|\!\frac{k,t}{s,c}\, \texttt{Reg}(z \leftarrow y) :: \Big(z : \bullet^{s-t} T^s 1\Big)$ *(by inversion on* Reg*)*
where $s > t + d$
$y \in \Delta\Delta_C$ *(By (proc))*
Either $y \in \Delta$ *or* $y \in \Delta_C$ *(By Lemma 2)*

Case I ($y \in \Delta$)**.** *By definition of communication,* $\mathcal{C}$ *is communicating through* $y \in \Delta$ *so case 2 applies.*

Case II ($y \in \Delta_C$)**.**

$\mathcal{C} = \mathcal{C}'', \texttt{proc}\Big(q \,\Big|\, \texttt{Reg}(z \leftarrow y) \,\Big|\, z\Big), \texttt{proc}\Big(q^y \,\Big|\, P^y \,\Big|\, y\Big)$ *(by applications of Lemma 1)*
for some $\mathcal{C}''$ *and* q^y
(by inversions on (Comb) and (Signal-1))
$P^y = \texttt{Comb}(g, p, y \leftarrow (w_1, \cdots, w_m))$ *or* $P^y = \texttt{Sig}(d, y \leftarrow e)$
for some $m, p, g, w_1, \cdots, w_m$ *and* e

Case II.I ($P^y = \texttt{Comb}(g, p, y \leftarrow (w_1, \cdots, w_m))$)**.**

$\mathcal{C} = \mathcal{C}'', \texttt{proc}\Big(q_k \,\Big|\, \texttt{Comb}(g, p, y \leftarrow (w_1, \cdots, w_m)) \,\Big|\, y_k\Big)$ *(by substitution and generalization)*
for some $\mathcal{C}''$
By Lemma 4, either $\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{D}$*, for some* $\mathcal{D}$ *(case 1), or* $\mathcal{C}$ *is communicating through* $c \in \Delta$ *or* x *(case 2).*

Case II.II ($P^y = \texttt{Sig}(d, y \leftarrow e)$)**.**

(by substitution and generalization)
$\mathcal{C} = \mathcal{C}''', \texttt{proc}\Big(q \,\Big|\, \texttt{Reg}(z \leftarrow y) \,\Big|\, z\Big), \texttt{proc}\Big(q^y \,\Big|\, \texttt{Sig}(d, y \leftarrow e) \,\Big|\, y\Big), \underline{\texttt{env}}(c, s, t, k)$
where $\mathcal{C}'' = \mathcal{C}''', \underline{\texttt{env}}(c, s, t, k)$

(by Reg)

$\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{C}'', \texttt{idle}(r), \texttt{proc}\big(- \,\big|\, \texttt{tick}\ s-d;\ \texttt{clock};\ a \leftarrow \texttt{Sig}(0, a' \leftarrow e) \leftarrow \{-;-\};\ y \leftarrow a \,\big|\, y\big)$

fresh a and a′

Case 1 applies.

■

Lemma 6 (Internal resource request). *If* $\Sigma_C; \Delta\Delta_C \models \mathcal{C} :: (\Sigma_C; \Delta_C, x : A)$ *and* $\mathcal{C}$ *is requesting resource* $m \in \Sigma_C$ *then* $\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{D}$*, for some* $\mathcal{D}$*.*

Demonstração.

$\Sigma_C; \Delta\Delta_C \models \mathcal{C} :: (\Sigma_C; \Delta_C, x : A)$ *(from main assumption)*

$\mathcal{C} = \mathcal{C}', \texttt{proc}\big(p \,\big|\, b \leftarrow m \leftarrow \{\Sigma^p; \Delta^p\}; P \,\big|\, d\big)$ *(from Definition 38)*

for some $\mathcal{C}'$*,* b*,* Σ^p*,* Δ^p*,* P *and* d

$\mathcal{C} = \mathcal{C}'', \texttt{idle}(m := M[\Sigma^m][\Delta^m][y])$ *or* $\mathcal{C} = \mathcal{C}', \texttt{proc}\big(m \,\big|\, M[\Sigma^m][\Delta^m][y] \,\big|\, y\big)$ *(by Lemma 3)*

for some $\mathcal{C}''$*,* M*,* Σ^m*,* Δ^m *and* y

We proceed analysing both cases:

Case I ($\mathcal{C} = \mathcal{C}'', \texttt{idle}(m := M[\Sigma^m][\Delta^m][y])$).

(from Definition of context)

$\mathcal{C} = \mathcal{E}, \texttt{idle}(m := M[\Sigma^m][\Delta^m][y]), \texttt{proc}\big(p \,\big|\, b \leftarrow m \leftarrow \{\Sigma^p; \Delta^p\}; P \,\big|\, d\big)$

(by `(inst-1)`*)*

$\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{E}, \texttt{proc}\big(m \,\big|\, M[\Sigma^p/\Sigma^m][\Delta^p/\Delta^m][a/y] \,\big|\, a\big), \texttt{proc}\big(p \,\big|\, P[a/b] \,\big|\, d\big)$

(fresh a)

Case II ($\mathcal{C} = \mathcal{C}', \texttt{proc}\big(m \,\big|\, M[\Sigma^m][\Delta^m][y] \,\big|\, y\big)$).

(from Definition of context)

$\mathcal{C} = \mathcal{E}, \texttt{proc}\big(m \,\big|\, M[\Sigma^m][\Delta^m][y] \,\big|\, y\big), \texttt{proc}\big(p \,\big|\, b \leftarrow m \leftarrow \{\Sigma^p; \Delta^p\}; P \,\big|\, d\big)$

(by `(ext-2)`*)*

$\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{E}, \texttt{proc}\big(m \,\big|\, M[(\Sigma^p \times \Sigma^m)/\Sigma^m][(\Delta^p \times \Delta^m)/\Delta^m][a/y] \,\big|\, y\big), \texttt{proc}\big(p \,\big|\, P[a/b] \,\big|\, d\big)$

(fresh a)

■

Lemma 7 (Internal communication). *If* $\Sigma_C; \Delta\Delta_C \models \mathcal{C} :: (\Sigma_C, r : R; \Delta_C, x : A)$ *and* $\mathcal{C}$ *is communicating through* $c \in \Delta_C$ *then either*

1. $\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{D}$*, for some* $\mathcal{D}$*, or*

2. $\mathcal{C}$ *is communicating through* $c \in \Delta$ *or* x*.*

Demonstração.

$\Sigma_C; \Delta\Delta_C \models \mathcal{C} :: (\Sigma_C, r : R; \Delta_C, x : A)$ *(from main assumption)*

$(c : S) \in \Delta_C$ *(generalization)*

  for some S

$\mathcal{C} = \mathcal{C}', \underline{\mathsf{env}}(T_c, T_s, k, t)$ *(by Def. of Well-Formed Configuration)*

  for some $\mathcal{C}', T_s, T_c, k$ *and* t

$\mathcal{C}' = \mathcal{C}'', \mathsf{proc}\big(r^R \mid P^R \mid c\big)$ *(by Lemma 1)*

  where $\Sigma^R; \Delta^R \vdash^{k,t}_{T_s,T_c} P^R :: \big(c : S\big)$

  for some $\mathcal{C}'', r^R, P^R, \Delta^R$ *and* Σ^R

$\mathcal{C}' = \mathcal{C}''', \Big[\mathsf{proc}\big(r_i^L \mid P_i^L \mid x_i\big)\Big]_{\forall i \in [1,n]}$ *(by Lemma 2)*

  where $\Sigma_i^L; \Delta_i^L, c : S_i \vdash^{k,t}_{T_s,T_c} P_i^L :: \big(x_i : A_i\big)$ and $\prod_{i=1}^{n} S_i = S$

  for any $\mathcal{C}''', n, r_i^L, P_i^L, x_i, A_i, \Sigma_i^L$ and Δ_i^L

$\mathcal{C}' = \mathcal{C}^*, \Big[\mathsf{proc}\big(r_i^L \mid P_i^L \mid x_i\big)\Big]_{\forall i \in [1,n]}, \mathsf{proc}\big(r^R \mid P^R \mid c\big)$ *(by Def. of Well-Formed Configuration)*

  since $\forall i \in [1,n]. x_i \neq c$

  for some $\mathcal{C}^*$

$\mathcal{C} = \mathcal{C}^*, \Big[\mathsf{proc}\big(r_i^L \mid P_i^L \mid x_i\big)\Big]_{\forall i \in [1,n]}, \mathsf{proc}\big(r^R \mid P^R \mid c\big), \underline{\mathsf{env}}(T_c, T_s, k, t)$ *(by substitution)*

