Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor Thesis

Optimal Interaction with the Real World

Marvin Borner

August 01, 2025

Reviewer

Jun.-Prof. Dr. Jonathan Immanuel Brachthäuser
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Marvin Borner:
6082310
Optimal Interaction with the Real World
Bachelor Thesis
Eberhard Karls Universität Tübingen
Thesis period: 04/01/2025–08/01/2025

Abstract

When evaluating functional programs, there are various strategies for reducing expressions. A strategy is Lévy-optimal if it never does redundant work such as reducing terms that are later erased, while also sharing reduction work maximally. To implement this efficiently, graphical representations like interaction nets can be used to encode the sharing of duplicated terms. Beyond optimality, interaction nets provide elegant solutions for parallelization and distributed computing, making them an attractive backend for programming languages. The communication between interaction nets and the real world is rarely discussed, despite the necessity for side effects such as IO in real-world applications. In this thesis, we will therefore analyze existing implementations, and formalize and implement an optimal runtime with side effects.
Since side effects are inherently sequential and interaction nets are not, the main challenge lies in embedding such sequentialization in an otherwise non-sequential system. Our approach introduces a novel formalization using a token-based understanding of execution. We further present two solutions for enforcing a strict sequence upon this system using a custom token-passing semantics. Finally, we argue about the feasibility of our approach by providing initial practical results.

Acknowledgements

First of all, I would like to thank Jonathan Brachthäuser and Philipp Schuster for suggesting a topic that genuinely interests me, allowing me to work on it for my thesis, and supporting me greatly.
Similarly, I want to thank my parents for allowing me to pursue whichever goals and interests I develop, and continually supporting me through them all. I am very grateful for this privilege.
I also believe my studies and research would have gone an entirely different way if not for my friends, who showed me the many benefits of collaboration and social interaction with the real world.
I am grateful to the HVM and Vine communities for the interesting discussions.
Lastly, I am thankful for helpful reviews and support from my family and friends.
Special thanks to Timber, René, Patricia, and Lars ♥

Contents

Introduction **3**

1 Foundations **9**
1.1 Pure Lambda Calculus 9
1.2 Interaction Nets 12
1.2.1 Interaction Calculus 13
1.2.2 Symmetric Interaction Combinators 14
1.3 Lambda Calculus and Interaction Nets 17
1.3.1 Polarization 17
1.3.2 Lambda Calculus by Polarization 18
1.3.3 Translation 21
1.3.4 Read-Back 22
1.3.5 Optimality 22
1.3.6 Bookkeeping Oracle 23
1.4 Side Effects 24

2 Related Work **27**
2.1 Higher-Order Virtual Machine 27
2.2 Ivy 29
2.3 More Interaction Nets 31
2.4 Concurrent Clean 33

3 Optimal Effects **35**
3.1 Related Work 35
3.2 Effectful Lambda Calculus 36
3.3 Problem of Order 39
3.4 Asynchronous Actions 41
3.5 Interpretations 42
3.6 Recursion 43
3.7 Effectful Optimality 44

4 A Simple Language **47**
4.1 The Language . . . 47
4.2 Translations . . . 49
4.3 Recursion . . . 52
4.4 Applications of Forks . . . 52
4.5 Foreign Functions . . . 54

5 Effectful Agents **57**
5.1 Introduction . . . 57
5.2 Agents . . . 58
5.3 Executing Actors . . . 60
5.4 Partial Application . . . 61
5.5 Forks . . . 62
5.5.1 Communication . . . 65
5.6 Translation . . . 66
5.7 Evaluation . . . 67
5.7.1 Approach . . . 67
5.7.2 Results . . . 68
5.8 Discussion . . . 70

6 Monadic Style **71**
6.1 Monads . . . 71
6.2 Language Extension . . . 72
6.3 Semantics . . . 73
6.4 Monadic Agents . . . 76
6.5 Translations . . . 77
6.6 Recursion . . . 78
6.7 Evaluation . . . 79
6.8 Discussion . . . 81

7 Direct Style **83**
7.1 Introduction . . . 83
7.2 Language Extension . . . 84
7.3 World-State Passing . . . 85
7.4 Token Passing . . . 87
7.4.1 Semantics . . . 87
7.4.2 Redirector Agents . . . 90
7.4.3 Token-Passing Overhead . . . 94
7.4.4 Inference of Action Potential . . . 94
7.4.5 Recursion . . . 98
7.4.6 Redirectors Generalize Monads . . . 98

7.5 Evaluation 100
7.5.1 Comparison 100
7.5.2 Join 102
7.6 Discussion 103

8 Conclusion **105**
8.1 Summary 105
8.2 Future Work 107
8.3 Conclusion 109

Bibliography **111**

A Implementation & Usage **117**
A.1 Usage 117
A.2 User Interface 118
A.3 Structure 119
A.4 Tests & Samples 121
A.5 Differences 121

Introduction

We start with a language that many students are introduced to in primary school: *Elementary arithmetic*. It consists only of an infinite set of symbols (numbers), a few infix operators (addition, subtraction, multiplication, division) and their reduction rules, a simple syntax on how to combine them arbitrarily, as well as syntactic precedences on certain operators (such as multiplying before adding).

Yet, writing and composing terms in this simple language allows us to describe all kinds of complex relationships. The emerging complexity can also be seen in the many decisions that have to be made when we want to evaluate the term efficiently. For example, say we want to calculate

$$0 \cdot 1 \cdot (2 + 3) \cdot 4 \cdot 5 - 4 \cdot 5.$$

We can observe how this calculation can be evaluated—*reduced*—using different strategies: Reduction could start in the parenthesized term, from left to right, from right to left, or in random order.

In this case, using a strategy in which the reduction of the parenthesized term is *delayed* as long as possible will require fewer steps than when it is reduced at the start, as $0 \cdot (2 + 3) = 0$ can be reduced in a single step. It can also be seen that the strategy with the *minimum number* of reduction steps is the strategy which generally reduces multiplications with 0 eagerly before inspecting or reducing its other argument. Additionally, the multiplication $(4 \cdot 5)$ may be *shared* in such a way that it is only calculated once and then substituted into both the left and the right terms of the subtraction—thus preventing redundant calculations. Finally, this specific term could be reduced *in parallel*, as $(2 + 3)$ and $(4 \cdot 5)$ do not interfere and may be calculated at the same time. However, this parallelism must happen in cooperation with the mechanisms required for sharing, as $(4 \cdot 5)$ could otherwise still be calculated twice.

Minimal modifications to the syntax, symbols, and reduction rules of this calculus will yield a kind of programming language. The calculus that we focus on in this thesis is called the *λ-calculus*. The λ-calculus forms the basis of many modern programming languages. Its minimalism comes with a cost not dissimilar to the challenge of reducing elementary arithmetic efficiently: Reducing terms in

parallel, sharing equivalent terms, choosing the reduction strategy, as well as delaying the reduction of certain terms until they are needed are all problems that have been exhaustively studied in the λ-calculus.

Of course, the effects of improving upon these challenges are much larger in the case of the λ-calculus than in the basic arithmetic that we have shown before. Being the backbone of millions of programs, languages making use of the parallel nature of our hardware or requiring less reduction work in the first place, has tremendous implications on their overall efficiency and energy consumption.

Interaction nets as introduced by Lafont [1] provide an elegant solution to the mentioned problems. They consist of agents (nodes) connected via wires (edges). Being a graph-based encoding, the support for sharing terms is inherent in its design. Because of properties called locality and strong confluence, interaction nets are particularly useful for reducing terms in parallel and with arbitrary reduction strategies.

Furthermore, interaction nets can provide the *optimality* property [2]. Reducing languages optimally *guarantees* to never do work redundantly, therefore requiring the minimum number of reduction steps. No unneeded terms are ever reduced and reduction of terms is always shared as much as possible [2, p. 146].

Unfortunately, optimality can come with a certain overhead, which is why minimizing the reduction work does, importantly, not imply that the term will be reduced in the minimal *time* in practice [3]. Additionally, it is an unsolved problem to reduce optimally while at the same time striving for maximal parallelism [2, p. 148].

This becomes more complex when recognizing the following fact: Actual programming languages will have to support side effects. Interacting with the user, reading files, or requesting data from the internet are required for a language to be useful. As an implication of the physical nature, interacting with the environment typically happens in a strict sequence indexed by time. Embedding side effects in pure languages such as the λ-calculus therefore quickly gives rise to questions regarding the order in which terms with side effects are to be reduced.

Take for example an extension of elementary arithmetic that can read integers from a user input, potentially yielding different results depending on the order in which the `readInt`s are executed:

$$(\texttt{readInt()} + 4) \cdot (\texttt{readInt()} + 2)$$

We then state the following question: *How can side effects be executed in strict sequentiality while the system itself is supposed to be inherently non-sequential? How does this interact with the challenge of minimizing redundancy while maximizing sharing?*

Existing solutions of forcing sequential execution either limit the possibility for parallelism and sharing to certain cases, or eliminate it completely. On the

other hand, implementations based on interaction nets either do not support side effects at all, only to a limited extent, or support side effects only after the rest of the term has been fully reduced.

Separating side effects from the rest of the reduction completely is problematic: There are indeed cases where parallel or non-sequential execution of effectful terms is *desired*, since not all side effects affect or depend on each other. Reading from different files, for example, does not normally require a strict sequential order and could also be done in parallel.

```
let fileA = readFile "a.txt";
let fileB = readFile "b.txt";
let fileC = readFile "c.txt";
print (fileA + fileB + fileC)
```

Side effects are traditionally not added to optimal languages and seemingly eliminate many of their desired properties while adding a lot of complexity. Only recently with the introduction of the Higher-Order Virtual Machine (HVM) [4] have languages based on interaction nets started targeting a broader audience and therefore started exploring implementations of side effects.

Contributions

We address the problem of embedding sequential execution into the otherwise non-sequential system of interaction nets, while focussing on the optimality property. Concretely, our contributions consist of four main parts:

Optimal Effects We develop a minimal effectful λ-calculus and describe a translation to effectful agents of interaction nets that can be reduced optimally. We provide a token-based understanding of executing effectful terms sequentially. To our knowledge, we present the first formalization of side effects and their interactions in the context of interaction nets.

Implementation We design and implement minimal languages extending the introduced effectful λ-calculus and interaction net encodings with different methods of sequentializing the execution of side effects. Notably, we embed *monadic*- and *direct*-style semantics into the effectful λ-calculus, the introduced language, and interaction nets.

Survey We discuss existing implementations of side effects in languages based on interaction nets. We further relate our work to research from different domains with similar goals.

State of the Art We build upon the current state of interaction net research by using symmetric combinators and polarity types. We improve upon this state of the art by showing how polarization of agents in a way *prescribes* the execution order of effectful terms.

Language Overview

In the course of this thesis we define several languages in order to present our approach and argue about its feasibility:

- $\mathcal{L}_{\text{sim}}$: A simple programming language that we use as a common base.
- $\mathcal{L}_{\text{dir}}$: An extension of $\mathcal{L}_{\text{sim}}$ with support for direct-style semantics.
- $\mathcal{L}_{\text{mon}}$: An extension of $\mathcal{L}_{\text{sim}}$ with support for monadic-style semantics.
- Λ: The λ-calculus in its purest form.
- Λ_{eff}: An extension of Λ with support for side effects.
- Λ_{mon}: An extension of Λ_{eff} with support for monadic-style semantics.
- Σ: The base of interaction nets.
- Σ_{eff}: An extension of Σ with support for side effects.
- Σ_{mon}: An extension of Σ_{eff} with support for monadic-style semantics.
- Σ_{pol}: The polarized base of interaction nets.
- Σ_{peff}: An extension of Σ_{pol} with support for side effects.
- Σ_{pdir}: An extension of Σ_{peff} with support for direct-style semantics.
- Σ_{pmon}: An extension of Σ_{peff} with support for monadic-style semantics.

The relation between these languages is laid out in figure 1.

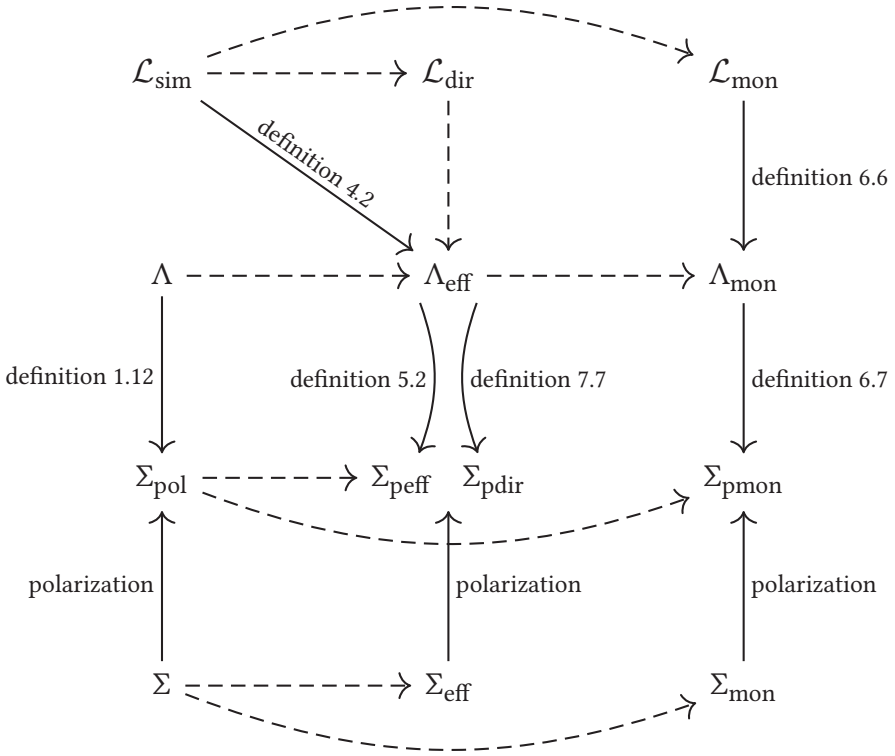


Figure 1 The languages described in this thesis and their relations together with the definitions in which they are described. Each arrow indicates the existence of a translation from one language to another.

Chapter 1

Foundations

This chapter provides the foundations for understanding the rest of our work. In section 1.1 we define the λ-calculus—the base of this thesis. We then go on to discuss interaction nets and their relations to the λ-calculus. Finally, we introduce our definition of the term "side effects".

1.1 Pure λ-Calculus

The λ-calculus forms the foundation of well-known functional programming languages such as Haskell, OCaml, or F#.

We recall relevant definitions of Λ based on the writing of Barendregt [5].

Definition 1.1 (Terms of the Pure λ-Calculus)**.** The terms of the pure λ-calculus Λ are:

$$
\begin{array}{rll}
t & ::= (\lambda x.t) & \textsc{Abstraction} \\
& \mid (t\ t) & \textsc{Application} \\
& \mid x & \textsc{Binding} \\
x & \in a, b, abc, \ldots & \textsc{Symbol}
\end{array}
$$

Applications default to left associativity, allowing the shorthand $(t_1\ t_2\ t_3\ \ldots\ t_n)$ for $((\ldots((t_1\ t_2)\ t_3)\ \ldots)\ t_n)$. Nested abstractions can be contracted: $(\lambda a.(\lambda b.M))$ can be written equivalently as $(\lambda ab.M)$. We omit outermost and redundant parentheses, and abstractions have a higher precedence than applications.

Terms can then be inductively constructed using these rules. Notably, different symbols can have the same meaning. For example, a term $\lambda xy.xy$ is equivalent to $\lambda ab.ab$ in every way that matters for reduction (α*-equivalence*).

A binding *binds* a symbol to a single ABSTRACTION with the same name. Bindings are resolved by an application to an ABSTRACTION. The process of such an application is called *β-reduction*.

Definition 1.2 (*β*-Reduction)**.** Let $M, N \in t$ be arbitrary terms of the pure λ-calculus. An APPLICATION $(\lambda x.M)\ N$ is called a *β-redex*. A β-redex β-reduces with the following rewrite rule:

$$(\lambda x.M)\ N \overset{\beta}{\rightsquigarrow} M\{x \mapsto N\},$$

where any symbol x within M is substituted by the term N.

With β-reduction, ABSTRACTIONS resemble *functions* in functional programming languages.

Example 1.3. Let $M = (\lambda ab.a)\ \lambda x.x$ be a β-redex that applies the identity function $\lambda x.x$ to a term receiving two consecutive arguments and only keeping the first one. Then, M β-reduces to:

$$M \overset{\beta}{\rightsquigarrow} \lambda bx.x$$

Symbols not bound by an ABSTRACTION are *free* and can not be substituted.