$\Sigma^{I*}; \Delta^{I*} \models \mathcal{C}^*, \underline{\mathsf{env}}(T_c, T_s, k, t) :: \big(\Sigma^{O*}; \Delta^{O*}\big)$ *(by generalization)*

  for some $\Sigma^{I*}, \Delta^{I*}, \Sigma^{O*}$ and Δ^{O*}

$\Sigma^L; \Delta^L, c : S \models \Big[\mathsf{proc}\big(r_i^L \mid P_i^L \mid x_i\big)\Big]_{\forall i \in [1,n]}, \underline{\mathsf{env}}(T_c, T_s, k, t)$ *(by (proc) and (compose))*

  $:: r_1^L : R_1^L, \cdots, r_n^L : R_n^L; x_1 : A_1, \cdots, x_n : A_n$

  where $\prod_{i=1}^{n} \Sigma_i^L = \Sigma^L$ and $\prod_{i=1}^{n} \Delta_i^L = \Delta^L$

  for some $R_1^L \cdots R_n^L$

$\Sigma^R; \Delta^R \models \mathsf{proc}\big(r^R \mid P^R \mid c\big), \underline{\mathsf{env}}(T_c, T_s, k, t) :: \big(r^R : R^R; c : S\big)$ *(by (proc) and (compose))*

  for some R^R

$\Sigma^{I*} \times \Sigma^L \times \Sigma^R; \Delta^{I*} \times \Delta^L \times \Delta^R, c : S$ *(by (proc) and (Compose)*

  $\models \mathcal{C}^*, \Big[\mathsf{proc}\big(r_i^L \mid P_i^L \mid x_i\big)\Big]_{\forall i \in [1,n]}, \mathsf{proc}\big(r^R \mid P^R \mid c\big), \underline{\mathsf{env}}(T_c, T_s, k, t)$

  $:: \big(\Sigma^{O*}, r^R : R^R, r_1^L : R_1^L, \cdots, r_n^L : R_n^L; \Delta^{O*}, c : S, x_1 : A_1, \cdots, x_n : A_n\big)$

$\Sigma_C = \Sigma^{I*} \times \Sigma^L \times \Sigma^R$ *(by Substitution)*

$\Sigma_C = \Sigma^{O*}, r^R : R^R, r_1^L : R_1^L, \cdots, r_n^L : R_n^L$

$\Delta_C, x : A = \Delta^{O*}, c : S, x_1 : A_1, \cdots, x_n : A_n$

$\Delta\Delta_C = \big(\Delta^{I*} \times \Delta^L \times \Delta^R\big), c : S$

Now we analyse each case of $c : S$ covering all possible cases in which $\mathcal{C}$ is communicating through $c \in \Delta_C$. We will, for each case of $c : S$, use inversion steps to infer the possible values of P^R, P_i^L and more generally $\mathcal{C}^*$ (excluding cases where $\mathcal{C}$ is not communicating through c), to prove that, for every case, either $\mathcal{C}$ is communicating externally or an operational semantics rule applies.

Case I ($S = \overrightarrow{\alpha}^\tau S'$).

$P^R = c \leftarrow \mathsf{put}\ v; P^{R\prime}$ *(by inversion of $\rightarrow R$)*

$\prod_{i=1}^{n} S_i = \overrightarrow{\alpha}^\tau S'$ *(by substitution)*

$S_i = \overrightarrow{\alpha}^\tau S_i'$ *or* $S_i = \bullet^\tau S_i'$ *(by definition of Merge ($\times$))*
  with at least one $\overrightarrow{\alpha}^\tau S_i'$ *instance*

$P_i^L = v_i \leftarrow \mathsf{get}\ c; P_i^{L\prime}$ *or* $P_i^L = P_i^{\mathsf{tick}}$ *(by inversion of ($\rightarrow$ L) and generalization)*
  with at least one $v_i \leftarrow \mathsf{get}\ c; P_i^{L\prime}$ *instance*
  for any v_i *and* P_i^{tick} *such that* $\Sigma_i^L; \Delta_i^L, c : \bullet^\tau S_i' \mid\!\frac{k,t}{T_s,T_c}\ P_i^{\mathsf{tick}} :: \left(x_i : A_i\right)$

$\mathcal{C} = \mathcal{E}, \left[\mathsf{proc}\left(r_i^L \mid v_i \leftarrow \mathsf{get}\ c; P_i^{L\prime} \mid x_i\right)\right]_{\forall i \in I^{\mathsf{get}}}, \mathsf{proc}\left(r^R \mid c \leftarrow \mathsf{put}\ v; P^{R\prime} \mid c\right)$ *(by substitution)*
  for $\mathcal{E} = \mathcal{C}^*, \left[\mathsf{proc}\left(r_i^L \mid P_i^{\mathsf{tick}} \mid x_i\right)\right]_{\forall i \in I^{\mathsf{tick}}}, \underline{\mathsf{env}}(T_c, T_s, k, t)$
  and any $I^\bullet$ *and* $I^\rightarrow$ *such that* $\{I^{\mathsf{tick}}, I^{\mathsf{get}}\}$ *is a partition of [1,n]*

$\Sigma^R; \Delta^{R\prime}, v : \alpha^\tau 1 \mid\!\frac{k,t}{T_s,T_c}\ c \leftarrow \mathsf{put}\ v; P^{R\prime} :: \left(c : \overrightarrow{\alpha}^\tau S_i\right)$ *(by inversion on ($\rightarrow$ R))*
  for any $\Delta^{R\prime}$

$(v : \alpha^\tau 1) \in \Delta^R$ *(by definition of context)*

$(v : \alpha^\tau 1) \in \left(\Delta^{I*} \times \Delta^L \times \Delta^R\right)$ *(by Lemma X)*

$(v : \alpha^\tau 1) \in \left(\Delta^{I*} \times \Delta^L \times \Delta^R\right), c : S$ *(by definition of Merge ($\times$))*

$(v : \alpha^\tau 1) \in \Delta\Delta_C$ *(by substitution)*

Either $(v : \alpha^\tau 1) \in \Delta$ *or* $(v : \alpha^\tau 1) \in \Delta_C$ *(by Lemma 2)*

Case I.I ($(v : \alpha^\tau 1) \in \Delta$).

$\mathcal{C} = \mathcal{A}, \mathsf{proc}\left(r^R \mid c \leftarrow \mathsf{put}\ v; P^{R\prime} \mid c\right)$ *(by generalization)*
  for some $\mathcal{A}$

$(v : \alpha^\tau 1) \in \Delta$ *(Case I.I)*

$\mathcal{C}$ *is communicating through* $v \in \Delta$ *(by definition of communication)*

Case 2 applies.

Case I.II ($(v : \alpha^\tau 1) \in \Delta_C$).

$\mathcal{C} = \mathcal{A}, \mathsf{proc}\left(r_{sig} \mid \mathsf{Sig}(\rho, v \leftarrow e) \mid v\right)$ *(by Lemma 1 and inversion on ($Signal - 2$))*
  for any e, and some $\mathcal{A}$ *and* $\rho \leq 0$

(by substitution and definition of Well-Formed Configuration)

$\mathcal{C} = \mathcal{E}', \Big[\mathsf{proc}\Big(r_i^L \;\Big|\; v_i \leftarrow \mathsf{get}\; c; P_i^{L\prime} \;\Big|\; x_i\Big)\Big]_{\forall i \in I^{\mathsf{get}}}, \mathsf{proc}\Big(r^R \;\Big|\; c \leftarrow \mathsf{put}\; v; P^{R\prime} \;\Big|\; c\Big)$
  $, \mathsf{proc}\Big(r_{sig} \;\Big|\; \mathsf{Sig}\,(\rho, v \leftarrow e) \;\Big|\; v\Big)$
  where $\mathcal{E} = \mathcal{E}', \mathsf{proc}\Big(r_{sig} \;\Big|\; \mathsf{Sig}\,(\rho, v \leftarrow e) \;\Big|\; v\Big)$
$\mathcal{C} \xrightarrow{\mathsf{Closed}} \mathcal{E}', \Big[\mathsf{proc}\Big(r_i^L \;\Big|\; P_i^{L\prime}[v/v_i] \;\Big|\; x_i\Big)\Big]_{\forall i \in I^{\mathsf{get}}}, \mathsf{proc}\Big(r^R \;\Big|\; P^{R\prime} \;\Big|\; c\Big)$ *(by $(\rightarrow 1)$)*
  $, \mathsf{proc}\Big(r_{sig} \;\Big|\; \mathsf{Sig}\,(\rho, v \leftarrow e) \;\Big|\; v\Big)$
Case 1 applies.

Case II ($S = \overleftarrow{\alpha}^\tau S'$).

$P^R = v \leftarrow \mathsf{get}\; c; P^{R\prime}$ *(by inversion of $(\leftarrow R)$)*

$\prod_{i=1}^{n} S_i = \overleftarrow{\alpha}^\tau S'$ *(by substitution)*

$S_i = \overrightarrow{\alpha}^\tau S_i'$ *or* $S_i = \overleftarrow{\alpha}^\tau S_i'$ *or* $S_i = \bullet^\tau S_i'$ *(by definition of Merge $(\times)$)*
  with exactly one $\overleftarrow{\alpha}^\tau S_i'$ *instance*

(by inversion of $(\rightarrow L)$ $(\leftarrow L)$ and generalization)

$P_i^L = v_i \leftarrow \mathsf{get}\; c; P_i^{L\prime}$ *or* $P_i^L = c \leftarrow \mathsf{put}\; v_i; P_i^{L\prime}$ *or* $P_i^L = P_i^{\mathsf{tick}}$
  with exactly one $c \leftarrow \mathsf{put}\; v_i; P_i^{L\prime}$ *instance*
  for any v_i *and* P_i^{tick} *such that* $\Sigma_i^L; \Delta_i^L, c : \bullet^\tau S_i' \vdash_{T_s, T_c}^{k,t} P_i^{\mathsf{tick}} :: \big(x_i : A_i\big)$
$\mathcal{C} = \mathcal{E}, \Big[\mathsf{proc}\Big(r_i^L \;\Big|\; v_i \leftarrow \mathsf{get}\; c; P_i^{L\prime} \;\Big|\; x_i\Big)\Big]_{\forall i \in I^{\mathsf{get}}}, \mathsf{proc}\Big(r_k^L \;\Big|\; c \leftarrow \mathsf{put}\; v_k; P_k^{L\prime} \;\Big|\; x_k\Big)$ *(by substitution)*
  $, \mathsf{proc}\Big(r^R \;\Big|\; v \leftarrow \mathsf{get}\; c; P^{R\prime} \;\Big|\; c\Big)$
  for $\mathcal{E} = \mathcal{C}^*, \Big[\mathsf{proc}\Big(r_i^L \;\Big|\; P_i^{\mathsf{tick}} \;\Big|\; x_i\Big)\Big]_{\forall i \in I^{\mathsf{tick}}}, \underline{\mathsf{env}}\,(T_c, T_s, k, t)$
  for any $k \in [1, n]$ *and* $I^{\bullet}$ *and* $I^{\rightarrow}$ *such that* $\{I^{\mathsf{tick}}, I^{\mathsf{get}}, \{k\}\}$ *is a partition of [1,n]*
$\Sigma_k^L; \Delta_k^{L\prime}, v_k : \alpha^\tau 1, c : \overleftarrow{\alpha}^\tau S_k' \vdash_{T_s, T_c}^{k,t} c \leftarrow \mathsf{put}\; v_k; P_k^{L\prime} :: \big(x_k : A_k\big)$ *(by inversion on $(\leftarrow L)$)*
  for any $\Delta_k^{L\prime}$
$(v_k : \alpha^\tau 1) \in \Delta_k^L$ *(by definition of context)*
$(v_k : \alpha^\tau 1) \in \Delta^L$ *(because $\prod_{i=1}^n \Delta_i^L = \Delta^L$ and Lemma X)*
$(v_k : \alpha^\tau 1) \in \big(\Delta^{I*} \times \Delta^L \times \Delta^R\big)$ *(by Lemma X)*
$(v_k : \alpha^\tau 1) \in \big(\Delta^{I*} \times \Delta^L \times \Delta^R\big), c : S$ *(by definition of Merge $(\times)$)*
$(v_k : \alpha^\tau 1) \in \Delta\Delta_C$ *(by substitution)*
Either $(v_k : \alpha^\tau 1) \in \Delta$ *or* $(v_k : \alpha^\tau 1) \in \Delta_C$ *(by Lemma 2)*

Case II.I ($(v_k : \alpha^\tau 1) \in \Delta$).

$\mathcal{C} = \mathcal{A}, \mathsf{proc}\Big(r_k^L \;\Big|\; c \leftarrow \mathsf{put}\; v_k; P^{L\prime}{}_k \;\Big|\; x_k\Big)$ *(by generalization)*
  for some $\mathcal{A}$
$(v_k : \alpha^\tau 1) \in \Delta$ *(Case II.I)*
$\mathcal{C}$ *is communicating through* $v_k \in \Delta$ *(by definition of communication)*
Case 2 applies.

Case II.II ($(v_k : \alpha^\tau 1) \in \Delta_C$).

$\mathcal{C} = \mathcal{A}, \mathtt{proc}\Big(r_{sig} \,\Big|\, \mathtt{Sig}(\rho, v_k \leftarrow e) \,\Big|\, v_k\Big)$ *(by Lemma 1 and inversion on $(Signal-2)$)*

for any $\mathcal{A}$, e and ρ such that $\rho \leq 0$

(by substitution and definition of Well-Formed Configuration)

$\mathcal{C} = \mathcal{E}', \Big[\mathtt{proc}\Big(r_i^L \,\Big|\, v_i \leftarrow \mathtt{get}\ c; P_i^{L\prime} \,\Big|\, x_i\Big)\Big]_{\forall i \in I^{\mathtt{get}}}, \mathtt{proc}\Big(r_k^L \,\Big|\, c \leftarrow \mathtt{put}\ v_k; P_k^{L\prime} \,\Big|\, x_k\Big)$

$, \mathtt{proc}\Big(r^R \,\Big|\, v \leftarrow \mathtt{get}\ c; P^{R\prime} \,\Big|\, c\Big), \mathtt{proc}\Big(r_{sig} \,\Big|\, \mathtt{Sig}(\rho, v_k \leftarrow e) \,\Big|\, v_k\Big)$

where $\mathcal{E} = \mathcal{E}', \mathtt{proc}\Big(r_{sig} \,\Big|\, \mathtt{Sig}(\rho, v_k \leftarrow e) \,\Big|\, v_k\Big)$

$\mathcal{C} \xrightarrow{\mathtt{Closed}} \mathcal{C}^{*\prime}, \Big[\mathtt{proc}\Big(r_i^L \,\Big|\, P_i^{L\prime}[v_k/v_i] \,\Big|\, x_i\Big)\Big]_{\forall i \in I^{\mathtt{get}}}, \mathtt{proc}\Big(r_k^L \,\Big|\, P_k^{L\prime} \,\Big|\, x_k\Big)$ *(by $(\rightarrow 2)$)*

$, \mathtt{proc}\Big(r^R \,\Big|\, P^{R\prime}[v_k/v] \,\Big|\, c\Big), \mathtt{proc}\Big(r_{sig} \,\Big|\, \mathtt{Sig}(\rho, v_k \leftarrow e) \,\Big|\, v_k\Big)$

Case 1 applies.