Definition 1.4 (Free Symbols)**.** The set of free symbols $\mathsf{fv}(M)$ is constructed inductively for a given term M:

$$\begin{aligned} \mathsf{fv}(\lambda x.M) &= \mathsf{fv}(M) \setminus \{x\} \\ \mathsf{fv}(M\ N) &= \mathsf{fv}(M) \cup \mathsf{fv}(N) \\ \mathsf{fv}(x) &= \{x\} \end{aligned}$$

If a term M has $\mathsf{fv}(M) = \emptyset$, it is *closed*. The number of free symbols of name x in a term M is written as $\#_x(M)$.

β-reduction has implicit functionality that is relevant when discussing efficient implementations and reduction techniques:

Duplication $(\lambda x.M)\ N$ with $\#_x(M) = n > 1$ *duplicates* N n times.

Erasure $(\lambda x.M)\ N$ with $\#_x(M) = 0$ *erases* N.

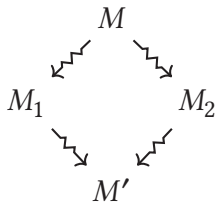
In the aspect of efficiency and *sharing* it would make sense to reduce terms before substituting when $\#_x(M) > 1$, and not reducing terms at all when $\#_x(M) = 0$. However, having $\#_x(M) > 1$ does not mean that every term substituted into x will eventually be used, as they could also get erased. We discuss this problem of optimization under the name of *optimality* in section 1.3.5.

The reflexive, transitive closure of β-reduction is written as $\overset{\beta}{\rightsquigarrow\!\!\!\!\rightsquigarrow}$ [6], such that $\overset{\beta}{\rightsquigarrow\!\!\!\!\rightsquigarrow}$ is the smallest relation that satisfies:

Reflexivity Every term can reduce to itself: $M \overset{\beta}{\twoheadrightarrow} M$.

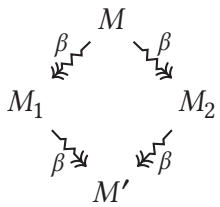
Transitivity Reductions can be chained: If $M \overset{\beta}{\twoheadrightarrow} N$ and $N \overset{\beta}{\twoheadrightarrow} P$, then $M \overset{\beta}{\twoheadrightarrow} P$.

Definition 1.5 (Diamond Property)**.** A reduction relation $\rightsquigarrow$ satisfies the *diamond property* for a language $\mathcal{L}$, if for $M, M_1, M_2 \in \mathcal{L}$ and $M_1 \neq M_2$, $M \rightsquigarrow M_1$ and $M \rightsquigarrow M_2$, there exists $M' \in \mathcal{L}$, such that $M_1 \rightsquigarrow M'$ and $M_2 \rightsquigarrow M'$:



The λ-calculus satisfies this property only for the reflexive, transitive closure of $\overset{\beta}{\rightsquigarrow}$, not $\overset{\beta}{\rightsquigarrow}$ itself [5, p. 61].

Theorem 1.6 (Church-Rosser, Barendregt [5])**.** *β-reduction is* Church-Rosser*: $\overset{\beta}{\twoheadrightarrow}$ satisfies the diamond property of Λ.*



When a term does not contain further β-redexes, the term has reached the *β-normal form*. We write $M \downarrow M'$, if M reduces to M' and M' is a normal form.

As a direct implication of the Church-Rosser property, the order in which β-redexes are reduced does not influence the β-normal form: Every term has either a single β-normal form, or reduces indefinitely. Therefore, if $M \downarrow M'$ and $M \downarrow M''$, then $M' = M''$.

Notably, since $\overset{\beta}{\twoheadrightarrow}$—not $\overset{\beta}{\rightsquigarrow}$—satisfies the diamond property, there can exist multiple reduction paths with different length. For example in the diagram above, the reduction $M_1 \overset{\beta}{\twoheadrightarrow} M'$ could use n one-step β-reductions, while $M_2 \overset{\beta}{\twoheadrightarrow} M'$ may take n, fewer, more, or none at all. This is true analogously for $M \overset{\beta}{\twoheadrightarrow} M_1$ and $M \overset{\beta}{\twoheadrightarrow} M_2$.

1.2 Interaction Nets

Interaction nets were introduced by Lafont [1] as a graphical model of computation with inherent parallelism and determinism.

Interaction nets consist of *agents* a with a *principal port* and $\mathrm{ar}(a) = n$ *auxiliary ports*. Ports of agents are connected by *wires*. Any number of interconnected agents can be abstracted into a "black box" with free wires representing the unconnected ports of the agents it contains.

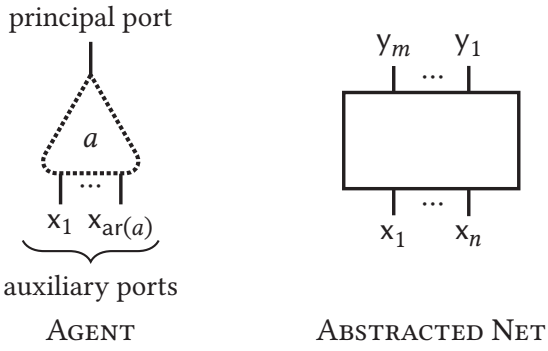


Figure 1.1 Interaction nets consist of interconnected agents, which can be abstracted into boxes.

Interaction *rules* specify the behavior of two agents connected by their principal ports. Agents connected by their principal ports form an *active pair* and can then *interact*. The result is a net with an equivalent set of free wires. There only exists *at most* one interaction rule for any given (unordered) active pair ("no ambiguity" [1]). If no rule exists, the agents do not interact. Once no further rule can be applied to any agents in a net, it is in normal form.

In our visualizations, we highlight active pairs using a red wire, as in figure 1.2.

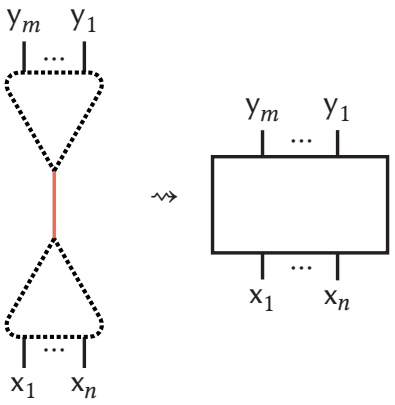


Figure 1.2 Interaction of two agents forming an active pair.

We write an interaction between two agents as the one-step reduction relation $\rightsquigarrow$. It has been noted by Lafont [1] that this relation is purely *local*: Interaction rules only ever apply to two agents and cannot interfere with other interactions.

This property comes in addition to *strong confluence*[1].

Theorem 1.7 (Strong Confluence, Lafont [1])**.** *The reduction relation $\rightsquigarrow$ of interaction nets satisfies the diamond property.*

Strong confluence and locality have several notable implications on the reduction behavior: The order in which interaction rules are applied has (1) no influence on the result of the reduction, and (2) no influence on the total number of interactions required until a normal form is reached (if existing). Therefore, "if one abstracts from the irrelevant order of application of rules, there is only one possible reduction" [7] from an interaction net to its normal form.

1.2.1 Interaction Calculus

In order to argue about interaction nets in a formal way, we introduce two notations: Lafont's notation [1] for describing an interaction rule formally, and the interaction calculus [8] for describing the structure of a single net.

We recall the definitions of Fernández; Mackie [8]:

Names $\mathcal{N}$ is the set of *names* ranged over by $x, x_1, y, \ldots$. Names are the formalization of ports of agents.

Agents $\mathcal{A}$ is the set of *agents* ranged over by $\alpha, \beta, \gamma, \triangleleft, \triangleright, \ldots$.

[1] Historically, the term "strong confluence" is overloaded. We use the definition by Lafont [1], different from the one by Huet [6].

Terms A *term* is constructed by $t ::= x \mid a(t_1, \dots, t_n)$, where $x \in \mathcal{N}$ and $a \in \mathcal{A}$ with $\mathrm{ar}(a) = n$. t_i are terms such that each name only appears at most twice, thus corresponding to wires between two equivalently named ports.

Equations Given terms t and u, $t == u$ is an *equation*. Multisets of equations are written as $\Delta, \Theta, \dots$. Such equations correspond to wires between named ports or the principal ports of terms.

Configurations In a *configuration* $\langle t_1, \dots, t_n \mid \Delta \rangle$, the terms t_i represent the *interface* of the encoded net—i. e. the wires open for connection.

Interaction Rules Two agents $a, b \in \mathcal{A}$ form an active pair in $a[t_1, \dots, t_n] \bowtie b[u_1, \dots, u_m]$. t_i corresponds to the resulting term at the ith auxiliary port of a, and equivalently for u_i and b. We refer to it as *Lafont's notation*, as it was presented first by Lafont [1]. As in interaction nets, these rules are symmetric and unique for a pair of agents.

We do not discuss the reduction rules of the presented calculus further and instead refer to Fernández; Mackie [8] for more details.

1.2.2 Symmetric Interaction Combinators

> The fundamental laws of computation are *commutation* and *annihilation*.
>
> *Lafont (1997)*

Lafont [7] introduced the interaction combinators as a minimal, yet complete subset of interaction nets that suggests its laws may be the fundamental laws of computation itself. The system consists of three agents: The CONSTRUCTOR, DUPLICATOR, and ERASER. The interaction combinators are universal: "Any interaction system can be translated into the system of interaction combinators." [7, Theorem 1].

Systems using interaction combinators are easier to implement and optimize than general interaction nets since they have a fixed, small set of agents. The *symmetric* interaction combinators were introduced in the PhD thesis of Mazza [9] and provide a further simplification of the interaction combinators: They require only one instead of two rules for the "annihilation" of constructors and duplicators.

We extend the system minimally with an INITIATOR[2] that represents the *root* of an interaction net. This agent allows for interaction specifically with the root

of a term and makes translated programs not have a "dangling" wire that would otherwise not connect to anything at all. [10]

Definition 1.8 (Agents of Σ)**.** We define as $\Sigma \subset \mathcal{A}$ the set of agents of the symmetric interaction combinators with the initiator.

$$\Sigma = \{\zeta_\circ, \iota_\circ, \delta_\circ, \varepsilon_\circ\},$$

with $\mathrm{ar}(\zeta_\circ) = \mathrm{ar}(\delta_\circ) = 2$ and $\mathrm{ar}(\iota_\circ) = \mathrm{ar}(\varepsilon_\circ) = 0$.

Figure 1.3 The symmetric interaction combinators plus initiator. From left to right: $\zeta_\circ, \iota_\circ, \delta_\circ, \varepsilon_\circ$.

The interaction rules of Σ can be categorized into *annihilation* and *commutation*. Annihilations describe the interactions between equivalent agents, and commutations describe interactions between two agents that are not equivalent. [9]

When a non-duplicating agent a interacts with a DUPLICATOR $\delta_\circ$, the agent is duplicated on both of $\delta_\circ$'s auxiliary ports, while the DUPLICATOR gets *propagated* through the $\mathrm{ar}(a) = n$ ports of a. In figure 1.4, x_1 will be connected to a duplicate of the agent $a(\mathsf{y}_1, \ldots, \mathsf{y}_n)$. In order for a net connected to a DUPLICATOR eventually being iteratively duplicated completely, the DUPLICATORS are correspondingly connected to the auxiliary ports of the duplications of a.

Unintuitively, as a direct implication of theorem 1.7, the number of duplications during reduction, as any other interaction, is constant and cannot be changed by prioritizing the reduction of other agents such as ERASERS.

When an ERASER interacts with an agent a, it removes the agent and continues at its auxiliary ports. Via DUPLICATOR and ERASER, commutations allow a form of *iterative* memory management which happens independently of the reductions of the term being duplicated or erased.

[2]The naming comes from the reference implementation [11] of Lambdascope [12], although its use is not mentioned in the published work itself.

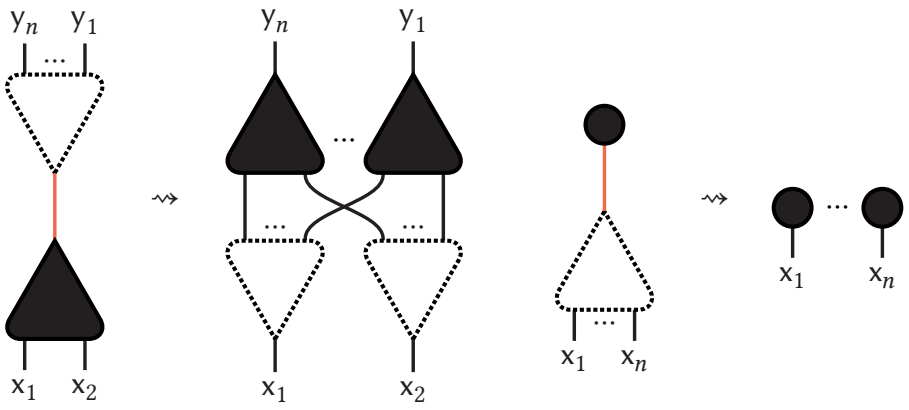


Figure 1.4 Commutations in Σ, where △ ≠ ▲ is an arbitrary non-duplicating agent.

The other case of interactions are the annihilations in figure 1.5. When two CONSTRUCTORS or DUPLICATORS interact, they will both get erased and their ports get connected symmetrically (x_1 to y_1, x_2 to y_2). When two ERASERS interact they will also both get erased, but with no wires remaining.

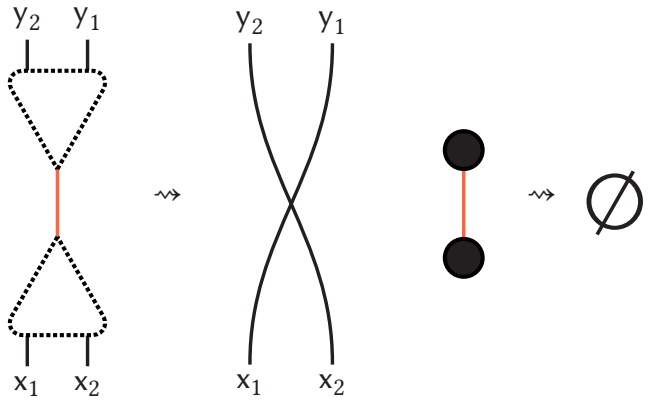


Figure 1.5 Annihilations in Σ, where △ ∈ {△, ▲} and ∅ is the empty set.

Definition 1.9 (Interaction rules of Σ)**.** The commutation and annihilation rules of Σ are formalized using Lafont's notation:

$$\begin{aligned}
\delta_\circ[a(x_1,\dots,x_n), a(y_1,\dots,y_n)] &\bowtie a[\delta_\circ(x_1,y_1),\dots,\delta_\circ(x_n,y_n)] \\
\varepsilon_\circ &\bowtie a[\varepsilon_\circ,\dots,\varepsilon_\circ] \\
\delta_\circ[x,y] &\bowtie \delta_\circ[x,y] \\
\zeta_\circ[x,y] &\bowtie \zeta_\circ[x,y] \\
\varepsilon_\circ &\bowtie \varepsilon_\circ
\end{aligned}$$

In following definitions we do not define the interactions with δ and ε explicitly, and assume them to always be applicable.

1.3 λ-Calculus and Interaction Nets

In section 1.1 we have shown $\overset{\beta}{\twoheadrightarrow}$ as having the diamond property, but not $\overset{\beta}{\to}$. One of the motivations for translating the λ-calculus to interaction nets is that it can then indirectly provide the property of *strong confluence*. With strong confluence, every reduction path has the same length.

Furthermore, encoding the λ-calculus as interaction nets can provide the *optimality* property. This property by itself *guarantees* that the reduction will not duplicate reducible terms, i. e. do redundant work, therefore minimizing the number of β-reductions. We describe this property further in section 1.3.5.

The expression of λ-terms in interaction nets has been achieved using several techniques: Lamping [13] described an encoding of λ-terms using agents similar to interaction agents via an extension of a graph-based encoding by Staples [14]. Lamping's graphs were formalized and described in the context of interaction nets by Gonthier; Abadi; Lévy [10]. The interaction combinators by Lafont [7] described in detail the mechanisms involved in duplicating and erasing agents—an essential part in reducing the λ-calculus. Further developments [15, 16, 2, 17, 12] differ mostly in general optimizations and the implementation of the *bookkeeping oracle* (see section 1.3.6).

The presented encoding uses a polarized variant of the symmetric interaction combinators, Σ_{pol}.

1.3.1 Polarization

The concept of polarization was introduced by Lafont [1]. There, they assign ports of interaction agents to *polarities* $p \in \{\oplus, \ominus\}$ in addition to type labels.

A simpler type system is the one used by Fernández; Mackie [18], where polarity itself is the only type. In an abstract sense, a positive polarity $\oplus$ means that the port produces something during interaction. Similarly, a negative polarity $\ominus$ means that the port consumes something during interaction.

Ports can only be connected to ports of opposing polarity. Polarized interaction rules must preserve polarization. A polarized network that follows these rules is called *well-typed* [18]. If an unpolarized network can be well-typed by polarization, the resulting network encodes a specific flow of information; Lafont [7] described polarized combinators as *directed combinators*.