Case III ($S = \alpha^\tau S'$).

$P^R = \mathtt{Sig}(\tau, c \leftarrow e)$ *(by inversion of (Signal-2))*

$\prod_{i=1}^{n} S_i = \alpha^\tau S'$ *(by substitution)*

$S_i = \alpha^\tau S_i'$ *(by definition of Merge ($\times$))*

where $\prod_{i=1}^{n} S_i' = S'$

$P_i^L = \mathtt{Comb}(f_i, p_i, x_i \leftarrow (y_{i1}, \cdots, y_{im}))$, *or* *(by inversion of (Comb), (Reg), $(\rightarrow R)$ and $(\leftarrow L)$)*

$= \mathtt{Reg}(x_i \leftarrow c)$, *or*

$= z_i \leftarrow \mathtt{put}\ c; P_i'^L$

where there is one $k \in [1, m]$ such that $y_{ik} = c$

for any f_i, p_i, m, y_{ij}, y_i, z_i and $P_i'^L$

$\mathcal{C} = \mathcal{E}, \Big[\mathtt{proc}\Big(r_i^L \,\Big|\, \mathtt{Comb}(f_i, p_i, x_i \leftarrow (y_{i1}, \cdots, y_{im})) \,\Big|\, x_i\Big)\Big]_{\forall i \in I^{\mathtt{comb}}}$ *(by substitution)*

$, \Big[\mathtt{proc}\Big(r_i^L \,\Big|\, \mathtt{Reg}(x_i \leftarrow c) \,\Big|\, x_i\Big)\Big]_{\forall i \in I^{\mathtt{reg}}}, \Big[\mathtt{proc}\Big(r_i^L \,\Big|\, z_i \leftarrow \mathtt{put}\ c; P_i'^L \,\Big|\, x_i\Big)\Big]_{\forall i \in I^{\mathtt{put}}}$

$, \mathtt{proc}\Big(r^R \,\Big|\, \mathtt{Sig}(\tau, c \leftarrow e) \,\Big|\, c\Big)$

for some $\mathcal{E} = \mathcal{C}^*, \underline{\mathtt{env}}(T_c, T_s, k, t)$

for any ρ, $I^{\mathtt{comb}}$, $I^{\mathtt{reg}}$ and $I^{\mathtt{put}}$ where $\{I^{\mathtt{comb}}, I^{\mathtt{reg}}, I^{\mathtt{put}}\}$ is a partition of [1,n]

If $I^{\mathtt{comb}}$ is not empty Lemma 4 is applied and if $I^{\mathtt{reg}}$ is not empty Lemma 5 is applied. In both of these cases, $\mathcal{C} \xrightarrow{\mathtt{Closed}} \mathcal{D}$, for some $\mathcal{D}$, and Case 1 applies. The remaining case is the one where both $I^{\mathtt{comb}}$ and $I^{\mathtt{reg}}$ are empty and $I^{\mathtt{put}} = [1, n]$:

$\mathcal{C} = \mathcal{E}, \Big[\mathtt{proc}\Big(r_i^L \,\Big|\, z_i \leftarrow \mathtt{put}\ c; P_i'^L \,\Big|\, \rho\Big)\Big]_{\forall i \in [1,n]}, \mathtt{proc}\Big(r^R \,\Big|\, \mathtt{Sig}(\tau, c \leftarrow e) \,\Big|\, c\Big)$

For simplification, we choose to focus on only one specific $k \in [1, m]$

$\mathcal{C} = \mathcal{E}', \mathtt{proc}\Big(r_k^L \,\Big|\, z_k \leftarrow \mathtt{put}\ c; P_k'^L \,\Big|\, x_k\Big), \mathtt{proc}\Big(r^R \,\Big|\, \mathtt{Sig}(\tau, c \leftarrow e) \,\Big|\, c\Big)$ *(by substitution)*

for some $\mathcal{E}'$

Now we analyse separately the case where $x_k = z_k$ and the case where $x_k \neq z_k$:

Case III.I ($x_k = z_k$).

$\Sigma_k; \Delta_k, c : \alpha^\tau S'_k \vdash^{k,t}_{T_s,T_s} \left(x_k \leftarrow \mathsf{put}\ c; P'^L_k\right) :: \left(x_k : \overrightarrow{\alpha}^\tau A_k\right)$ *(by inversion on (→ R))*

$\left(x_k : \overrightarrow{\alpha}^\tau A_k\right) \in \Delta_C$

Since $x_k \neq x$

$\mathcal{C} = \mathcal{E}'', \left[\mathsf{proc}\left(q_i \mid P^*_i \mid w_i\right)\right]_{\forall i \in [1,l]}$ *(by Lemma 2)*

where $w_i : A_i$ *and* $\prod_{i=1}^{l} B_i = \overrightarrow{\alpha}^\tau A_k$

$B_i = \overrightarrow{\alpha}^\tau B'_i$ *or* $B_i = \bullet^\tau B'_i$ *(by Definition of merge (×))*

for any B'_i

$P^*_i = v_i \leftarrow \mathsf{get}\ x_k; P^{*\prime}_i$ *or* $P^*_i = \mathsf{tick}\ \tau; P^{*\prime}_i$ *(by inversion of (→ L) and (tick))*

for any v_i *and* $P^{*\prime}_i$

(by substitution and Definition of Configuration)

$\mathcal{C} = \mathcal{A}, \left[\mathsf{proc}\left(q_i \mid v_i \leftarrow \mathsf{get}\ x_k; P^{*\prime}_i \mid w_i\right)\right]_{\forall i \in I^{\mathsf{get}}}$

$, \mathsf{proc}\left(r^L_k \mid x_k \leftarrow \mathsf{put}\ c; P'^L_k \mid x_k\right), \mathsf{proc}\left(r^R \mid \mathsf{Sig}(\tau, c \leftarrow e) \mid c\right)$

for some $\mathcal{A}$ *and* $I^{\mathsf{get}} \subseteq [1, l]$

(by (→ 1))

$\mathcal{C} \xrightarrow{\mathsf{Closed}} \mathcal{A}, \left[\mathsf{proc}\left(q_i \mid P^{*\prime}_i[c/v_i] \mid w_i\right)\right]_{\forall i \in I^{\mathsf{get}}}, \mathsf{proc}\left(r^L_k \mid P'^L_k \mid x_k\right), \mathsf{proc}\left(r^R \mid \mathsf{Sig}(\tau, c \leftarrow e) \mid c\right)$

Case 1 applies.

Case III.II ($x_k \neq z_k$).