We draw wires between polarized ports with an arrow to indicate the flow of the net. In fact, we will discuss in later chapters how it prescribes exactly the

sequential execution of effectful terms. Arrows point from negative to positive, as in figure 1.6.

Figure 1.6 Directed wire between two polarized ports.

Polarization allows *overloading* agents to different interpretations. Where, say, an ABSTRACTION produces a new term and a binding while consuming an argument, an APPLICATION consumes two terms and produces the result of their application. Both of these interpretations can be encoded as opposing polarization of the same agent. Importantly, the overloaded, polarized agents do not modify the rules of the unpolarized agents, but inherit their behavior.

We write a polarized variant a of an agent $a_\circ$ with $\mathrm{ar}(a_\circ) = n$ as the terms $a \equiv a_\circ(\mathsf{x}_{1,p_1}, \dots, \mathsf{x}_{n,p_1})^{p_0}$, where $p_{1,\dots n} \in \{\ominus, \oplus\}$ are the polarities of the auxiliary ports, and $p_0 \in \{\ominus, \oplus\}$ denotes the polarity of the principal port.

1.3.2 λ-Calculus by Polarization

The λ-calculus emerges from viewing the symmetric interaction combinators as polarized agents representing individual λ-term constructors. This follows from two observations: Traditional interaction systems encoding the λ-calculus are polarized [19], and: "Any polarized INS [interaction system] can be translated into symmetric combinators." [9, theorem 1.14]

We use the port layout used in HVM by Taelin [4].

Definition 1.10 (Agents of Σ_{pol})**.** The system of $\Sigma_{\mathrm{pol}} \subset \mathcal{A}$ extends Σ from definition 1.8 by polarization and consists of eight agents:

$$\Sigma_{\mathrm{pol}} = \{\alpha, \lambda, \delta, \sigma, \iota, \kappa, \varepsilon, \nu\},$$

with $\mathrm{ar}(\alpha) = \mathrm{ar}(\lambda) = \mathrm{ar}(\delta) = \mathrm{ar}(\sigma) = 2$ and $\mathrm{ar}(\iota) = \mathrm{ar}(\kappa) = \mathrm{ar}(\varepsilon) = \mathrm{ar}(\nu) = 0$.

The binary agents of Σ_{pol} polarize Σ in the following way: $\alpha \equiv \zeta_\circ(\mathsf{N}_\ominus, \mathsf{ret}_\oplus)^\ominus$, $\lambda \equiv \zeta_\circ(\mathsf{x}_\oplus, \mathsf{M}_\ominus)^\oplus$, $\delta \equiv \delta_\circ(\mathsf{M}_{1,\oplus}, \mathsf{M}_{2,\oplus})^\ominus$, and $\sigma \equiv \sigma_\circ(\mathsf{M}_\ominus, \mathsf{N}_\ominus)^\ominus$. The nilary agents: $\iota \equiv \iota_\circ^\ominus$, $\kappa \equiv \iota_\circ^\oplus$, $\varepsilon \equiv \varepsilon_\circ^\ominus$, and $\nu \equiv \varepsilon_\circ^\oplus$.

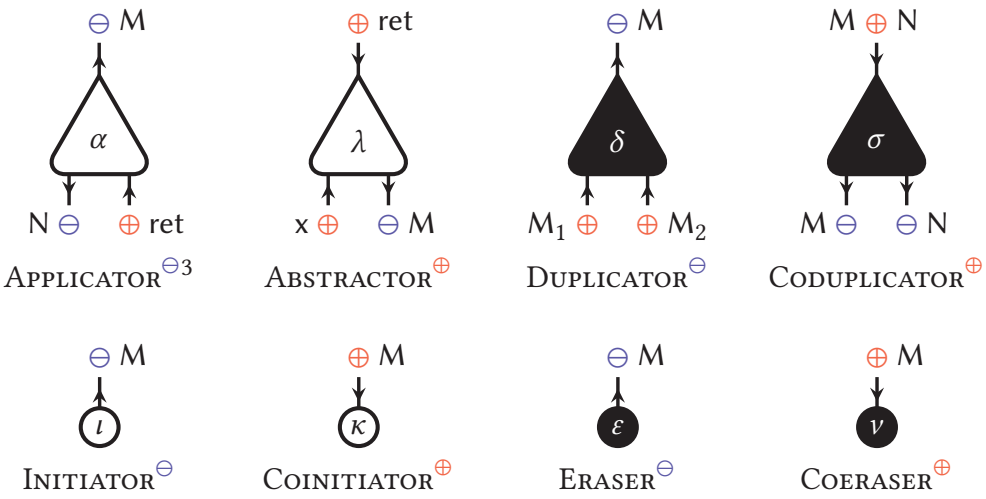


Figure 1.7 Polarized agents of Σ_{pol}.

We can now describe the polarized agents of Σ_{pol} separately:

Applicator$^{\ominus}$ A constructor agent encoding the application of λ-terms $M\ N$. It consumes two terms M and N, and produces the result of the application. Its principal port is on M, as M may be an abstraction with which an application should interact (β-reduction)

Abstractor$^{\oplus}$ A constructor agent encoding the abstraction of a λ-term $\lambda x.M$. It consumes the term in its body, and produces a binding x and the result of the abstraction (i. e. M after x has been bound to another term). Its principal port is on the ret-port, as in an interaction with an applicator$^{\ominus}$, x and M should be connected to N and ret of the applicator$^{\ominus}$, respectively.

Duplicator$^{\ominus}$ A duplicator agent encoding the implicit cloning of λ-terms explicitly, e. g. in $(\lambda x.x\ x)\ \lambda y.M$. It consumes a term, and produces two of its copies. Its principal port is on M, as it is connected to the term that should be duplicated in an interaction.

Coduplicator$^{\oplus}$ A duplicator agent capable of duplicating terms encoded as agents with a negative principal port—thus it coconsumes a term, and coproduces two of its copies.

Initiator$^{\ominus}$ An initiator agent that marks the root of the term M. It consumes the body of the term. Our translation and encoding will only ever contain one.

[3]We identify the polarized agents based on the polarity of their principal ports.

Coinitiator$^{\oplus}$ An initiator agent that is dual to the initiator$^{\ominus}$. It does not normally occur in well-formed nets.

Eraser$^{\ominus}$ An eraser agent encoding the implicit erasing of λ-terms explicitly, e.g. in $\lambda xy.y\ \lambda z.M$. It erases producing agents.

Coeraser$^{\oplus}$ An eraser agent that is dual to the eraser$^{\ominus}$. It acts equivalently, just that it erases consuming agents.

Through propagation of the agents $\delta, \sigma, \varepsilon, \nu$ responsible for memory management, interaction with the polarized constructors always creates two dual versions of the agent. Furthermore, the duplication or erasure of an entire encoded term necessarily reduces the term as a side effect, as the respective agents only interact with their principal ports [15]. The polarized β-reduction visualized in figure 1.8 must therefore happen before further terms can be duplicated or erased.

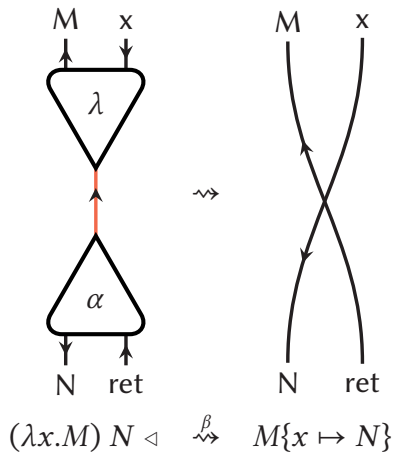


Figure 1.8 β-reduction in Σ_{pol}.

Definition 1.11 (Interaction rules of Σ_{pol})**.** The interaction rules of Σ_{pol} are implied by the rules of Σ described in definition 1.9:

$$\delta[\lambda(x_1, x_2), \lambda(y_1, y_2)] \bowtie \lambda[\sigma(x_1, y_1), \delta(x_2, y_2)]$$
$$\sigma[\alpha(x_1, x_2), \alpha(y_1, y_2)] \bowtie \alpha[\delta(x_1, y_1), \sigma(x_2, y_2)]$$
$$\lambda[x, y] \bowtie \alpha[x, y]$$
$$\varepsilon \bowtie \lambda[\nu, \varepsilon]$$
$$\nu \bowtie \alpha[\varepsilon, \nu]$$
$$\nu \bowtie \varepsilon$$

1.3.3 Translation

Terms of the λ-calculus can be inductively translated to Σ_{pol}. We base this procedure on the visualizations for the unpolarized combinators with oracle provided by van Oostrom; van de Looij; Zwitserlood [12] as well as Asperti; Guerrini [2].

We introduce the *multiplexer*. A MULTIPLEXER δ^n with $\text{ar}(\delta^n) = n$ duplicates an agent n times using $n-1$ chained DUPLICATORS [11]. If $n = 0$, i. e. nothing has to be duplicated, the agent is instead erased.

$$\delta^n(\mathsf{x}_1, \dots, \mathsf{x}_n) \quad \equiv \quad \delta(\mathsf{x}_1, \delta(\mathsf{x}_2, \dots)) \qquad\qquad \delta^0() \quad \equiv \quad \varepsilon$$

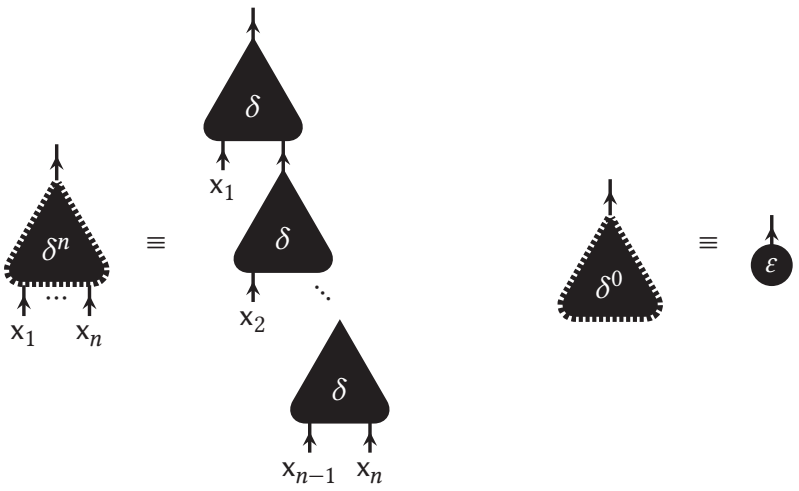


Figure 1.9 A MULTIPLEXER duplicates an agent n times.

In ABSTRACTIONS, the symbol x is duplicated the number of times it is bound in M. Both sub-terms of APPLICATIONS are translated, with the free symbols being free wires of the resulting net. Symbols are translated to wires.

Definition 1.12 (Translation $\Lambda \to \Sigma_{\text{pol}}$)**.** The configuration $\langle\mid \iota \Coloneqq \mathsf{p}, [\![M]\!]_{\mathsf{p}} \rangle$ encodes a closed term $M \in \Lambda$ using the following $[\![-]\!]_{\mathsf{p}}$:

$$\begin{aligned}
[\![\lambda x.M]\!]_{\mathsf{p}} &= \{\mathsf{p} \Coloneqq \lambda(\delta^n(\mathsf{x}_1, \dots, \mathsf{x}_n), \mathsf{m})\} \cup [\![M]\!]_{\mathsf{m}}, \quad n = \#_x(M) \\
[\![M\ N]\!]_{\mathsf{p}} &= \{\mathsf{m} \Coloneqq \alpha(\mathsf{n}, \mathsf{p})\} \cup [\![M]\!]_{\mathsf{m}} \cup [\![N]\!]_{\mathsf{n}} \\
[\![x]\!]_{\mathsf{p}} &= \{\mathsf{p} \Coloneqq \mathsf{x}_i\},
\end{aligned}$$

where x_i informally connects to a free wire x_j created by δ^n.

We visualize the translation function of definition 1.12 in figure 1.10.

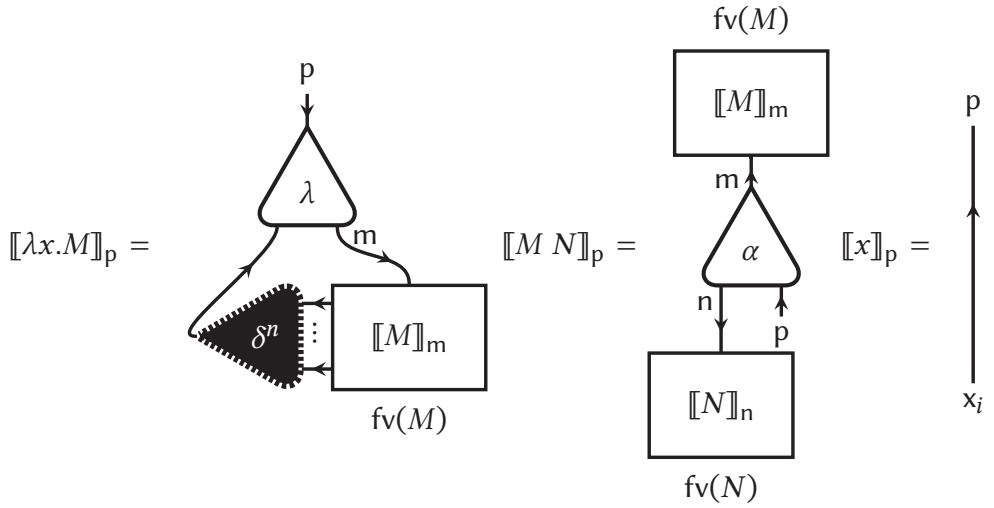


Figure 1.10 Translation from Λ to Σ_{pol}.

This minimal translation comes with the cost of inaccuracy in certain cases, which are described in section 1.3.6.

1.3.4 Read-Back

The process of reading a term back to its λ-term representation in Λ is called *read-back*. When the translation of a λ-term M is written as $[\![M]\!]$, the read-back is written as $[\![M]\!]^{-1} = M$. For detailed read-back algorithms, we refer readers to Lamping [13], Gonthier; Abadi; Lévy [10], and a book by Asperti; Guerrini [2].

Importantly, as we will see in later chapters, reading back the λ-term representation of interaction nets is not relevant to our work: The "results" of the program can be read back by looking only at the changes the *side effects* applied to the Real World.

1.3.5 Optimality

The notion of optimality in the context of β-reduction of Λ was proposed by Lévy [20]. Essentially, this property refers to reaching "the normal form in a minimum number of steps" [20]. By intuition, such minimization should have implications on the time and computing power required to evaluate programs.

In practice, β-optimality comes down to (1) reducing β-redexes of the same origin (*redex family*) in a single step, and (2) never reducing β-redexes that would be erased with different reduction strategies [2].

It has been observed by Lamping [13] that graph-based models offer a solution to the then open question of achieving β-optimality. Later, work by Asperti; Giovannetti; Naletto [15] presented an initial implementation with a base in Lafont's interaction nets ("BOHM"). The applicability of interaction nets to optimality follows from its inherent properties:

Requirement (1) is typically referred to as *sharing*, as the reduction of a term shall be shared between binding instances. In interaction nets, sharing terms requires the use of DUPLICATOR agents. As defined, λ-terms encoded as interaction nets can only be fully duplicated when they are in their normal form. β-redexes of the same origin are therefore always fully shared and never reduced multiple times across bindings [15].

Requirement (2) is more problematic. Interaction nets are inherently parallel due to their strong confluence. Yet, predicting whether parts of a net may be required or not in the future of the reduction is not possible. Reducing a term such as $(\lambda xy.y)$ (`fac` 100) requires a strict reduction order, as the term `fac` 100 must never be touched by an optimal reducer. Asperti; Guerrini [2, p. 148] argue this order to be chosen optimally by the *Call-by-Need* reduction strategy. Therefore, existing implementations such as BOHM do not incorporate parallelism into their optimal reduction [15]. Also, once a net attached to an ERASER is detached from the rest of the net, it is garbage collected externally instead of iteratively via ERASER, as it would still be reduced otherwise.

All of these restrictions change when side effects are introduced in section 3.7.

1.3.6 Bookkeeping Oracle

> ...even an optimal reduction technique, as parsimonious as it could be, cannot do any magic.
>
> *Asperti; Coppola; Martini (2000)*

As of now described, Σ_{pol} does not fully reflect β-reduction of the λ-calculus. Duplicating terms that duplicate their arguments results in duplications of DUPLICATORS—annihilations. Such annihilations imply incorrect duplication behavior, as part of the net is thus not duplicated but annihilated.