$\Sigma_k; \Delta_k, c : \alpha^\tau S'_k, z_k : \overleftarrow{\alpha}^\tau A_k \vdash^{k,t}_{T_s,T_s} \left(z_k \leftarrow \mathsf{put}\ c; P'^L_k\right) :: \left(x_k : C\right)$ *(by inversion on (← L))*

$\left(z_k : \overleftarrow{\alpha}^\tau A_k\right) \in \Delta^{I*}$

$\left(z_k : \overleftarrow{\alpha}^\tau A_k\right) \in \Delta\Delta_C$ *(by substitution)*

Either $z_k \in \Delta$ *or* $z_k \in \Delta_C$ *(by Lemma X)*

If $z_k \in \Delta$ *then Case 2 applies. Now we continue analysing the case where* $z_k \in \Delta_C$*. In this case, there could be other processes consuming* z_k *with other types. By Definition of merge, for the configuration to be well-typed, the types are either* $z_k : \overleftarrow{\alpha}^\tau A'_k$ *or* $z_k : \bullet^\alpha A'_k$*, and the result of their merge is* $z_k : \overleftarrow{\alpha}^\tau A^R_k$*. Now we proceed applying Lemma 2 on* $z_k : \overleftarrow{\alpha}^\tau A'_k$ *and Lemma 1 on* $z_k : \overleftarrow{\alpha}^\tau A^R_k$*, leaving us with the configuration:*

$\mathcal{C} = \mathcal{E}'', \mathsf{proc}\left(q \mid v \leftarrow \mathsf{get}\ z_k; Q' \mid z_k\right), \left[\mathsf{proc}\left(q_i \mid v_i \leftarrow \mathsf{get}\ z_k; Q_i \mid w_i\right)\right]_{\forall i \in I^{\mathsf{get}}}$

for any v, Q', q_i, v_i, w_i *and* I^{get}

(by substitution and Definition of Configuration)

$\mathcal{C} = \mathcal{A}, \mathsf{proc}\left(q \mid v \leftarrow \mathsf{get}\ z_k; Q' \mid z_k\right), \left[\mathsf{proc}\left(q_i \mid v_i \leftarrow \mathsf{get}\ z_k; Q_i \mid w_i\right)\right]_{\forall i \in I^{\mathsf{get}}}$

$, \mathsf{proc}\left(r^L_k \mid z_k \leftarrow \mathsf{put}\ c; P'^L_k \mid x_k\right), \mathsf{proc}\left(r^R \mid \mathsf{Sig}(\tau, c \leftarrow e) \mid c\right)$

for some $\mathcal{A}$ *and* $I^{\mathsf{get}} \subseteq [1, l]$

$\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{A}, \mathsf{proc}\Big(q \mid Q'[c/v] \mid z_k\Big), \Big[\mathsf{proc}\Big(q_i \mid Q_i[c/v_i] \mid w_i\Big)\Big]_{\forall i \in I^{\texttt{get}}}$ *(by (→ 2))*
$\quad , \mathsf{proc}\Big(r^L_k \mid P'^L_k \mid x_k\Big), \mathsf{proc}\Big(r^R \mid \texttt{Sig}(\tau, c \leftarrow e) \mid c\Big)$
Case 1 applies.

Case IV ($S = \overrightarrow{\oplus}_c\{\ell : S_\ell\}_{\ell \in L}$).

$P^R = c.k; P^{R\prime}$ *(by inversion of ($\overrightarrow{\oplus}$R))*
$\quad$ *for any $P^{R\prime}$ and $k \in L$*
$\prod_{i=1}^{n} S_i = \overrightarrow{\oplus}_c\{\ell : S_\ell\}_{\ell \in L}$ *(by substitution)*
$S_i = \overrightarrow{\oplus}_c\{\ell : S_{i\ell}\}_{\ell \in L}$ *or S_i = any other type* *(by definition of Merge (×))*
$\quad$ *with at least one $\overrightarrow{\oplus}_c\{\ell : S_{i\ell}\}_{\ell \in L}$ instance*
$P^L_i = \mathsf{case}\ c\ \mathsf{of}\ \Big\{\ell \Rightarrow P^{L\prime}_{i\ell}\Big\}_{\ell \in L}$ *or P^L_i = any other process* *(by inversion of ($\overrightarrow{\oplus}$ L))*
$\quad$ *with at least one* $\mathsf{case}\ c\ \mathsf{of}\ \Big\{\ell \Rightarrow P^{L\prime}_{i\ell}\Big\}_{\ell \in L}$ *instance*
$\quad$ *for any $P^{L\prime}_{i\ell}$*
$\mathcal{C} = \mathcal{E}, \mathsf{proc}\Big(r^R \mid c.k; P^{R\prime} \mid c\Big), \Big[\mathsf{proc}\Big(r^L_i \mid \mathsf{case}\ c\ \mathsf{of}\ \Big\{\ell \Rightarrow P^{L\prime}_{i\ell}\Big\}_{\ell \in L} \mid x_i\Big)\Big]_{\forall i \in I^{\oplus}}$ *(by substitution)*
$\quad$ *where $\mathcal{E} = \mathcal{C}^*, \underline{\mathsf{env}}(T_c, T_s, k, t)$ and $I^{\oplus} \subseteq [1, n]$*
$\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{E}, \mathsf{proc}\Big(r^R \mid P^{R\prime} \mid c\Big), \Big[\mathsf{proc}\Big(r^L_i \mid P^{L\prime}_{ik} \mid x_i\Big)\Big]_{\forall i \in I^{\oplus}}$ *(by (⊕ 1))*
Case 1 applies.

Case V ($S = \overleftarrow{\oplus}_c\{\ell : S_\ell\}_{\ell \in L}$).

$P^R = \mathsf{case}\ c\ \mathsf{of}\ \Big\{\ell \Rightarrow P^{R\prime}_\ell\Big\}_{\ell \in L}$ *(by inversion of ($\overleftarrow{\oplus}$R))*
$\quad$ *for any $P^{R\prime}_\ell$*
$\prod_{i=1}^{n} S_i = \overleftarrow{\oplus}_c\{\ell : S_\ell\}_{\ell \in L}$ *(by substitution)*
$S_i = \overleftarrow{\oplus}_c\{\ell : S_{i\ell}\}_{\ell \in L}$ *or* $\overrightarrow{\oplus}_c\{\ell : S_{i\ell}\}_{\ell \in L}$ *or S_i = any other type* *(by definition of Merge (×))*
$\quad$ *with exactly one $\overleftarrow{\oplus}_c\{\ell : S_{i\ell}\}_{\ell \in L}$ instance*
(by inversion of ($\overleftarrow{\oplus}$ L), ($\overrightarrow{\oplus}$ L))
$P^L_i = c.k$ *or* $P^L_i = \mathsf{case}\ c\ \mathsf{of}\ \Big\{\ell \Rightarrow P^{L\prime}_{i\ell}\Big\}_{\ell \in L}$ *or P^L_i = any other process*
$\quad$ *with exactly one $c.k$ instance*
$\quad$ *for any k and $P^{L\prime}_{i\ell}$*
$\mathcal{C} = \mathcal{E}, \mathsf{proc}\Big(r^L_j \mid c.k; P^{R\prime}_j \mid x_j\Big), \mathsf{proc}\Big(r^R \mid \mathsf{case}\ c\ \mathsf{of}\ \Big\{\ell \Rightarrow P^{R\prime}_\ell\Big\}_{\ell \in L} \mid c\Big)$ *(by substitution)*
$\quad , \Big[\mathsf{proc}\Big(r^L_i \mid \mathsf{case}\ c\ \mathsf{of}\ \Big\{\ell \Rightarrow P^{L\prime}_{i\ell}\Big\}_{\ell \in L} \mid x_i\Big)\Big]_{\forall i \in I^{\oplus}}$
$\quad$ *where $\mathcal{E} = \mathcal{C}^*, \underline{\mathsf{env}}(T_c, T_s, k, t)$, $j \notin I^{\oplus}$ and $\{I^{\oplus}, \{j\}\} \subseteq [1, n]$*
$\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{E}, \mathsf{proc}\Big(r^L_j \mid P^{R\prime}_j \mid x_j\Big), \mathsf{proc}\Big(r^R \mid P^{R\prime}_k \mid c\Big), \Big[\mathsf{proc}\Big(r^L_i \mid P^{L\prime}_{ik} \mid x_i\Big)\Big]_{\forall i \in I^{\oplus}}$ *(by (⊕ 2))*
Case 1 applies.

Case VI ($S = S_1 \otimes S_2$).

$P^R = (c \to (c_1, c_2)).\left(P_1^R \parallel P_2^R\right)$ *(by inversion of (⊗ R))*

for any c_1, c_2, P_1^R *and* P_2^R

$\prod_{i=1}^{n} S_i = S_1 \otimes S_2$ *(by substitution)*

$S_i = S_{i1} \otimes S_{i2}$ *(by definition of Merge (×))*

for any S_{i1} *and* S_{i2}

$P_i^L = (c_{i1}, c_{i2}) \leftarrow c; P_i^{L\prime}$ *(by inversion of (⊗ L))*

for any c_{i1}, c_{i2}, $P_i^{L\prime}$

$\mathcal{C} = \mathcal{E}, \mathtt{proc}\left(r^R \mid (c \to (c_1, c_2)).\left(P_1^R \parallel P_2^R\right) \mid c\right)$ *(by substitution)*

$, \left[\mathtt{proc}\left(r_i^L \mid (c_{i1}, c_{i2}) \leftarrow c; P_i^{L\prime} \mid x_i\right)\right]_{\forall i \in [1,n]}$

where $\mathcal{E} = \mathcal{C}^*, \underline{\mathtt{env}}(T_c, T_s, k, t)$

$\mathcal{C} \xrightarrow{\mathtt{Closed}} \mathcal{E}, \mathtt{proc}\left(r_1^R \mid P_1^R[a_1/c_1][a_2/c_2] \mid a_1\right), \mathtt{proc}\left(r_2^R \mid P_2^R \mid a_2\right)$ *(by (⊗ 2))*

$, \left[\mathtt{proc}\left(r_i^L \mid P_i^{L\prime}[a_1/c_{i1}][a_2/c_{i2}] \mid x_i\right)\right]_{\forall i \in [1,n]}$

(fresh r_1^R, r_2^R, a_1 *and* a_2*)*

Case 1 applies.

Case VII ($S = \bullet^{\tau_1} T^{\tau_2} 1$).

$P^R = \mathtt{Comb}(f, p, c \leftarrow w_1, \cdots, w_n)$, *or* *(by inversion of (Comb), (Reg) and (Signal-1))*

$= \mathtt{Reg}(c \leftarrow w)$, *or*

$= \mathtt{Sig}(\tau_1, c \leftarrow e)$.

where $p > 0$ *and* $\mathtt{max}(\mathtt{delay}(w_1), \cdots, \mathtt{delay}(w_n)) + p = \tau_1$

for any f, n, w_i, w *and* e

(It is important to note another possibility is $\mathtt{tick}\ \tau_1; aux \leftarrow \mathtt{Sig}(0, aux \leftarrow e); c \leftarrow aux$ *but from process equivalence we have that* $\mathtt{tick}\ \tau_1; aux \leftarrow \mathtt{Sig}(0, aux \leftarrow e); c \leftarrow aux \equiv \mathtt{Sig}(\tau_1, aux \leftarrow e)$ *so we only consider the latter version)*

We analyse each case separately:

Case VII.I ($P^R = \mathtt{Comb}(f, p, c \leftarrow w_1, \cdots, w_n)$).

$\mathcal{C} \xrightarrow{\mathtt{Closed}} \mathcal{D}$, *for some* $\mathcal{D}$ *(by Lemma 4)*

Case 1 applies.

Case VII.II ($P^R = \mathtt{Reg}(c \leftarrow w)$).

$\mathcal{C} \xrightarrow{\mathtt{Closed}} \mathcal{D}$, *for some* $\mathcal{D}$ *(by Lemma 5)*

Case 1 applies.

Case VII.III ($P^R = \mathtt{Sig}(\tau_1, c \leftarrow e)$).

$\prod_{i=1}^{n} S_i = \bullet^{\tau_1} T^{\tau_2} 1$ *(by substitution)*

$S_i = \bullet^{\tau_1} T^{\tau_2} 1$ *(by definition of Merge (×))*

$P_i^L = \mathsf{Comb}\,(f_i, p_i, z_i \leftarrow (u_1, \cdots, u_m))$ *or* $P_i^L = \mathsf{Reg}\,(c \leftarrow u)$ *(by inversion of (Comb) and (Reg))*

where there is one $j \in [1, m]$ *such that* $y_j = c$

for some m, f_i, p_i, z_i, u_i *and* u.

In the case of Comb *we apply Lemma 4 and in the case of* Reg *we apply Lemma 5. In both cases we get that* $\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{D}$ *for some* $\mathcal{D}$. *Case 1 applies.*

Case VIII ($S = \mu z.S$)**.**

$P^R = L : P^{R\prime}$ *(by inversion of (μ R))*

for any L *and* $P^{R\prime}$

$\mathcal{C} = \mathcal{E}, \mathsf{proc}\left(r^R \,\middle|\, L : P^{R\prime} \,\middle|\, c\right)$ *(by substitution)*

for some $\mathcal{E}$

$\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{E}, \mathsf{proc}\left(r^R \,\middle|\, P^{R\prime}\left[\left(L : P^{R\prime}\right)/L\right] \,\middle|\, c\right)$ *(by (Loop))*

Case 1 applies.

Case IX ($S = 1$)**.**

$P^R = \mathsf{end}\ c$ *(by inversion of* $(1$ *R))*

for any L *and* $P^{R\prime}$

$\mathcal{C} = \mathcal{E}, \mathsf{proc}\left(r^R \,\middle|\, \mathsf{end} \,\middle|\right)$ *(by substitution)*

for some $\mathcal{E}$

$\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{E}, \mathtt{idle}\left(r^R\right)$ *(by* (1)*)*

Case 1 applies.

■

Lemma 8. *If* $\Sigma^I; \Delta^I \models \mathcal{C} :: \left(\Sigma^O; \Delta^O\right)$, *then either*

I. $\mathcal{C} \rightarrow \mathcal{C}'$, *for some* $\mathcal{C}'$, *or*

II. $\mathcal{C}$ *is communicating through* $c \in \Delta^I$, *or*

III. $\mathcal{C}$ *is communicating through* $c \in \Delta^O$, *or*

IV. $\mathcal{C}$ *is requesting resource from* $r \in \Sigma^I$, *or*

V. $\mathcal{C}$ *does not have* proc *objects (computation is over).*

Demonstração. By induction on the configuration type rules that form $\mathcal{C}$.