For example, the following duplication of a Church numeral as described by Barendregt [5] requires duplication of a DUPLICATOR:

$$(\lambda x.x\ x)\ \lambda sz.s\ (s\ (s\ z))$$

Only when the DUPLICATORS belong to the same "duplication process" [15], the annihilating behavior is in fact correct. Take for example the DUPLICATOR agents in the identity function, propagated through γ and Μ of its ABSTRACTOR agent:

$$(\lambda x.x\ x)\ \lambda y.y$$

Supporting all functions requires *bookkeeping* in the sense of an *oracle* [21]. This bookkeeping typically consists of labelling agents before reduction such that they interact differently depending on their labels. There must then exist additional bookkeeping agents that increment or decrement these labels. The state-of-the-art bookkeeping technique is Lambdascope by van Oostrom; van de Looij; Zwitserlood [12], which requires only a single bookkeeping agent.

The reductions for the encoding only consisting of the agents contained in Σ_{pol} is referred to as the *abstract algorithm* [21]. It has been shown by Asperti; Coppola; Martini [21] that there exists a usable subset $\Lambda^* \subset \Lambda$, where $M \overset{\beta}{\rightsquigarrow\!\!\!\!\rightarrow} M'$ corresponds to the abstract algorithm reduction $[\![M]\!] \rightsquigarrow\!\!\!\!\rightarrow [\![M']\!]$ for all $M \in \Lambda^*$. This subset is at least as large as the set of λ-terms typeable in the *elementary affine logic* (ELA) [21]. Affinity refers to the property of all symbols x being bound in a term M at most once: $\#_x(M) \leq 1$. In the abstract algorithm, the duplication of affine terms is not problematic, as no DUPLICATORS are duplicated. Importantly, β-optimality for this subset remains even without bookkeeping [21].

Existing bookkeeping procedures add a significant overhead to the reduction. Achieving optimality for the full λ-calculus through interaction nets is therefore not generally efficient [3]. Practical efficiency of optimal reduction is still possible, for example by adapting the frontend language to the affinity constraints of the abstract algorithm used in the backend, as done in the Bend language.[4]

In this thesis we focus only on the abstract algorithm, as it makes arguing about its extensions simpler. In our function and code samples we do not make use of duplications requiring an oracle. Still, full support for the λ-calculus could always be regained by incorporating existing bookkeeping techniques in the translations and interactions.

1.4 Side Effects

We define as the REAL WORLD the state of everything outside the reduction system. This includes the user's or computer's internal state as well as the internet, sensor data, or time.

Any term that modifies or depends on the REAL WORLD is considered to have a *side effect* [22]—or *is effectful*. Otherwise, it is *pure*.

[4] `https://github.com/HigherOrderCO/Bend/blob/d184863f03e796d1d657958a51dd6dd331ade92d/FEATURES.md#some-caveats-and-limitations`, 07/10/2025

We call a term *idempotent* when it has the same observable behavior—i. e. effect on reduction and the Real World—whether it is evaluated once or multiple times. Importantly, a term can be idempotent *even if* it has side effects, provided the side effects are consistent with regard to the function call.

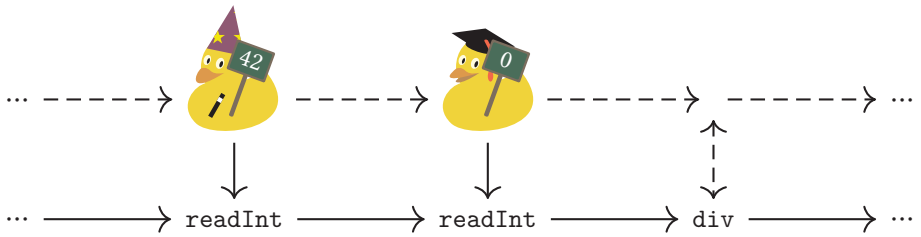


Figure 1.11 The behavior of the Real World is sequential yet unpredictable.

Side effects are inherently sequential. Modifications to the Real World can depend on previous events, therefore requiring a strict order of execution. However, the order of execution becomes less relevant when two side effects *commute*, therefore allowing a certain notion of asynchrony.

In our model, we consider everything external to the pure calculi as effectful. The implications of this definition are further discussed in chapter 3. Importantly, due to potential commutativity and tolerance of non-determinstic behavior, this does not necessarily imply strict sequentialization of all impure terms.

Chapter 2

Related Work

The inclusion of side effects in interaction nets is rare and only occurred in published implementations in recent years. In this chapter we discuss the existing implementations and specifications alongside their limitations. We first discuss two programming languages specifically targeting interaction nets. We then go on to briefly discuss other work discussing side effects in the context of interaction nets. Finally, we discuss the Clean programming language which has similar objectives to our work, although being unrelated to interaction nets. Of the presented related projects, only the Higher-Order Virtual Machine, Optiscope, and the work of Salikhmetov have the explicit goal of achieving optimality.

2.1 Higher-Order Virtual Machine

The higher-order virtual machine (HVM) was created by the Higher Order Company by Taelin [4]. It implements a runtime based on interaction nets and the polarized symmetric interaction combinators with labels. The HVM targets the GPU and is capable of massive parallelism as an implication of strong confluence.

The HVM and its frontend language Bend support basic IO primitives. We focus on the mostly undocumented IO implementation of *HVM2*, as it is the most developed. Aside from CONSTRUCTORS, DUPLICATORS and wires, HVM2 also has agents for NUMBERS, STRINGS and built-in operations.

At its core, the HVM2 runtime has a loop that keeps evaluating the net until it is in normal form. We reconstruct the loop's behavior from the source code[1]:

1. Reduce all active pairs until the net is in normal form.

[1] `https://github.com/HigherOrderCO/HVM/blob/654276018084b8f44a22b562dd68ab18583bfb5b/src/run.c#L740`, 07/01/2025

2. Read back the net as a Church tuple $\lambda s.s$ `tag` `magic` `name` `args` `cont`, where `magic` is a specific magic IO NUMBER, `name` is the effectful function name, `args` are the arguments as another tuple, and `cont` is the *continuation*.

3. If `tag` $\neq$ `IO_CALL` $= 1$ or the net did not converge to this specific structure, stop the loop.

4. Find a function with a matching `name` in the library or environment ("*book*").

5. If the function was not found, apply an error object to `cont` and repeat the loop.

6. Otherwise, call the function with `args` and repeat the loop with its result applied to `cont`.

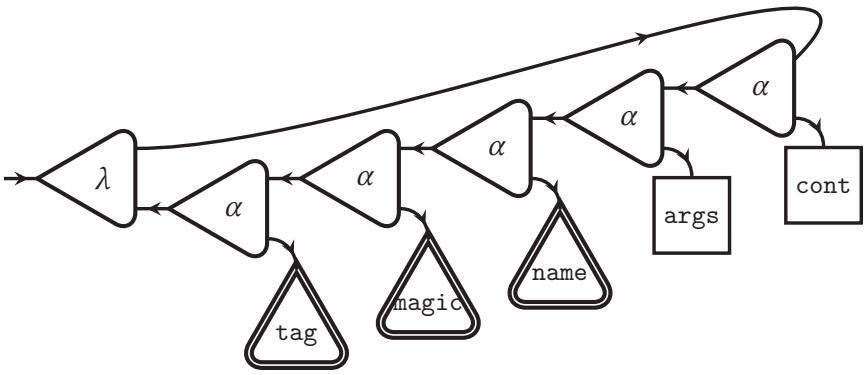


Figure 2.1 The IO structure of HVM2.

As explained by Taelin [4], the HVM does sophisticated optimizations limiting the availability of additional agents. Presumably, this is why an approach with magic numbers was chosen over dedicated effectful agents.

The approach works well with massive parallelism and bypasses the described issues where the order of side effect becomes difficult to maintain. However, it also has several limitations:

Neither can side effects happen in parallel to pure reduction, nor can side effects happen in parallel to other side effects. This makes every effect *synchronous* and potentially delayed, since the entire net has to be normalized first. We discuss uses of *asynchronous* side effects in section 3.4.

Additionally, the frontend language's syntax must either be written in or transformed to a style where the results of effectful functions are passed along to a continuation. In the case of Bend, a monadic syntax is used. We discuss a variation of this approach in chapter 6 while retaining parallel reduction and execution.

Listing 1 An example program in Bend. The syntax requires IO to happen in a continuation-based construct.

```
def foo(str: String) -> IO(u24):
  with IO:
    * <- IO/print(str)
    return wrap(0)

def main() -> IO(u24):
  foo("Hello, world!")
```

There is also a general computational overhead: Every call to an effectful function must also encode the required fields. Every time an effectful function is called, the net has to be traversed by the runtime to extract the IO structure and the arguments. By encoding functions as STRINGS, the function has to be searched and handled by the runtime, where all arguments are in turn encoded as nets.

2.2 Ivy

Ivy is a low-level programming language for the unpolarized interaction combinators[2], using a similar syntax as Lafont's notation introduced in section 1.2.1.

In comparison to Σ as described in definition 1.8, Ivy does not differentiate between CONSTRUCTORS and DUPLICATORS, and instead either commutes or annihilates based on their labels[3]. Effectful IO is implemented via *extrinsic agents*.

[2] `https://vine.dev/docs/ivy-ivm`, 07/01/2025

[3] `https://vine.dev/docs/ivy-ivm/interaction-system`, 07/01/2025

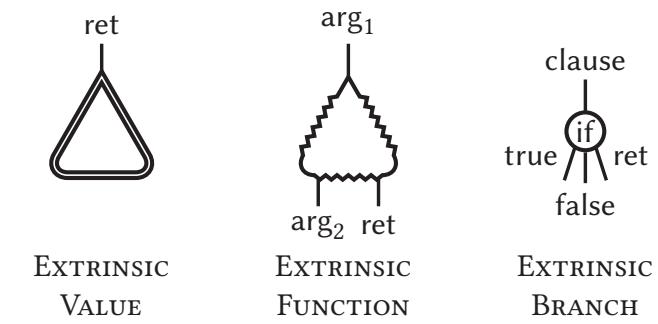


Figure 2.2 Extrinsic agents of Ivy. A visual interpretation of `https://vine.dev/docs/ivy-ivm/extrinsics`, 07/01/2025.

Extrinsic functions consume two extrinsic values by principal port and left auxiliary port. Receiving arg_2 informally requires interaction with an auxiliary port. In practice, a method similar to *effectful currying* described in section 5.4 is used to make this formally correct[4].

Extrinsic values include external data such as integers or floating-point numbers, but also a dedicated *IO handle*. This IO handle serves as a representation of the Real World and is passed through the program. It can be produced only by extrinsic functions as the modified version of the Real World after a side effect has occurred. This approach is related to the *environment passing style* of the Clean programming language [22], that we similarly discuss in section 2.4 and section 7.3.

Extrinsic branches are used to branch on extrinsic values. If the clause reduces to `true`, the `true` wire is connected to ret. If not, the `false` wire is connected to ret. These branches are the only way to extract information about extrinsic values in a way that affects the flow of the program.

Duplicating the IO handle by force encodes a *forking* behavior not dissimilar to the *fork* agent we introduce in section 3.2, and permits side effects being executed in parallel. However, IO handles can not be produced since they represent a composition of modifications to the Real World. Therefore, parallel execution can not happen everywhere, but only by having been forked before. We describe this missing feature under the name of *asynchronous actions* in section 3.4.

Extrinsic functions may only produce extrinsic values. In the case of mathematical operations this may be a number, while it is an updated version of the Real World in the case of side effects. From the perspective of a functional language based on the λ-calculus, this behavior limits the usefulness of extrinsic functions. Producing λ-terms or its corresponding net-encoding allows for

[4] `https://github.com/VineLang/vine/blob/7815b782379ce3d2901f7c96bb691ea6a348cb2c/ivm/src/ext.rs#L58-L70`, 07/01/2025

elegant notions of recursion, as described in section 3.6 and section 3.5. Furthermore, the need for EXTRINSIC BRANCHES disappears when EXTRINSIC FUNCTIONS can instead converge to Church booleans as in section 4.2, thus branching on EXTRINSIC VALUES directly.

A further limitation is the fixed number of arguments, which—minus the REAL WORLD—is 1. Functions such as `writeFile(path, data, size, flags)` are not supported in the current implementation. We discuss our approach of solving this problem using similar agents in example 3.4.

A frontend language called *Vine* compiles to Ivy[5] and uses a Rust-like syntax.

Listing 2 An example program in Vine. The IO handle must be passed to every function using IO.

```
fn foo(&io: &IO, str: String) -> N32 {
  io.println(str);
  return 0
}

pub fn main(&io: &IO) {
  foo(&io, "Hello, World!");
}
```

In order to preserve sequential execution, a reference to the IO handle representing the REAL WORLD must be passed to every function using IO. The modified IO handle resulting from EXTRINSIC FUNCTIONS is implicit in the way references are translated to interaction nets. We discuss a related approach which does not require passing the handle explicitly in section 7.4.

2.3 More Interaction Nets

There have been various other approaches to adding side effects to interaction nets that we want to discuss briefly.

Optiscope is an implementation of the Lambdascope [12] project[6]. It therefore uses interaction nets with an oracle to encode the λ-calculus. It supports side effects using monad-like functions `perform(action, k)` and `bind(x, action, k)`, which bind the results of effectful terms to their continuations.

[5] `https://vine.dev/docs/compiler/architecture`, 07/01/2025

[6] `https://github.com/etiamz/optiscope/blob/ffbcddb4d2f733f4068c126d19068d8ae7a2c490/README.md#-optiscope`, 07/29/2025

The project does not support executing effectful terms in parallel to the reduction or other side effects, as the entire net is reduced in a specific order[7] (Call-by-Need, as per β-optimality [2, p. 148]). As in related languages such as Bend, effectful terms must be wrapped in carefully placed syntactic constructs in order to guarantee correct sequential execution.

In a language by Salikhmetov [23], side effects can be specified directly in the specifications of the interaction rules. Additional data may be attached to agents in the form of infinitary agents, such that $\Sigma \cup \{a_d \mid d \in \mathcal{D}, a \in \Sigma\}$ is an extended interaction system where $\mathcal{D}$ contains any data of the Real World. This additional data could then be used to choose different interaction rules depending on the attached data, as well as producing agents with manipulated data after interaction [23].

Listing 3 Example adapted from Salikhmetov [23], where side effects are specified in the interaction rules. The syntax is a mix between Lafont's notation [1] and LaTeX.

```
\eraser {
  console.log("eraser >< duplicator");
} \duplicator[\eraser, \eraser];

\eraser {
  console.log("eraser >< constructor");
} \constructor[\eraser, \eraser];

\duplicator[\constructor(x, y), \constructor(v, w)] {
  console.log("duplicator >< constructor");
} \constructor[\duplicator(x, v), \duplicator(y, w)];
```

Here, sequential execution is not guaranteed except when a specific reduction order for the active pairs is chosen. They did not develop any notion of parallel execution or reduction.

In a more traditional implementation before interaction combinators were introduced, Gay [24] proposed a stream-based approach aiming to implement a read-eval-print loop (REPL) for a minimal language. In its essence, it makes IORead agents continuously consume nilary agents containing characters until it interacts with an agent containing a newline. During this consumption, it builds up a tree of the parsed tokens, which is then evaluated. The IORead agent is then transformed to a IOWrite agent.

[7] https://github.com/etiamz/optiscope/blob/ffbcddb4d2f733f4068c126d19068d8ae7a2c490/README.md#rules-for-side-effects, 07/29/2025

The approach taken by Gay [24] is inflexible regarding other types of side effects, since not all side effects can be represented using streams of nilary agents. If additional extension agents supporting other side effects were added, confluence in the aspect of parallelism could break.

2.4 Concurrent Clean

Clean is a functional programming language with a uniqueness type system. Although Clean has otherwise no relation at all to interaction nets, it still has a notable similarity: It, too, has the goal of reduction staying generally unordered while having a strict sequence of side effects embedded within [25].

The uniqueness type system guarantees that an object may only ever be referenced once in a program. Effectively, this property can be used to pass an environment around containing state of the Real World. The type system then guarantees that (1) the order of this passing is guaranteed, and (2) that there can only ever exist a single version of this state, thus preventing race conditions [25].

Functions aiming to modify this state must receive an object by argument, and produce the updated version as a result. Specifically, Clean provides a *hierarchy* system of the Real World: If one wants to write to files, one may receive an object that contains only the state of all files, and return this object with the respective file modification [25].

The elegance of this model comes with limitations: For one, functions may never be effectful if they do not receive and produce according environments. This requires threading environments through the entire program, potentially leading to worse readability. It also does not directly support parallelism: Since environments are tracked by the uniqueness system, different parts of the same environment may not be modified in parallel.

Chapter 3

Optimal Effects

The relationship between the pure λ-calculus, interaction nets, and optimal reduction have been thoroughly investigated, notably by Lévy [20], Lamping [13], and in a book by Asperti; Guerrini [2].

Extensions adding side effects to the λ-calculus have also been discussed—in fact, most modern functional programming languages are based on the pure λ-calculus. We describe these existing extensions in section 3.1 and discuss limitations regarding their applicability to interaction nets.

The calculus we introduce in section 3.2 is a minimal extension of Λ. It is defined in such a way that we can later realize clear relationships to existing paradigms such as monads or state-passing, while also allowing trivial translations to interaction nets. In the following sections we discuss the properties, limitations, and interpretations of the introduced calculus. Finally, in the last sections we come back to the concept of optimality and relate the addition of side effects to the challenges introduced in chapter 1.

3.1 Related Work

In chapter 2 we have discussed the implementation of side effects in several programming languages. We now focus on solutions from the perspective of the pure λ-calculus.

Moggi [26] has introduced a monadic extension of the λ-calculus named *computational λ-calculus*. There, the sequentialization of side effects is guaranteed by *monad laws*. This extension has the benefit of not affecting the reduction behavior of the rest of Λ. Although not specified in the computational λ-calculus itself, extensions also allow parallel as well as asynchronous side effects [27, 28]. However, the execution order is generally fixed by the way monadic bindings are linked to each other. It also requires strict syntactic separation of effectful and

pure terms, in the sense that every effectful term must be part of the monadic construct. Together, these limitations typically imply a certain syntax inherited by the frontend, where side effects can only be used as part of a construct threaded through the program. Still, we discuss this approach further in chapter 6.

Call-by-Push-Value (CBPV) is a framework introduced in the PhD thesis of Levy [29]. In contrast to Moggi's monadic style, it allows for modelling various reduction strategies even in the presence of side effects. Similar to monads, it has a sequentialization construct binding multiple computations together. It also differentiates strictly between effectful terms (*computations*) and pure terms (*values*). Translation between the two is accomplished by "thunking" and "forcing" the terms respectively.

In a way, the calculus we introduce in section 3.2 has strong similarities to CBPV. However, our core calculus abstracts away fully from the notion of sequentialization or reduction strategies. Instead, such sequentialization is added via extension, thus becoming more closely related to CBPV (see section 7.6). By initially ignoring such details, a general translation to interaction nets becomes trivial.

Our core calculus uses *tokens* as a way of tracking the execution of effectful terms. Similar tokens have been used in abstract machines to track the reduction state in an accumulating manner [30]. In recent work by Ahman; Pretnar [31], a term-traversing token ("signal") is used to track effects. There, signals are attached to a continuation and propagate outwards or inwards of a term depending on their kind. Once an outgoing signal would propagate through a parallel broadcaster, it is duplicated and sent to the thread attached to the broadcaster. Similarly, an incoming signal is duplicated and propagated through both threads. This allows for a certain notion of asynchrony and parallelism, while still enforcing sequential execution. We take the approach as inspiration for our implementation of *forks* (see definition 3.3).

However, by construction, the propagated signals must always be caught by special constructs. Interacting with the Real World requires sending a signal which is then propagated outwards. In comparison, our tokens do not generally require propagation or their handling. Instead, applying tokens can immediately cause side effects independent of the general reduction state.

3.2 Effectful λ-Calculus

We extend the pure λ-calculus introduced in definition 1.1 with actions that cause side effects. We introduce an additional kind of term called data that can only be produced by actions. In order to allow fine-grained control of when the actions are executed, we also introduce a token that executes an action upon

application.

The TOKEN is related to the unary reduction symbol used by Sinot [32]. There it is used to denote the evaluation function *entering* into and *returning* from a term of the λ-calculus. In order to derive a semantics for executing ACTIONS in a specific order without interfering with the reduction of pure terms, we make the TOKENS first-class citizens of the calculus itself. We correspondingly use the COTOKEN to mark returning TOKENS.

Definition 3.1 (Terms of Λ_{eff})**.** The terms of the effectful λ-calculus Λ_{eff} are:

t	$::= (\lambda x.t) \mid (t\ t) \mid x$	TERMS OF Λ
	$\mid \triangleleft$	TOKEN
	$\mid \triangleright$	COTOKEN
	$\mid \wedge$	CONJUNCTIVE FORK
	$\mid \vee$	DISJUNCTIVE FORK
	$\mid d$	DATA
	$\mid a$	ACTION
d	$::= \langle\rangle$	UNIT
	$\mid \langle\cdots\rangle$	ARBITRARY DATA
a	$\in$ `read`, `write`, ...	ACTION NAME
x	$\in$ a, b, ... , y, z	SYMBOL

We write a general ACTION as $\xi^i_{\mathcal{C}} \in a$, with $\mathcal{C}$ being a *list*[1] of partially applied arguments, and i being the *arity*. The arity of an ACTION together with the number of partially applied arguments is always greater than zero, $|\mathcal{C}| + i > 0$.

We differentiate between *executing* and *reducing* terms, where we refer to execution as requiring synchronization with the REAL WORLD—i. e. potentially modifying or depending on its state.

Definition 3.2 (ξ-Reduction)**.** ACTIONS ξ-reduce when applied to DATA and return the same ACTION with the DATA appended to its list of arguments. Given $\pi \in d$ and $i > 0$,

$$\xi^i_{\mathcal{C}}\ \pi \overset{\xi}{\rightsquigarrow} \xi^{i-1}_{\mathcal{C}+\!\!+[\pi]}.$$

As a special case of ξ-reduction, ACTIONS get *executed* when $i = 0$ and applied to the TOKEN $\triangleleft$, and produce a term while causing a side effect.

$$\xi^0_{\mathcal{C}} \triangleleft \overset{\xi}{\rightsquigarrow} M, \quad \text{with } M \in t \text{ closed},$$

[1] an ordered multiset $[a, b, \dots]$ equipped with concatenation $+\!\!+$.

where the second arrow indicates that the execution may interact with the Real World. Only ξ-execution is capable of causing side effects.

If $i > 0$ and a token is applied, it is *reflected* as a cotoken:

$$\xi_{\mathcal{C}}^{i} \triangleleft \overset{\xi}{\rightsquigarrow} \xi_{\mathcal{C}}^{i} \triangleright.$$

In order to prevent the further existence of such cotoken, it may also be applied to a token, thus *annihilating*:

$$M\ (\triangleright\ \triangleleft) \overset{\xi}{\rightsquigarrow} M$$
$$(\triangleright\ \triangleleft)\ M \overset{\xi}{\rightsquigarrow} M$$

With the addition of ξ-reduction and -execution, the erasing and duplicating behavior implicit in β-reduction has changed: Where previously any term not substituted during reduction does not have to be reduced—say $(\lambda x.M)\ N$ with $\#_x(M) = 0$, ξ-execution may still require N to be $\beta\xi$-reduced in order for its actions to be executed. On the other hand, substitution with $\#_x(M) > 1$ may imply that actions will be executed more than once. In these cases, we therefore define that β-reduction is allowed only once N is in $\beta\xi$-normal form, or it does not contain any token and cotoken. We further do not allow β-reduction of tokens, such that tokens by themself can not be substituted, duplicated, or erased. Importantly, without the use of tokens this does not diverge from the defined behavior of Λ.

Forks duplicate a token and send it to two distinct *threads*. We take as inspiration the similar `race` (disjunctive fork) and `concurrently` (conjunctive fork) from Haskell's `Control.Concurrent.Async` package [33]. Depending on whether the fork is conjunctive or disjunctive, it reacts differently when the tokens eventually return as cotokens.

Definition 3.3 (ψ-Execution)**.** Given a fork with two threads $M, N \in \Lambda_{\text{eff}}$ is executed, the token propagates through both threads:

$$\wedge\ M\ N \triangleleft \overset{\psi}{\rightsquigarrow} \wedge\ (M \triangleleft)\ (N \triangleleft), \quad \text{and}$$
$$\vee\ M\ N \triangleleft \overset{\psi}{\rightsquigarrow} \vee\ (M \triangleleft)\ (N \triangleleft).$$

In the conjunctive case, both tokens must return in order for the terms to be applied to each other. In the disjunctive case, either token can return, with the other term being erased:

$$\wedge\ (M \triangleright)\ (N \triangleright) \overset{\psi}{\rightsquigarrow} M\ N \triangleleft,$$
$$\vee\ (M \triangleright)\ N \overset{\psi}{\rightsquigarrow} M \triangleright, \quad \text{and}$$
$$\vee\ M\ (N \triangleright) \overset{\psi}{\rightsquigarrow} N \triangleright.$$

As with ξ-reduction, fully erasing M or N may only happen once it can not cause side-effects—i. e. if it is in $\beta\xi$-normal form, or does not contain TOKENS or COTOKENS.

Notably, the DISJUNCTIVE FORK is ambiguous in the sense that it is not clear what will happen in the case of two returning COTOKENS $\vee\ (M \triangleright)\ (N \triangleright)$. In this case, choosing the reduction rule which is applied first may be done at random. We discuss applications of such ambiguity in section 4.4.

Functions with side effects in imperative programming languages often receive multiple arguments. In order to write such functions in Λ_{eff}, we describe a concept called *effectful currying* that resembles the behavior of functions with multiple arguments in the pure λ-calculus Λ by repeatedly applying the ACTION *partially*.

Example 3.4 (Effectful Currying)**.** Let $\texttt{writeFile}^3_{\mathcal{C}}$ be an effectfully curried ACTION of arity 2, receiving a path string and some content. It could be implemented in such a way that it reduces like the following:

$$
\begin{aligned}
&\texttt{writeFile}^2_{\emptyset}\ \langle\text{“example.txt”}\rangle\ \langle\text{“hello, world”}\rangle\ \triangleleft \\
\overset{\xi}{\rightsquigarrow}\ &\texttt{writeFile}^1_{[\langle\text{“example.txt”}\rangle]}\ \langle\text{“hello, world”}\rangle\ \triangleleft \\
\overset{\xi}{\rightsquigarrow}\ &\texttt{writeFile}^0_{[\langle\text{“example.txt”}\rangle,\langle\text{“hello, world”}\rangle]}\ \triangleleft \\
\overset{\xi}{\rightsquigarrow}\ &\langle\rangle\ \triangleright
\end{aligned}
$$

A type system for a language built upon Λ_{eff} may provide further information and guarantees about the kind of DATA arguments allowed. For now, we assume that terms are written in a way that matches the internal definitions of the used ACTIONS.

As with any execution behavior of ACTIONS, the final TOKEN reflection is also implemented by the host or reducer and may not be relied upon. We state this behavior because there exist reasons why specific ACTIONS may not want to reflect a COTOKEN upon execution: TOKENS effectively represent threads, as discussed in section 3.5. In a sense, ACTIONS producing a term without (CO-)TOKEN *terminate* a thread, whereas ACTIONS producing a term with multiple (CO-)TOKENS *spawn* threads.

3.3 Problem of Order

The pure λ-calculus has the Church-Rosser property, meaning that it does not influence the resulting term in which order a term is reduced. With the addition of side effects, the reduction of terms may depend on state outside the domain

of the term itself–the Real World. Side effects may also behave differently depending on the *point in time* in which they are executed. Since, in practice, the point in time of execution depends on the order of reduction, actions can only ever be confluent if we ignore the time of execution.

Therefore, the effectful λ-calculus has lost the Church-Rosser property. We see this problem in a continuation to the example 3.4 above.

Example 3.5 (Reduction Order)**.** Let `readFile` be an action that receives a path string and results in data with the content of the file. We then write an application of the results of actions using `writeFile` and `readFile` to a term printing its first argument using a `print` action:

$$
\begin{aligned}
&(\lambda xy.\texttt{print}^1_\emptyset\ x \triangleleft)\\
&\ (\texttt{readFile}^1_\emptyset\ \langle\text{“example.txt”}\rangle\ \triangleleft)\\
&\ (\texttt{writeFile}^2_\emptyset\ \langle\text{“example.txt”}\rangle\ \langle\text{“hello, world”}\rangle\ \triangleleft)
\end{aligned}
$$

We observe that it is unclear when and in which order the two arguments should be reduced. When analyzed using traditional reduction strategies for the pure λ-calculus:

- *Call-by-Value*: The arguments get reduced in right-to-left order and then substituted, therefore printing “hello, world”.
- *Call-by-Name*: The arguments get substituted before they are reduced, therefore the previous, unknown content of “example.txt” is printed.

Aside from ξ-execution and the disjunctive fork, Λ_{eff} is a straightforward extension of Λ, therefore retaining its behavior and laws.

Corollary 3.6 (Confluence of $\overset{\beta\xi\psi}{\rightsquigarrow\!\!\!\!\rightarrow}$)**.** *The reduction relation* $\overset{\beta\xi\psi}{\rightsquigarrow\!\!\!\!\rightarrow}$ *of* $\Lambda_{\text{eff}} \setminus \{\vee\}$ *satisfies the diamond property (effectively:* is Church-Rosser*). Notably, this does not include* $\overset{\xi}{\rightsquigarrow}$.

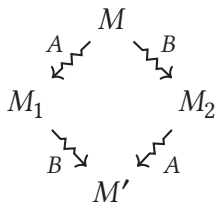
In general, ξ-execution with multiple tokens is not confluent. If, instead, there is only one token which is passed and reflected in a specific way—defined by a *token-passing semantics*—we regain confluence to some extent. To argue about the token-passing semantics, we make use of the cotoken, which may be the result of a reflected token. We discuss the semantics of token-passing for a monadic style in section 6.3, and for a direct style in section 7.4.1.

Still, multiple tokens can make sense even in a confluent setting: If the side effects *do not* influence or depend on each other, the actions can also be executed without the sequentiality constraint, or even in parallel. The use of tokens permits an elegant notion of encoding this behavior.

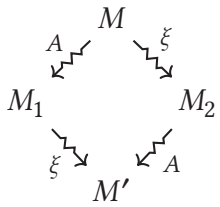
3.4 Asynchronous Actions

Asynchronous ACTIONS are ACTIONS not influencing the rest of the program—i. e. the execution behavior of any other ACTIONS. We argue about asynchronous execution using *commutativity*.

When two ACTIONS commute, the order in which they are executed does not induce a difference in the observable behavior. Say we have ACTIONS A, B with arity 0. Then we can define the reduction relation $\overset{A}{\rightsquigarrow}$ for $A \triangleleft \overset{\xi}{\rightsquigarrow} A'$ and $\overset{B}{\rightsquigarrow}$ for $B \triangleleft \overset{\xi}{\rightsquigarrow} B'$. We can visualize the commutativity in the following diagram:



This leads to a stronger property that we call *orthogonality*. An ACTION A is *orthogonal* if its execution commutes with the execution of all ACTIONS ξ in a term:



We come to the conclusion that any orthogonal ACTION may also be executed asynchronously. Without a corresponding effect-tracking system, orthogonality is not a property that can be detected. Instead, certain ACTIONS are marked as orthogonal by being applied to a TOKEN—thus executing them independently of a token-passing semantics.

Our use of the term *asynchronous* is related to its use in Haskell, where `unsafePerformIO` marks terms applicable to asynchronous execution [27]. Examples of orthogonal ACTIONS where the use of a TOKEN is *safe*, include:

- Incrementing an external, write-only counter.
- Logging to an unordered program log, locked by newline.
- Accessing resources such as files or websites by their checksum hash—assuming hash collisions are not possible.
- Using pure or mathematical operations such as `add` or `div`, assuming they do not require sequentiality by low-level hardware restrictions.

- Calling pure, foreign functions, assuming they do not change over time.
- Loading pure, constant DATA such as strings or integers.

In combination with idempotence, orthogonality can provide a stronger property: Since the term may then be executed asynchronously and it does not matter whether it is *actually* executed multiple times, the ACTION execution is applicable to being *shared asynchronously*. We explore this property in the aspect of optimality in section 3.7. In the list above, the bottom four examples may be interpreted as being idempotent.

Of course, if an ACTION is *not* orthogonal but still executed asynchronously, confluence is broken.

Corollary 3.7 (Confluence of Orthogonal $\overset{\beta\xi\psi}{\rightrightarrows}$)**.** *Assuming all ACTIONS to be orthogonal, the reduction relation* $\overset{\beta\xi\psi}{\rightrightarrows}$ *of* $\Lambda_{\text{eff}} \setminus \{\vee\}$ *satisfies the diamond property modulo time.*

In corollary 3.7, ignoring the *point in time* of execution is required as asynchronous, orthogonal ACTIONS may still depend on a specific point in time of execution—which may not be guaranteed in practice without sequential and deterministic reduction.

3.5 Interpretations

In a sense, ξ-execution describes the *communication* between the pure world of the λ-calculus and the contaminated REAL WORLD. DATA encodes *links* or pointers from the λ-calculus to the REAL WORLD, uninterpretable by λ-terms themself. Equivalently, in our model, pure λ-terms are not interpretable by the REAL WORLD. Arbitrary λ-terms, including such DATA or ACTION backlinks, may be *received* by the pure world, while it is only capable of *sending* impure DATA which it has previously received.