Case X (One semantic object)**.**

Case X.I (idle)**.**

$$\frac{\texttt{inst}(\texttt{DEF}(P), R)}{-;- \models \texttt{idle}(r := P), \underline{\texttt{env}}(s,c,k,t) :: (r : R; -)}\,\texttt{idle}$$

$\mathcal{C}$ does not have proc *objects (computation is over).*

Case X.II (proc)**.**

$$\frac{[\Sigma]^{+T}; [\Delta]^{+T} \stackrel{k,t}{\underset{s,c}{\vdash}} P :: \left(x : [A]^{+T}\right) \quad \texttt{inst}((\Sigma;\Delta;A), R) \quad (T = k \times s + t)}{\Sigma;\Delta \models \texttt{proc}\big(r \,\big|\, P \,\big|\, x\big), \underline{\texttt{env}}(s,c,k,t) :: (r : R; x : A)}\,\texttt{proc}$$

For every process P, one of the cases apply. For instance:

- *If $P = x \leftarrow y$, then $\mathcal{C} \to \mathcal{C}'$, for some $\mathcal{C}'$,*
- *if $P = x \leftarrow$ get y, then $\mathcal{C}$ is communicating through $c \in \Delta^O$ or $c \in \Delta^I$,*
- *if $P =$ tick τ, then $\mathcal{C} \to \mathcal{C}'$, for some $\mathcal{C}'$,*
- *if $P = x \leftarrow r \leftarrow \{\Sigma;\Delta\}$, then $\mathcal{C}$ is requesting resource from $r \in \Sigma^I$.*

Case XI (compose)**.**

$$\frac{\Sigma_1^I;\Delta_1^I \models \mathcal{C}_1, \underline{\texttt{env}}(s,c,k,t) :: \left(\Sigma_1^O;\Delta_1^O\right) \quad \Sigma_2^I;\Delta_2^I \models \mathcal{C}_2, \underline{\texttt{env}}(s,c,k,t) :: \left(\Sigma_2^O;\Delta_2^O\right)}{\Sigma_1^I \times \Sigma_2^I; \Delta_1^I \times \Delta_2^I \models \mathcal{C}_1\mathcal{C}_2, \underline{\texttt{env}}(s,c,k,t) :: \left(\Sigma_1^O\Sigma_2^O;\Delta_1^O\Delta_2^O\right)}\,\texttt{Compose}$$

By induction hypothesis, we have that **IH1** *and* **IH2** *simultaneously:*

(IH1) *Either*

I. *$\mathcal{C}_1, \underline{\texttt{env}}(c,s,k,t) \to \mathcal{C}_1', \underline{\texttt{env}}(c,s,k',t')$, or*

II. *$\mathcal{C}_1$ is communicating through $c \in \Delta_1^I$, or*

III. *$\mathcal{C}_1$ is communicating through $c \in \Delta_1^O$, or*

IV. *$\mathcal{C}_1$ is requesting resource from $r \in \Sigma_1^I$, or*

V. *$\mathcal{C}_1$ does not have* proc *objects (computation is over).*

(IH2) *Either*

I. *$\mathcal{C}_2, \underline{\texttt{env}}(c,s,k,t) \to \mathcal{C}_2', \underline{\texttt{env}}(c,s,k',t')$, or*

II. *$\mathcal{C}_2$ is communicating through $c \in \Delta_2^I$, or*

III. *$\mathcal{C}_2$ is communicating through $c \in \Delta_2^O$, or*

IV. *$\mathcal{C}_2$ is requesting resource from $r \in \Sigma_2^I$, or*

V. *$\mathcal{C}_2$ does not have proc objects (computation is over).*

We proceed analysing all possible cases provided by **(IH1)** *and* **(IH2)**. *We start analysing cases of* **(IH1)** *for any kind of $\mathcal{C}_2$, using* **(IH2)** *only in the last case:*

Case XI.I ($\mathcal{C}_1, \underline{\mathsf{env}}(c,s,k,t) \rightarrow \mathcal{C}_1', \underline{\mathsf{env}}(c,s,k',t')$, for some $\mathcal{C}_1'$).

$\mathcal{C}_1\mathcal{C}_2 \rightarrow \mathcal{C}_1'\mathcal{C}_2$ *(by Lemma 7)*

Case XI.II ($\mathcal{C}_1$ is communicating through $c \in \Delta_1^I$).

$\mathcal{C}_1\mathcal{C}_2$ is communicating through $c \in \Delta_1^I \times \Delta_2^I$ *(by Lemma 8)*

Case XI.III ($\mathcal{C}_1$ is communicating through $c \in \Delta_1^O$).

$\mathcal{C}_1\mathcal{C}_2$ is communicating through $c \in \Delta_1^O\Delta_2^O$ *(by Lemma 9)*

Case XI.IV ($\mathcal{C}_1$ is requesting resource from $r \in \Sigma_1^I$).

$\mathcal{C}_1\mathcal{C}_2$ is requesting resource from $r \in \Sigma_1^I \times \Sigma_2^I$ *(by Lemma 10)*

Case XI.V ($\mathcal{C}_1$ does not have *proc* objects). *By* **IH2***:*

Case XI.V.I ($\mathcal{C}_2 \rightarrow \mathcal{C}_2'$, for some $\mathcal{C}_2'$).

$\mathcal{C}_1\mathcal{C}_2 \rightarrow \mathcal{C}_1\mathcal{C}_2'$ *(by Lemma 7)*

Case XI.V.II ($\mathcal{C}_2$ is communicating through $c \in \Delta_2^I$).

$\mathcal{C}_1\mathcal{C}_2$ is communicating through $c \in \Delta_1^I \times \Delta_2^I$ *(by Lemma 8)*

Case XI.V.III ($\mathcal{C}_2$ is communicating through $c \in \Delta_2^O$).

$\mathcal{C}_1\mathcal{C}_2$ is communicating through $c \in \Delta_1^O\Delta_2^O$ *(by Lemma 9)*

Case XI.V.IV ($\mathcal{C}_2$ is requesting resource from $r \in \Sigma_2^I$).

$\mathcal{C}_1\mathcal{C}_2$ is requesting resource from $r \in \Sigma_1^I \times \Sigma_2^I$ *(by Lemma 10)*

Case XI.V.V ($\mathcal{C}_2$ does not have proc objects (computation is over)).

$\mathcal{C}_1\mathcal{C}_2$ does not have proc objects (computation is over) *(by Definition 3)*

■

A.3 Theorems

A.3.1 Preservation

Theorem 1 (Preservation). *If* $\Sigma^I;\Delta^I \models \mathcal{C} :: \left(\Sigma^O;\Delta^O\right)$ *and* $\mathcal{C} \to \mathcal{D}$ *then* $\Sigma^I;\Delta^I \models \mathcal{D} :: \left(\Sigma^O;\Delta^O\right)$.

Demonstração. Induction on all cases of $\mathcal{C} \to \mathcal{D}$, type checking $\mathcal{C}$ and $\mathcal{D}$ and asserting that all the types are the same. We demonstrate some of the cases:

Case I (Main).

$$\begin{aligned}&\textbf{(Main)}\ \mathtt{Main}(P[\Sigma][\Delta][x]), \underline{\mathtt{env}}(T_c, T_s, 0, 0)\\ &\quad \to \mathtt{Closed}\left\{\mathtt{proc}\left(p \mid P \mid x\right), [\mathtt{idle}(r)]_{\forall r \in \Sigma}, \underline{\mathtt{env}}(T_c, T_s, 0, 0)\right\} \qquad \textit{(fresh p)}\end{aligned}$$

Before rule:

$$\dfrac{\dfrac{\overset{\mathcal{A}_1}{(\forall(((\sigma := P_\sigma) : R_\sigma) \in \Sigma).\mathtt{ext}(\mathtt{DEF}(P_\sigma), R_\sigma))} \quad \overset{\mathcal{A}_2}{\Sigma;\Delta \vdash^{0,0}_{T_s,T_c} P :: \left(x : A\right)}}{\Sigma;\Delta \vdash^{0,0}_{T_s,T_c} \mathtt{Main}(P) :: \left(x : A\right)}\,\mathtt{Main}}{-;\Delta \models \mathtt{Main}(P), \underline{\mathtt{env}}(T_s, T_c, 0, 0) :: (-;x : A)}\,\mathtt{Main}$$