This view has implications on the way such language can be interpreted. The pure world may in fact be simulated in an *abstract machine* communicating with its hosting machine, which in turn communicates with the REAL WORLD. The abstract machine would then be initialized to $M \triangleleft \triangleleft$, M being an ACTION consuming a TOKEN and receiving the program it is supposed to evaluate. The second TOKEN will then trigger the token-passing semantics.

Furthermore, TOKENS resemble the *eyes* of the REAL WORLD from which it monitors the execution. It is a consequence of the ξ-execution rule that there can only ever be as many parallel or concurrent executions as there are TOKENS in a term. Therefore, there exists a correspondence between term-traversing TOKENS and *threads* of the evaluating machine.

ψ-execution as well as other ways of erasing or duplicating TOKENS consequently require communication with the REAL WORLD, as new TOKENS and threads are created.

3.6 Recursion

We have observed that the system evaluating a program written in a frontend programming language based on Λ_{eff} will require access to the REAL WORLD and therefore the definitions of any given program. Since ξ-execution may produce any closed term of Λ_{eff}, it can be used to simulate recursion.

This has implications on the problems described in section 1.3.6, since typical solutions to adding recursion to the λ-calculus use self-duplication unsupported by the abstract algorithm without bookkeeping oracle. Instead of self-duplicating iteratively, a recursive application will expand to the term representing the function.

$$\begin{aligned}
&\text{fac} = (\lambda x.x\ x)\ \lambda f n.\texttt{isZero}^1_\emptyset\ n \triangleleft \\
&\quad \langle 1 \rangle \\
&\quad (\texttt{mult}^2_\emptyset\ \langle 0 \rangle\ (f\ f\ (\texttt{pred}^1_\emptyset\ n \triangleleft)) \triangleleft)
\end{aligned}
\qquad
\begin{aligned}
&\text{fac} = \lambda n.\texttt{isZero}^1_\emptyset\ n \triangleleft \\
&\quad \langle 1 \rangle \\
&\quad (\texttt{mult}^2_\emptyset\ \langle 0 \rangle\ (\texttt{rec}^0_{[\langle\rangle]} \triangleleft (\texttt{pred}^1_\emptyset\ n \triangleleft)) \triangleleft) \\
&\text{where } \texttt{rec}^0_{[\langle\rangle]} \triangleleft \overset{\xi}{\rightsquigarrow} \lambda n.\texttt{isZero}^1_\emptyset\ n \ldots .
\end{aligned}$$

Figure 3.1 A self-duplicating vs. an effectfully recursive implementation of the factorial function for positive integers. We assume that no ACTIONS reflect a COTOKEN and `isZero` results in a Church boolean (see *Branching* in definition 4.1).

Both implementations in figure 3.1 suffer from the problem that, depending on the reduction order or strategy, one may end up recursing indefinitely in either self-duplicating or self-expanding `fac` functions. An appropriate token-passing semantics as in chapter 7 prevents this and eliminates the need for carefully placing TOKENS. In practice, this technique has limitations when terms depend on many other definitions: Since the resulting term must be *closed*, every recursive call must have its *closure* unwrapped and applied to itself. We describe the exact procedure in section 4.3.

It is worth noting that with this expansion, duplication of external terms can be gained back via ACTIONS—thus eliminating the general need for an oracle. However, in order to stay optimal, terms duplicated this way must then be

produced in normal form, as otherwise redexes will be reduced more than once. This property shall be guaranteed by the respective implementation.

Similarly, in order to mimic the optimal reduction of fixed-point recursion in the λ-calculus, all recursively expanded terms may be required to be in normal form up to execution of ACTIONS.

3.7 Effectful Optimality

With the addition of side effects, we can discuss a notion of optimality similar to β-optimality in section 1.3.5. There, Lévy's objective is to minimize the β-reduction steps by sharing all terms of the same origin while not doing any redundant reductions.

A notion of sharing can also be applied to ACTIONS: Any orthogonal *and* idempotent ACTION may be executed once and then substituted into all of its uses. In a ξ-optimal system, there must therefore exist a way to mark certain terms as being applicable to this sharing. In order to maximize sharing of ACTIONS, terms *becoming* orthogonal must be allowed to reduce asynchronously. In an effectful system, optimality then comes with the additional objective of *maximizing asynchrony*.

In Λ_{eff}, this marking of shareable terms happens via TOKENS. It is indeed another reason for our comparatively strict definition of purity: By counting the receiving of static or pure data as side effects, these loads must by definition also be shared maximally in an optimal system. As with sharing in pure β-reduction, maximizing the sharing at the term-level can only be guaranteed with systems such as interaction nets, which we extend accordingly in chapter 5.

A *hierarchy* of terms that may be executed in parallel can be constructed by chaining FORKS in such a way that the next chain of FORKS is only executed when all COTOKENS have returned. This way, parallel execution of effects can be staggered such that certain effects only happen once they become orthogonal.

The second objective is the one of not reducing terms that would be erased with any different reduction strategy. We have discussed in section 3.2 how β-reduction is not allowed when it influences the behavior of side effects, say by duplicating or erasing ξ-redexes with TOKENS. Therefore, in ξ-reduction it is less obvious whether reductions are redundant or not: Terms not being bound does not imply that they do not have to be reduced. Although sophisticated techniques such as type or effect systems [34] may provide further information about the parts of a term guaranteed to be redundant, we argue that *in general* this second objective of optimality does not have relevant meaning with the addition of side effects.

Still, existing optimizations may be applied via heuristics on unbound terms

when they are incapable of effects—say once they are in $\beta\xi$-normal form or do not contain TOKENS and COTOKENS. Reduction strategies such as Call-by-Need used to satisfy optimality [2, p. 148] would have to be extended with these heuristics while also reducing potentially effectful terms even when they are not *needed*.

We want to emphasize that parallelism in the context of optimality thus becomes more appropriate in Λ_{eff} than in Λ. It is indeed impossible to predict the exact behavior of any term and its potential for side effects without reducing it in the first place—which may as well be done in parallel. Massive parallelism in Λ can *always* cause redundant reductions, while massive parallelism in Λ_{eff} may only *sometimes* cause redundant reductions. It remains future work to make this distinction more precise, potentially merging it with type-based reasoning.

Chapter 4

A Simple Language

Programming in the pure or effectful λ-calculus is arguably not desirable. Difficulties arise from the lack of syntactic constructs for organizing the code into self-contained blocks, as well as the lack of data types such as strings or integers.

In order to showcase the practical application of interaction nets and the λ-calculus, we introduce the $\mathcal{L}_{\text{sim}}$ language that has close similarities to the introduced λ-calculi while also resembling the style of common programming languages. Further extensions to $\mathcal{L}_{\text{sim}}$ are described in chapter 6 and chapter 7.

In section 4.1 we define the syntax of $\mathcal{L}_{\text{sim}}$. We then describe the translations from and to the λ-calculus in section 4.2, eventually allowing a transitive translation to interaction nets. We continue with giving an in-depth description of the way FORKS may be used in $\mathcal{L}_{\text{sim}}$, finally leading to the use of ACTIONS to represent foreign functions.

4.1 The Language $\mathcal{L}_{\text{sim}}$

The language $\mathcal{L}_{\text{sim}}$ is a syntactic extension of Λ_{eff}. It supports named and nested definitions, a syntax for branches, and several built-in datatypes. In order to support asynchronous actions as defined in section 3.4, it also has a *force* operator which forces the (asynchronous) execution of ACTIONS or FORKS. Since the TOKENS themself are not a relevant part of its semantics, it does not have a COTOKEN.

Definition 4.1 (Terms of $\mathcal{L}_{\text{sim}}$)**.** The terms of the language $\mathcal{L}_{\text{sim}}$ are:

t	$::= \textbf{let}\ x\ x^* = t; t$	Let Expression
	$\mid \textbf{if}\ (t)\ \textbf{then}\ t\ \textbf{else}\ t$	If Expression
	$\mid \textbf{join}\ (t)^*$	Conj. Fork
	$\mid \textbf{race}\ (t)^*$	Disj. Fork
	$\mid t\ t$	Application
	$\mid (t)$	Block
	$\mid x$	Binding
	$\mid\ !$	Force (Token)
	$\mid a$	Action
	$\mid n \mid s \mid b \mid u$	Value
x	$\in \text{x}, \text{num}, \text{add}, \ldots$	Name
a	$\in \texttt{read}, \texttt{write}, \ldots$	Action Name
n	$\in 0, 1, -1, 2, -2, \ldots$	Integer
s	$\in$ “abc”, “hello, world”, …	String
b	$\in \texttt{true}, \texttt{false}$	Boolean
u	$::= \langle\rangle$	Unit

We use the **let** keyword to assign a name to a term, which can then be used in its following terms. This name can be called with multiple arguments, corresponding to the named identifiers bound by **let**. Definitions with multiple arguments are desugared such that each **let** receives at most a single argument. Let $f^{(i)} \notin \text{fv}(t)$ be fresh. Then,

$$
\begin{aligned}
\textbf{let}\ f\ a_1\ \ldots\ a_n = M; N \quad \equiv \quad & \textbf{let}\ f\ a_1 = (\\
& \quad \textbf{let}\ f'\ a_2 = (\ldots \\
& \quad\quad \textbf{let}\ f^{(n)}\ a_n = M; f^{(n)} \\
& \quad \ldots); f' \\
&); N.
\end{aligned}
$$

Forcing the execution of a term corresponds to the application of a token in Λ_{eff}. The translation of $\mathcal{L}_{\text{sim}}$ guarantees that no cotoken escapes (see section 4.2).

In contrast to Λ_{eff}, forks in $\mathcal{L}_{\text{sim}}$ allow branching on n terms instead of just two. Joining n threads will represent a right-fold on the results of the concurrently executed terms. Analogously, racing n threads returns only the thread which was done executing first.

We use $\mathcal{L}_{\text{sim}}$ only as an intermediate language, as we translate it to Λ_{eff} in definition 4.2. Its semantics is otherwise similar to common ML-like languages such as OCaml [35] and we do not define it further.

However, it is important to note that the semantics does not include the execution of effectful terms—this is only discussed in chapter 6 and chapter 7 . Even without the token-passing semantics, $\mathcal{L}_{\text{sim}}$ sequentializes the execution of effectful terms when they depend on the produced terms linearly.

Listing 4 A simple program in $\mathcal{L}_{\text{sim}}$. The ACTIONS are executed sequentially since every ACTION gets partially applied to the result of the previous ACTION.

```
let x = add (1 !) (2 !) !;

let str = if (isEqual x (3 !) !)
  then ("three" !)
  else ("hit by cosmic ray!" !);

print str !
```

This only works if the ACTIONS have higher arity, as their partial application requires the ACTIONS to be converged to DATA. Yet, as obvious in listing 4, there exists a syntactic overhead in executing ACTIONS asynchronously. For constant *data* without further arguments it also takes several assumptions about the implementation of how the *loading* of data is handled internally. In order to guarantee sequential execution and eliminate the syntactic TOKEN overhead, dedicated token-passing semantics are required.

4.2 Translations

In order to highlight the relation to Λ, we define the translation *from* Λ as well as *to* Λ_{eff}.

Proposition 4.1 (Translation $\Lambda \to \mathcal{L}_{\text{sim}}$)**.** *Any term of the pure λ-calculus Λ can be written in $\mathcal{L}_{\text{sim}}$.*

Proof. Obvious by translation using $[\![-]\!] : \Lambda \to \mathcal{L}_{\text{sim}}$:

$$
\begin{aligned}
[\![\lambda x.M]\!] &= \textbf{let } f\ [\![x]\!] = [\![M]\!]\textbf{; } f, \quad \text{with } f \notin \text{fv}(M) \text{ fresh} \\
[\![M\ N]\!] &= [\![M]\!]\ [\![N]\!] \\
[\![x]\!] &= x
\end{aligned}
$$

□

The translation of $\mathcal{L}_{\text{sim}}$ to Λ_{eff} will then allow a transitive translation from $\mathcal{L}_{\text{sim}}$ to interaction nets extended with side effects.

Definition 4.2 (Translation $\mathcal{L}_{\text{sim}} \to \Lambda_{\text{eff}}$)**.** Any term of $\mathcal{L}_{\text{sim}}$ can be written in Λ_{eff} using $[\![-]\!] : \mathcal{L}_{\text{sim}} \to \Lambda_{\text{eff}}$:

$$
\begin{array}{lll}
[\![\textbf{let } f\ a_1\ a_2\ \ldots\ a_i = M;\ N]\!] & = (\lambda f.[\![N]\!])\ \lambda a_1 a_2 \ldots a_i.[\![M]\!] & \\
[\![\textbf{if } (M) \textbf{ then } T \textbf{ else } F]\!] & = [\![M]\!]\ [\![T]\!]\ [\![F]\!] & \\
[\![\textbf{join } M_1\ M_2\ \ldots\ M_i]\!] & = \wedge\ [\![M_1]\!]\ (\wedge\ [\![M_2]\!]\ (\ldots\ (\wedge\ [\![M_{i-1}]\!]\ [\![M_i]\!])\ldots)) & \\
[\![\textbf{race } M_1\ M_2\ \ldots\ M_i]\!] & = \vee\ [\![M_1]\!]\ (\vee\ [\![M_2]\!]\ (\ldots\ (\vee\ [\![M_{i-1}]\!]\ [\![M_i]\!])\ldots)) & \\
[\![M\ N]\!] & = [\![M]\!]\ [\![N]\!] & \\
[\![(M)]\!] & = [\![M]\!] & \\
[\![!]\!] & = \triangleleft\ [\triangleleft] & \\
[\![\texttt{write}]\!] & = \texttt{write}^{i}_{\emptyset} & \\
[\![314]\!] & = \texttt{num}^{0}_{[\langle\rangle]}, & \text{where } \texttt{num}^{0}_{[\langle\rangle]} \triangleleft \overset{\xi}{\leftrightarrows} \langle 314 \rangle \\
[\![\text{“...”}]\!] & = \texttt{string}^{0}_{[\langle\rangle]}, & \text{where } \texttt{string}^{0}_{[\langle\rangle]} \triangleleft \overset{\xi}{\leftrightarrows} \langle \text{“...”} \rangle \\
[\![\text{true}]\!] & = \lambda t f.t & \\
[\![\text{false}]\!] & = \lambda t f.f & \\
[\![\langle\rangle]\!] & = \texttt{unit}^{0}_{[\langle\rangle]}\ \triangleleft, & \text{where } \texttt{unit}^{0}_{[\langle\rangle]} \triangleleft \overset{\xi}{\leftrightarrows} \langle\rangle\,.
\end{array}
$$

The translation follows from several observations:

Let Expression LET EXPRESSIONS have a close resemblance to ABSTRACTIONS. In definition 4.2 we abstract the body M with the required arguments a_j and bind it to the name f in the next definition N. This translation preserves the sharing of the definition, as the bindings in Λ_{eff} are shared. It also preserves the expected behavior of shadowed bindings, since the translation to ABSTRACTIONS takes care of renaming transitively (α-conversion)

Branching & Booleans An IF EXPRESSION can not trivially be translated to an ACTION as we have defined it, since ACTIONS only consume arguments when they have been reduced to DATA, therefore executing *both* branches instead of one of them. Instead, we use boolean combinators as described by Barendregt [5, p. 133] (*Church Booleans*).

They work by abstracting two times over a binding. If the Church boolean encodes `true`, the binding is bound to the outer ABSTRACTION. If it encodes `false`, it is bound to the inner ABSTRACTION. The condition b of an IF EXPRESSION must reduce to a Church boolean. Then, the translation in definition 4.2 guarantees

that either the true-case or the false-case is returned from the APPLICATION, as only one of them is bound in the Church boolean of the condition.

With an appropriate token-passing semantics which does not execute in ABSTRACTIONS, the execution of ACTIONS in branches can be deferred by abstracting all branches (commonly referred to as *thunks*) and forcing their evaluation by an application to the entire IF EXPRESSION. We use this translation when targeting the direct-style semantics described in chapter 7.

Force The specific translation of FORCE is implementation-defined. We have prescribed only that it must be guaranteed that the use of FORCE will not produce a rogue COTOKEN, which of course depends on whether ACTIONS consume or reflect the TOKEN. In the case of reflecting ACTIONS, forcing a term may translate to sending two TOKENS, thus potentially annihilating the reflected COTOKEN.

Strings & Integers The pure λ-calculus Λ as well as the extended Λ_{eff} do not, by definition, support values such as strings or integers. While pure implementations could be constructed using encodings such as the Church encoding, for performance functional programming languages will normally use either the host language's, or another implementation that is closer to the hardware of the reducer.

Typically, functional programming languages such as Haskell will then declare built-in values as *pure*. However, when considering what using these native versions *means* from a low-level perspective, one comes to the conclusion that they are *instructions* to load certain values into registers or the memory. Specifically, having an ACTION load an integer into, say, a register, with a consecutive ACTION using this register in an `add` instruction, could require a notion of sequentiality that may only be guaranteed by the token-passing semantics of ξ-execution. The use and construction of these values can furthermore leak or depend on information about hardware internals such as register sizes or memory limitations, potentially causing unexpected behavior in a seemingly pure system.

We therefore declare the native versions of STRINGS and INTEGERS as *impure* and translate them to ACTIONS. Depending on implementation details, the implied sequentiality may not always be required. In this case, one may use the FORCE operator to trigger asynchronous execution as described in section 3.4. In the aspect of ξ-optimality this definition then also implies maximal sharing of the produced data. It also complements the interpretation provided in section 3.5, in that using such values requires *communication* with the REAL WORLD.

Since LET EXPRESSIONS resemble a syntactic sugar of λ-terms and the additional differences to the λ-calculus are minimal, we arrive at the following corollary without further proof.

Corollary 4.3 ($\mathcal{L}_{\text{sim}} \to \Lambda_{\text{eff}}$ preserves semantics)**.** *Let* $M, N \in \mathcal{L}_{\text{sim}}$*. If* $M \rightsquigarrow N$*, then* $[\![M]\!] \overset{\beta\xi\psi}{\rightsquigarrow\!\!\!\!\rightarrow} [\![N]\!]$.

Lemma ($\mathcal{L}_{\text{sim}} \to \Lambda_{\text{eff}}$ preserves unbound symbols)**.** *Given* $M \in \mathcal{L}_{\text{sim}}$, $\text{fv}(M) = \text{fv}([\![M]\!])$.

4.3 Recursion

If the name of a LET EXPRESSION occurs inside its body, the function may want to call itself. In this case, the translation function supplies an additional argument to the body representing the defined term. As it will otherwise be infinitely large, this additional argument is encoded in an ACTION and must therefore be closed as discussed in section 3.6:

$$[\![\textbf{let } f\ a_1\ a_2\ \ldots\ a_i = M\textbf{; } N]\!] = (\lambda f.[\![N]\!])\ ((\lambda f a_1 a_2 \ldots a_i.[\![M]\!])\ \text{box}[f]).$$

The boxed term therefore executes such that it wraps all previous definitions $a, b, \ldots$ and their bodies $M_a, M_b, \ldots$ around it recursively:

$$\text{box}[f] \triangleleft \overset{\xi}{\underset{\lrcorner}{\circlearrowleft}} ((\lambda ab \ldots .((\lambda f a_1 a_2 \ldots a_i.[\![M]\!])\ \text{box}[f]))\ [\![M_a]\!]\ [\![M_b]\!]\ \ldots) \triangleright.$$

As every box$[f]$ has no arguments (arity 0), it must first get expanded via TOKEN. This allows the use of higher-order functions as arguments. If the recursive ACTIONS would be of higher arity directly, their arguments would be required to be DATA. In order to be shared maximally, box$[f]$ may then always be written with a TOKEN such that it executes asynchronously. This has the obvious drawback of infinitely recursing reductions as long as no priority is given to other reductions.

As discussed in section 3.6, recursion can be optimized by guaranteeing the recursively expanded term to be in normal form up to execution of ACTIONS. In practice, this may be done by *pre-reducing* any box at compile time.

4.4 Applications of Forks

FORKS are essential when interacting with the REAL WORLD: While multiple TOKENS may be written explicitly in programs to denote asynchronous execution, it is otherwise not generally possible to parallelize two different sequential executions.

DISJUNCTIVE FORKS using `race` represent a paradigm commonly referred to as *angelic nondeterminism* [36]. It is named *angelic* because it always "chooses the optimal solution", i. e. the shortest *terminating* path when applied to nonterminating and terminating terms. This type of nondeterminism was first proposed by McCarthy [37] with the amb construct.

In our interpretation with TOKENS, `race` should return (1) the path which is reduced *and* executed fastest, and (2) only paths which eventually reflect a TOKEN.

Listing 5 Example use of `race`: Downloading an executable from multiple mirrors, such that—optimally—the fastest one is chosen automatically.

```
let angelicPong = race
  (download ("www1.example.com/pong.exe" !))
  (download ("www2.example.com/pong.exe" !))
  (download ("www3.example.com/pong.exe" !))
  (download ("www4.example.com/pong.exe" !)) !;

exec angelicPong !
```

In listing 5, once a COTOKEN is returned from one of the racing executions, all other terms are erased as defined in definition 3.3.

There is a caveat to this specification of `race`: The nondeterministic branches are not *guaranteed* to be executed in parallel—it is implementation-defined. In listing 5, this could then result in always choosing, say, the first mirror instead of the fastest one. Still, as a consequence of the necessity for returning COTOKENS, non-terminating paths are guaranteed to never be chosen as the "winner" of the race.

In comparison to `race`, the CONJUNCTIVE FORK `join` only becomes useful with a token-passing semantics: definition 3.3 implies that when a term is executed using the incoming TOKEN, it must also supply a function operating on the results of the FORK's two children. The TOKEN must therefore traverse into the argument of the additionally provided function.

In listing 6, we assume that token-passing similar to the one defined in chapter 7 exists.

Listing 6 Example use of `join`: Reading multiple files—potentially in parallel—and concatenating them.

```
let file = join
  (concat (readFile ("partA.txt" !)))
  (concat (readFile ("partB.txt" !)))
  (concat (readFile ("partC.txt" !)))
          (readFile ("partD.txt" !)) !;

writeFile ("full.txt" !) file !
```

The example program in listing 6 will pass a TOKEN to every branch, with

a token-passing semantics passing this TOKEN along to the respective `readFile` argument. Once all COTOKENS return, the resulting content is concatenated via

$$\texttt{concat}\ \langle A\rangle\ (\texttt{concat}\ \langle B\rangle\ (\texttt{concat}\ \langle C\rangle\ \langle D\rangle)).$$

4.5 Foreign Functions

ACTIONS are capable of producing arbitrary terms. One can imagine an implementation which allows loading code remotely using an ACTION producing its corresponding term representation. Of course, in asynchronous execution orthogonality has to be guaranteed. In listing 7, this is done by assuming the resulting term to be pure and supplying a checksum hash such that its evaluation can not depend on any state.

A more sophisticated version of this approach to distributed programming is implemented in the Unison programming language[1].

Listing 7 Loading a pure function asynchronously from a foreign location enables it to be used like any other pure function.

```
let fac = loadPure ("www.example.com/library
  ?name=fac
  &checksum=84de84e07449e1c7ad1f684f13154981" !) !;

-- note how fac does not require a '!'.
writeInt (fac (100 !)) !
```

Further, ACTIONS may be used to execute different programming languages and return the result as DATA. This way, foreign functions can be called directly in the language, as in listing 8.

Listing 8 Evaluating the factorial of a number by racing the implementations of multiple programming languages.

```
let fac n = race
  (evalLang ("python" !) ("math.factorial" !) n)
  (evalLang ("r"      !) ("factorial"      !) n)
  (evalLang ("julia"  !) ("factorial"      !) n) !;

writeInt (fac (100 !)) !
```

[1] `https://www.unison-lang.org/`, 07/06/2025

Here, the `evalLang` ACTION calls the interpreter of the respective language with the given function name and numeric argument. By assuming such calls to be pure, the `race` is executed asynchronously and `fac` may then be used without a TOKEN.

Chapter 5

Effectful Agents

In this chapter we introduce the effectful extensions described in chapter 3 to interaction nets. In section 5.2, we present the additional agents and their polarized variants. We go on to showcase the resemblance to Λ_{eff} by describing in the context of interaction nets the executions of ACTIONS, the concept of effectful currying, as well as the behavior of FORKS. We end the chapter by providing the specifics of the translation from Λ_{eff} and discussing various aspects of the presented extension of interaction nets.

5.1 Introduction

The effectful agents correspond to the additional terms of Λ_{eff}, thus extending Σ with an ACTOR, DATA, FORKS, and the TOKEN and COTOKEN.

Similar to how a token in an abstract machine can be used to track and control the state of reduction [30], the concept of a token within an *interaction net* is also not new: A token agent is used by Sinot [32] to reduce an encoded term using the Call-by-Name or Call-by-Value reduction strategy (depending on how the token is passed), which was later used to derive a Call-by-Need strategy [38]. In recent work by Salikhmetov [23], Sinot's token-passing is extended with a *waiting construct* which delays β-reduction in order to gain optimality in the sense of a bookkeeping oracle.

Related work on *graphs* instead of interaction nets includes the Geometry of Interaction (GOI) by Girard [39], which also uses a token-like agent traversing the graph in a specific path. The work is especially notable as it allows for derivations of (possibly) efficient abstract machines [40], which may be transferable to our work as well. Further research by Fernández; Mackie [41] embedded a Call-by-Value reduction strategy into such a machine.

5.2 Agents

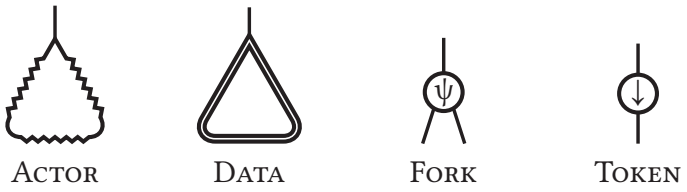


Figure 5.1 The effectful extension agents of Σ_{eff}.

The TOKEN agent travels through the net by interacting with certain agents and either continuing at one of their ports, or by getting *reflected*. The arrow always points to its principal port, which is the bottom in figure 5.1.

Through the interactions with DUPLICATOR and ERASER agents, DATA is effectively memory-managed: As described in section 3.5, DATA should be interpreted as being a reference to resources in the REAL WORLD. By tracking the interactions with DUPLICATORS and ERASERS, the referenced data in the REAL WORLD may be freed or otherwise removed once no such reference remains. For example, an interpreter may want to attach a *reference count* to memory-allocated data which gets incremented or decremented with the respective interactions. Once this count becomes zero, the data can be deallocated.

Similar to the agents of Σ_{pol}, the agents of Σ_{eff} can now be polarized to Σ_{peff}. This acts as a type system for the wiring and makes describing certain interactions such as annihilations of effectful agents irrelevant, as they cannot occur in well-typed nets.

Definition 5.1 (Agents of Σ_{eff} and Σ_{peff})**.** The system of Σ_{eff} extends Σ from definition 1.8 with agents representing the ACTION, DATA, FORKS, and TOKEN of Λ_{eff}. We write as $\mathcal{D}$ the set of agents encoding data of the REAL WORLD, and as Ξ the set of agents encoding actions. It is defined as

$$\Sigma_{\text{eff}} = \Sigma \cup \{\wedge_\circ, \vee_\circ, \triangleleft_\circ\} \cup \{\pi_\circ \mid \pi_\circ \in \mathcal{D}\} \cup \{\xi^i_{\mathcal{C}\circ} \mid \xi^i_{\mathcal{C}\circ} \in \Xi\},$$

with $\text{ar}(\xi^i_{\mathcal{C}\circ}) = \text{ar}(\pi_\circ) = 0$, $\text{ar}(\wedge_\circ) = \text{ar}(\vee_\circ) = 2$, and $\text{ar}(\triangleleft_\circ) = 1$.

Correspondingly, we also extend Σ_{pol} from definition 1.10 to Σ_{peff} by adding the polarized agents:

$$\Sigma_{\text{peff}} = \Sigma_{\text{pol}} \cup \{\wedge, \vee, \triangleleft, \triangleright\} \cup \{\pi \mid \pi \in \mathcal{D}\} \cup \{\xi^i_{\mathcal{C}} \mid \xi^i_{\mathcal{C}} \in \Xi\},$$

where $\xi^i_{\mathcal{C}} \equiv \xi^{i\,\oplus}_{\mathcal{C}\circ}$, $\pi \equiv \pi_\circ^{\oplus}$, $\wedge \equiv \wedge_\circ(x_\ominus, y_\ominus)^{\oplus}$, and $\vee \equiv \vee_\circ(x_\ominus, y_\ominus)^{\oplus}$.
The *directed* TOKENS also emerge from $\triangleleft_\circ$ by polarization: $\triangleleft \equiv \triangleleft_\circ(\text{cont}_\oplus)^{\ominus}$ and $\triangleright \equiv \triangleleft_\circ(\text{body}_\ominus)^{\oplus}$.

The agents Σ_{peff} of definition 5.1 are visualized in figure 5.2 with the addition of the agents dual to an ACTOR, DATA, and the FORKS. We discuss both FORKS as one agent, as they only differ in the way they handle returning COTOKENS.

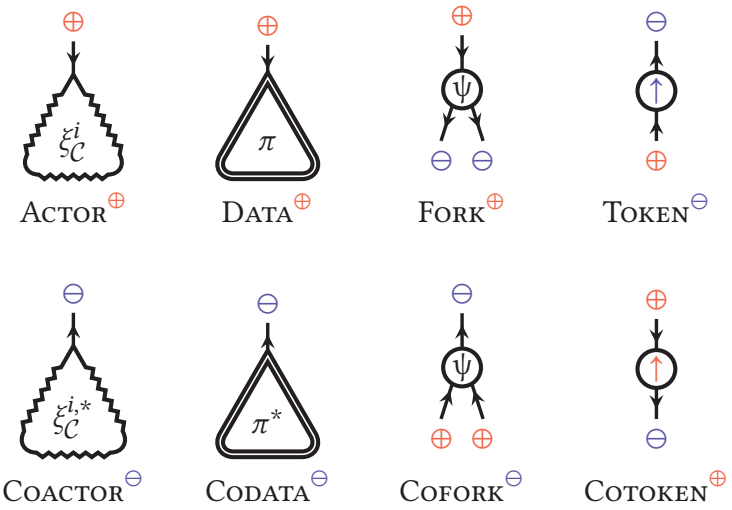


Figure 5.2 The polarized effectful agents.

ACTOR$^{\oplus}$ An ACTOR agent encoding ACTIONS of Λ_{eff}-terms $A \in a$, such as `read` or `write`. It produces either the resulting agents after execution ($\overset{\xi}{\dashrightarrow}$), or another ACTOR ($\overset{\xi}{\rightarrow}$).

DATA$^{\oplus}$ A DATA agent encoding data of Λ_{eff}-terms $D \in d$, such as $\langle\rangle$ or $\langle 42 \rangle$. It produces data consumable by ACTORS.

FORK$^{\oplus}$ A FORK agent produces the resulting term after it consumes the two forked terms. Its exact behavior depends on whether it is conjunctive or disjunctive.

TOKEN$^{\ominus}$ A TOKEN agent that is *heading* from the positive to the negative polarity. In an abstract sense it consumes DATA or the produced agents after it executed an ACTOR.

COTOKEN$^{\oplus}$ A TOKEN agent that is dual to TOKEN$^{\ominus}$. It is *returning* from the negative to the positive polarity.

Although there is no immediate use for the COACTOR, CODATA, and COFORK agents, we have presented them regardless. One could imagine their use coming from a different source language that has applications for such dualities: Since every agent has its dual equivalent, any presented interaction can also be expressed

in its dual world. For example, the dual ξ^*-reduction then coapplies CODATA to COACTORS via ABSTRACTORS, since ABSTRACTORS are dual to APPLICATORS.

By polarization of the effectful agents we have gained another property: The TOKEN always moves in the direction towards executing ACTORS, while the CO-TOKEN is always on the returning path—opposing the polarized direction of the wire. This is particularly important in the context of token-passing semantics we introduce in chapter 6 and chapter 7, as the order of execution is then—in a sense—prescribed by the passing of the TOKEN, which is in turn prescribed by polarization.

5.3 Executing Actors

The additional agents interact with the DUPLICATOR and ERASER as we have defined generically for Σ in figure 1.4.

When the downward-facing TOKEN interacts with an ACTOR of arity 0, the resulting net may encode any closed term of Λ_{eff}. The specific behavior of individual ACTORS is defined by the reducing host and can result in arbitrary interaction nets with a single free wire. This mirrors exactly the behavior of Λ_{eff} as described in chapter 3.

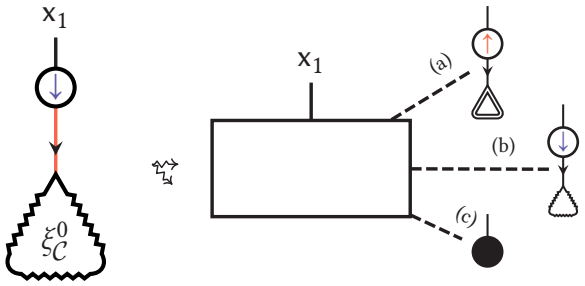


Figure 5.3 Execution of an effectful agent, resembling the ξ execution of Λ_{eff}.