After rule:

$$\dfrac{\dfrac{\dfrac{\overset{\mathcal{A}_2}{\Sigma;\Delta \vdash^{0,0}_{T_c,T_s} P :: \left(x : A\right)} \quad \overset{\mathcal{A}_3}{\mathtt{ext}((\Sigma;\Delta;A), R)}}{\Sigma;\Delta \models \mathtt{proc}\left(r \mid P \mid x\right), \underline{\mathtt{env}}(T_c, T_s, 0, 0) :: (r : R; x : A)}\,\mathtt{proc} \quad \dfrac{\overset{\mathcal{A}_1}{(\forall((\sigma : R_\sigma) \in \Sigma).\mathtt{ext}(\mathtt{DEF}(P_\sigma), R_\sigma))}}{-;- \models [\mathtt{idle}(\sigma := P_\sigma)]_{\forall \sigma \in \Sigma}, \underline{\mathtt{env}}(T_c, T_s, 0, 0) :: (\Sigma; -)}\,\mathtt{idle}}{\Sigma;\Delta \models \mathtt{proc}\left(r \mid P \mid x\right), [\mathtt{idle}(\sigma := P_\sigma)]_{\forall \sigma \in \Sigma}, \underline{\mathtt{env}}(T_c, T_s, 0, 0) :: (r : R, \Sigma; x : A)}\,\mathtt{Compose}}{-;\Delta \models \mathtt{Closed}\left\{\mathtt{proc}\left(r \mid P \mid x\right), [\mathtt{idle}(\sigma := P_\sigma)]_{\forall \sigma \in \Sigma_C}, \underline{\mathtt{env}}(T_c, T_s, 0, 0)\right\} :: (-; x : A)}\,\mathtt{Closed}$$

$\mathcal{A}_3$ holds because r is used only once, so $R = (\Sigma;\Delta;A)$ and $\mathtt{ext}((\Sigma;\Delta;A), (\Sigma;\Delta;A))$ *is trivial.*

Case II (id).

$$\textbf{(id)}\ \mathcal{C}, \mathtt{proc}\left(r \mid x \leftarrow y \mid x\right) \xrightarrow{\mathtt{Closed}} \mathcal{C}[y/x], \mathtt{idle}(r)$$

Before rule:

$$\dfrac{\overset{\mathcal{A}_1}{\Sigma_C, r:R;\Delta\Delta_C, x:A \vDash \mathcal{C}', \underline{\texttt{env}}(c,s,k,t) :: (\Sigma_C;\Delta_C, y:A', z:C)} \qquad \dfrac{\dfrac{\overset{\mathcal{A}_2}{\left([A']^{+T} = [A]^{+T} = B\right)} \quad \dfrac{}{-;y:B \vDash^{k,t}_{c,s} \left(x \leftarrow y\right) :: \left(x:B\right)}\,\texttt{id}}{-;y:[A']^{+T} \vDash^{k,t}_{c,s} \left(x \leftarrow y\right) :: \left(x:[A]^{+T}\right)}\,\texttt{eq} \quad \overset{\mathcal{A}_3}{\texttt{ext}((-;y:A';A),R)} \quad (T = sk+t)}{-;y:A' \vDash \texttt{proc}\left(r \mid x \leftarrow y \mid x\right), \underline{\texttt{env}}(c,s,k,t) :: (r:R;x:A)}\,\texttt{proc}}{\dfrac{\Sigma_C, r:R;\Delta\Delta_C, x:A, y:A' \vDash \mathcal{C}', \texttt{proc}\left(r \mid x \leftarrow y \mid x\right), \underline{\texttt{env}}(c,s,k,t) :: (\Sigma_C, r:R;\Delta_C, x:A, y:A'z:C)}{-;\Delta \vDash \texttt{Closed}\left\{\mathcal{C}', \texttt{proc}\left(r \mid x \leftarrow y \mid x\right), \underline{\texttt{env}}(c,s,k,t)\right\} :: (-;z:C)}\,\textit{Closed}}\,\textit{Compose}$$

After rule:

$$\dfrac{\dfrac{\dfrac{\overset{\mathcal{A}_1}{\Sigma_C, r:R;\Delta\Delta_C, x:A \vDash \mathcal{C}', \underline{\texttt{env}}(c,s,k,t) :: (\Sigma_C;\Delta_C, y:A', z:C)} \quad \overset{\mathcal{A}_2}{[A]^{+T} = [A']^{+T}} \quad (T = sk+t)}{\Sigma_C, r:R;\Delta\Delta_C, y:A' \vDash \mathcal{C}'[y/x], \underline{\texttt{env}}(c,s,k,t) :: (\Sigma_C;\Delta_C, y:A', z:C)}\,\texttt{Subst} \qquad \dfrac{\overset{\mathcal{A}_4}{\texttt{ext}(\texttt{DEF}(P_r),R))}}{-;- \vDash \texttt{idle}(r := P_r), \underline{\texttt{env}}(c,s,k,t) :: (r:R;-)}\,\texttt{idle}}{\Sigma_C, r:R;\Delta\Delta_C, y:A' \vDash \mathcal{C}'[y/x], \texttt{idle}(r), \underline{\texttt{env}}(c,s,k,t) :: (\Sigma_C, r:R;\Delta_C, y:A', z:C)}\,\textit{Compose}}{-;\Delta \vDash \texttt{Closed}\{\,\mathcal{C}'[y/x], \texttt{idle}(r), \underline{\texttt{env}}(c,s,k,t)\,\} :: (-;z:C)}\,\textit{Closed}$$

■

A.3.2 Global Progress

Theorem 3 (Global Progress). *If* $-;\Delta \vDash \texttt{Closed}\{\,\mathcal{C}\,\} :: (-;x:A)$*, then*

1. $\mathcal{C} \rightarrow \mathcal{D}$*, for some* $\mathcal{D}$*, or*
2. $\mathcal{C}$ *is communicating through* $c \in \Delta$ *or* x*, or*
3. $\mathcal{C}$ *does not have* `proc` *objects (computation is over).*

Demonstração.

$-;\Delta \vDash \texttt{Closed}\{\,\mathcal{C}\,\} :: (-;x:A)$ *(main assumption)*
$\Sigma_C;\Delta\Delta_C \vDash \mathcal{C} :: (\Sigma_C;\Delta_C, x:A)$ *(by inversion on (Closed))*
for some Σ_C *and* Δ_C

By Lemma 8, either

I. $\mathcal{C} \rightarrow \mathcal{D}$, for some $\mathcal{D}$, or

II. $\mathcal{C}$ is communicating through $c \in \Delta\Delta_C$, or

III. $\mathcal{C}$ is communicating through $c \in (\Delta_C, x:A)$, or

IV. $\mathcal{C}$ is requesting resource from $r \in \Sigma_C$, or

V. $\mathcal{C}$ does not have `proc` objects (computation is over).

We proceed proving global progress for each case:

Case I ($\mathcal{C} \to \mathcal{D}$, for some $\mathcal{D}$). *Case 1 applies.*

Case II ($\mathcal{C}$ is communicating through $c \in \Delta\Delta_C$). *By Lemma 2, $\mathcal{C}$ is either communicating through Δ or through Δ_C. We proceed analysing both subcases:*

Case II.I ($\mathcal{C}$ is communicating through $c \in \Delta$). *Case 2 applies.*

Case II.II ($\mathcal{C}$ is communicating through $c \in \Delta_C$). *By Lemma 7, either $\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{D}$, for some $\mathcal{D}$, (Case 1 applies) or $\mathcal{C}$ is communicating through Δ or x (Case 2 applies).*

Case III ($\mathcal{C}$ is communicating through $c \in (\Delta_C, x : A)$). *By Definition 3, of contexts, $\mathcal{C}$ is either communicating through Δ_C or through x. We proceed analysing each subcase:*

Case III.I ($\mathcal{C}$ is communicating through $c \in \Delta_C$). *Same as in Case II.II*

Case III.II ($\mathcal{C}$ is communicating through x). *Case 2 applies.*

Case IV ($\mathcal{C}$ is requesting resource from $r \in \Sigma_C$). *By Lemma 6, $\mathcal{C} \xrightarrow{\texttt{Closed}} \mathcal{D}$, for some $\mathcal{D}$, (Case 1 applies).*

Case V ($\mathcal{C}$ does not have `proc` objects (computation is over)). *Case 3 applies.*

■