Potential results of execution in figure 5.3 include:

(a) A DATA agent that *reflects* the TOKEN. This represents the expected behavior of ξ when an ACTION produces DATA. When the TOKEN does not get reflected, the entire net would contain one less TOKEN than before. This is used for *asynchronous actions*.

(b) An ACTOR agent that *bounces* the TOKEN. Since the produced ACTOR can behave differently on each bounce, this can be used for counting or delaying the bounces or reflection of the TOKEN.

(c) A COERASER that *consumed* the TOKEN. This can be used for panicking the current thread, as no further ACTORS can be executed by the consumed TOKEN and the COERASER starts erasing the net.

The resulting net may also be used for *recursion*, as described in section 3.6. The net would then encode the closed term required for the recursive call, where the free wire corresponds to the argument of the application. This approach is closely related to *references* used in HVM [4], where such agents may expand to externally specified nets as well.

ACTORS with non-zero arity i always reflect a TOKEN. The annihilation of TOKEN-COTOKEN interactions is preserved as well. Then, the behavior of Σ_{peff} mirrors Λ_{eff} perfectly:

$$\xi_{\mathcal{C}}^{0} \bowtie \triangleleft[\mathsf{M}] \qquad \xi_{\mathcal{C}}^{0} \triangleleft \overset{\xi}{\leftrightarrows} M$$

$$\xi_{\mathcal{C}}^{i} \bowtie \triangleleft[\triangleright(\xi_{\mathcal{C}}^{i})] \qquad \xi_{\mathcal{C}}^{i} \triangleleft \overset{\xi}{\rightsquigarrow} \xi_{\mathcal{C}}^{i} \triangleright$$

$$\triangleright[\mathsf{x}] \bowtie \triangleleft[\mathsf{x}] \qquad M\ (\triangleright\ \triangleleft) \overset{\xi}{\rightsquigarrow} M$$

$$(\triangleright\ \triangleleft)\ M \overset{\xi}{\rightsquigarrow} M$$

Compared to Λ_{eff}, nets encoding terms which may cause side effects are *by construction* only fully duplicated once they are unable to cause side effects. This is because the duplication happens iteratively via principal ports, therefore only duplicating asynchronous ACTIONS once they are no longer connected to a TOKEN. This happens analogously for DISJUNCTIVE FORKS, whereas this limitation was defined only informally for Λ_{eff} (see chapter 3).

5.4 Partial Application

Partial application in Σ_{peff} corresponds to the partial applications of Λ_{eff}, which we described in example 3.4. Specifically, just like for ACTIONS in ξ-reduction, ACTOR agents are *effectfully curried* and produce another ACTOR agent when applied to DATA.

Agents can only interact via their principal port. Formally, an ACTOR agent can thus not differentiate between its arguments when interacting with an APPLICATOR. We therefore introduce the temporary CURRY-ACTOR agent $\xi_{\mathcal{C}}^{i,c}$ that has two ports and only interacts with DATA:

$$\xi_{\mathcal{C}}^{i} \bowtie \alpha[\xi_{\mathcal{C}}^{i,c}(\mathsf{x}), \mathsf{x}] \text{ and } \xi_{\mathcal{C}}^{i,c}[\xi_{\mathcal{C}_{++}[\pi]}^{i-1}] \bowtie \pi \qquad \xi_{\mathcal{C}}^{i}\ \pi \overset{\xi}{\rightsquigarrow} \xi_{\mathcal{C}_{++}[\pi]}^{i-1}$$

The entire process of partial application is visualized in figure 5.4.

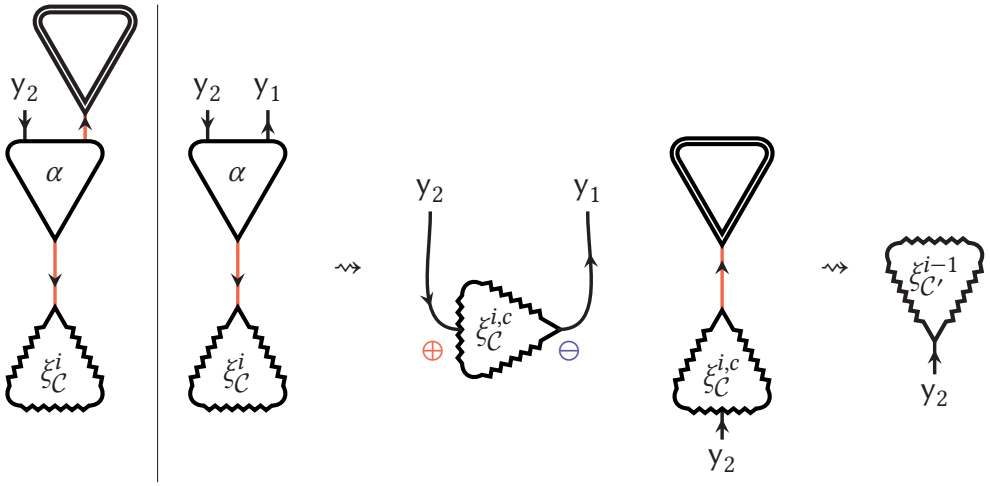


Figure 5.4 Partial application of an ACTOR, corresponding to $\overset{\xi}{\rightsquigarrow}$ of Λ_{eff}. Temporarily, the ACTOR is put into the curry-state to allow interaction with DATA.

This step is also necessary to keep the net well-typed: At their principal port, an ACTOR produces DATA, while a CURRY-ACTOR consumes DATA.

Note that implementations may diverge from the strict laws of interaction and polarization, and could instead interact with both the APPLICATOR and DATA directly (as shown on the left in figure 5.4).

5.5 Forks

The behavior of FORKS in Λ_{eff} as defined in definition 3.3 can be translated to Σ_{peff}. In general, the abstract interaction rules can be derived trivially from $\overset{\Downarrow}{\rightsquigarrow}$, as shown in figure 5.5. Compared to Λ_{eff}, the erasing in the DISJUNCTIVE FORKS is made explicit.

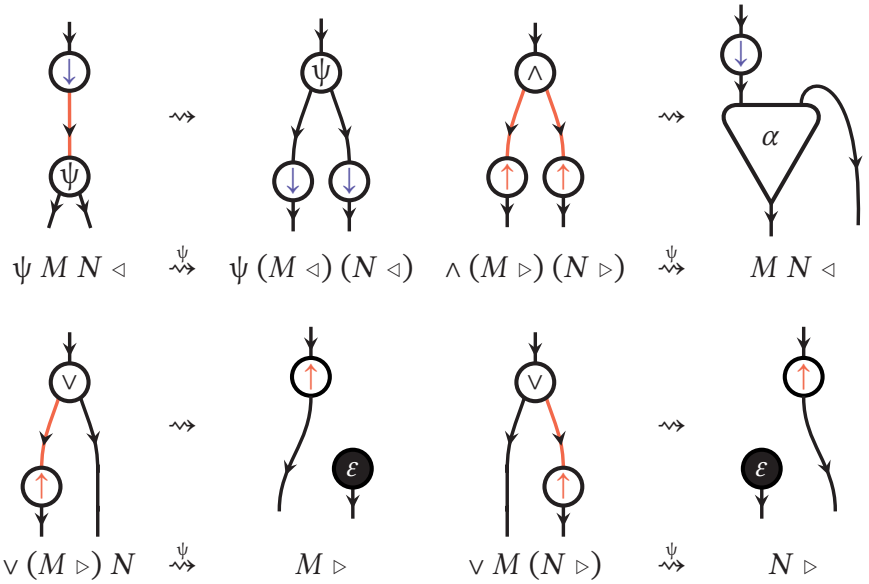


Figure 5.5 Abstract interactions of FORKS in Σ_{peff}, resembling the $\overset{\Downarrow}{\rightsquigarrow}$ reduction of Λ_{eff}. Here, $\psi \in \{\wedge, \vee\}$.

In figure 5.5 both the FORKS require interaction with *two principal ports*. In the conjunctive case, the interaction happens only when both TOKENS have returned as COTOKENS. In the disjunctive case, the interaction happens when either TOKEN returned as COTOKEN.

By its deterministic nature, the interactions of the CONJUNCTIVE FORK can be implemented with appropriate auxiliary agents. These auxiliary agents should effectively simulate an AND gate. We introduce the auxiliary *joining* agents $j_{1,2}$ and their activated versions $j^*_{1,2}$: When either $j_{1,2}$ agent interacts with a COTOKEN, it becomes activated. When two activated JOIN agents interact, they annihilate.

$$\begin{aligned} \wedge[\triangleright(j_1(x, \alpha(y, \triangleleft(z)))), \triangleright(j_2(x, y))] &\bowtie \triangleleft[z] \\ j_{1,2}[j_{1,2}(x, y), x] &\bowtie \triangleright[y] \\ j_1[x, x] &\bowtie j_2[y, y] \end{aligned}$$

Note how the agents $j_{1,2}$ differ only in the polarity of their left auxiliary port, since this is the port used to annihilate the activated JOIN agents. These auxiliary interactions are visualized in figure 5.6.

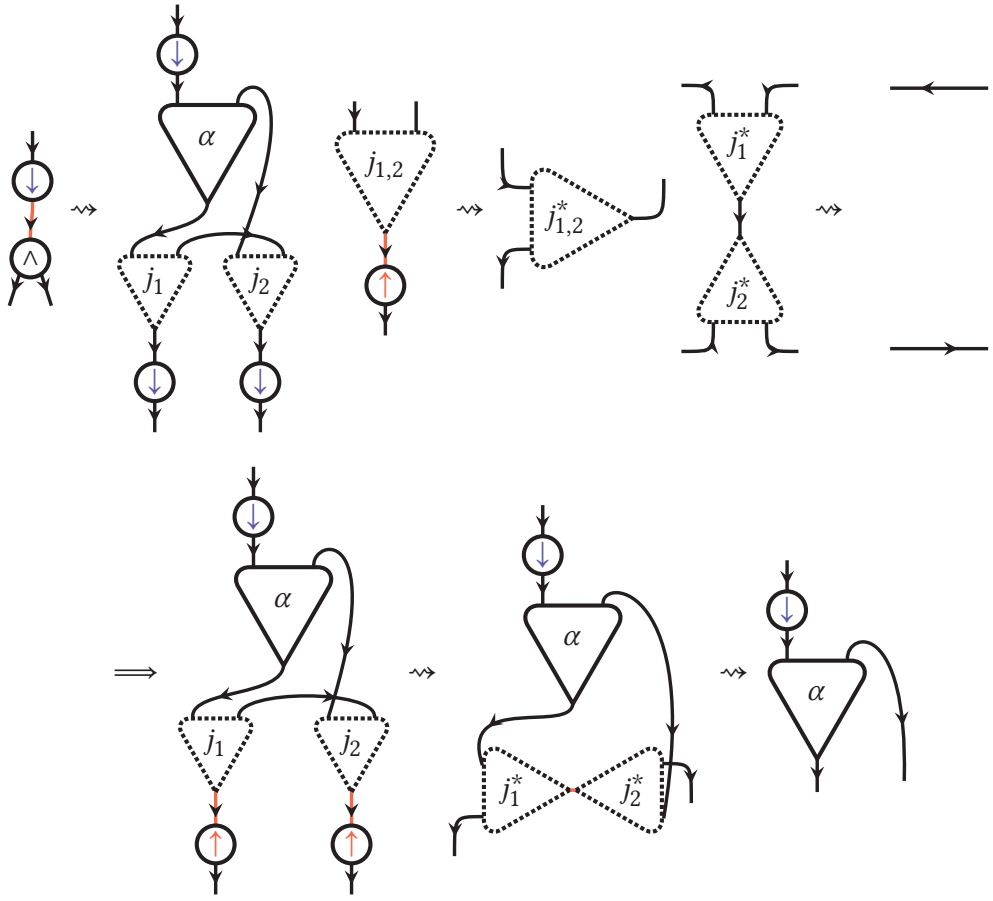


Figure 5.6 CONJUNCTIVE FORK interactions of Σ_{peff} using auxiliary agents $j_{1,2}$ and $j^*_{1,2}$. The rules are at the top, an example reduction is at the bottom.

In contrast to the CONJUNCTIVE FORKS, DISJUNCTIVE FORKS are nondeterministic. By the deterministic nature of interaction nets, they can not be implemented directly. Intuitively, this can be seen by imagining only the effect CONJUNCTIVE FORKS have on confluence: As long as either thread returns a COTOKEN, the net will still reduce to a normal form even if another thread reduces indefinitely. Therefore, in nets with DISJUNCTIVE FORKS, reduction paths may have different lengths and strong confluence is broken.

In consequence, DISJUNCTIVE FORKS require a nondeterministic extension of interaction nets. Such extensions have first been introduced in the PhD thesis of Alexiev [42]. Specifically, we use the extension allowing multiple principal ports ("INMPP"), which was further developed by Fernández; Khalil [36].

Upon interaction of $\vee$ with a TOKEN, it turns into the ambiguous agent $\vee^*$ with two principal ports. Interaction rules for this ambiguous agent can be defined using the syntax by Fernández; Khalil [36]. In this syntax, additional equations

are specified after the rule itself using $,\dots$. The state of the principal ports after interaction is specified as terms t_i on the left side of an agent via $[t_1, \dots, -, \dots, t_n]$, where "$-$" selects the port used for this interaction rule.

Then, the interactions seen in figure 5.5 are defined by

$$\begin{aligned} \vee[y, z] &\bowtie \triangleleft[x], [\triangleleft(y), \triangleleft(z)]\vee^*[x] \\ [-, \varepsilon]\vee^*[\triangleright(x)] &\bowtie \triangleright[x] \\ [\varepsilon, -]\vee^*[\triangleright(x)] &\bowtie \triangleright[x] \end{aligned}$$

5.5.1 Communication

In comparison to the λ-calculus, interaction nets allow connections between distinct terms. The Bend language defines these connections using *scopeless lambdas*[1]. Scopeless lambdas then allow a kind of global substitution, where a substitution to one variable has effects on an equivalently named variable in a completely different context.

In the context of FORKS, such wires between nets allow communication between threads. If the forked nets are instantiated with a connection between them, interactions may happen across this wire—thus exchanging information or DATA. We provide a proof-of-concept visualization in figure 5.7.

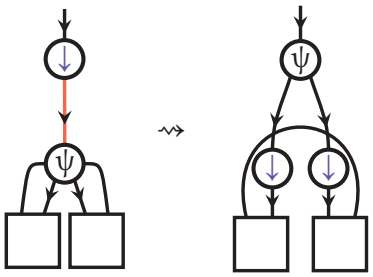


Figure 5.7 Wires between forked nets as a way of communication.

We conjecture that an appropriate syntax for this communication would have close relations to process calculi such as the π-calculus and the work of Jacobs [43]. The introduced wire would then represent a *channel*. The integration of such features into Λ_{eff} and Σ_{peff} remains future work.

[1] `https://github.com/HigherOrderCO/Bend/blob/d184863f03e796d1d657958a51dd6dd331ade92d/docs/using-scopeless-lambdas.md`, 07/09/2025

5.6 Translation

Definition 5.2 (Translation $\Lambda_{\text{eff}} \to \Sigma_{\text{peff}}$)**.** We extend definition 1.12 to the additional term kinds of Λ_{eff}:

$$\begin{aligned}
[\![M \triangleleft]\!]_{\mathsf{p}} &= \{\mathsf{m} == \triangleleft(\mathsf{p})\} \cup [\![M]\!]_{\mathsf{m}} \\
[\![\xi^i_{\mathcal{C}}]\!]_{\mathsf{p}} &= \{\mathsf{p} == \xi^i_{\mathcal{C}}\} \\
[\![\pi]\!]_{\mathsf{p}} &= \{\mathsf{p} == \pi\} \\
[\![\wedge\ M\ N]\!]_{\mathsf{p}} &= \{\mathsf{p} == \wedge(\mathsf{m}, \mathsf{n})\} \cup [\![M]\!]_{\mathsf{m}} \cup [\![N]\!]_{\mathsf{n}} \\
[\![\vee\ M\ N]\!]_{\mathsf{p}} &= \{\mathsf{p} == \vee(\mathsf{m}, \mathsf{n})\} \cup [\![M]\!]_{\mathsf{m}} \cup [\![N]\!]_{\mathsf{n}}
\end{aligned}$$

In order to trigger the token-passing semantics, it then makes sense to restate the configuration of a program $M \in \Lambda_{\text{eff}}$ as $\langle |\ \iota == \mathsf{p}, \mathsf{p} == [\![M \triangleleft]\!]_{\mathsf{p}} \rangle$. This can also be seen as the result of the interaction of an arbitrary agent and the INTIATOR sending out a TOKEN. It further allows eliminating the final returning COTOKEN:

$$\iota \bowtie \triangleright[\iota]$$

Example 5.3. We translate $M = \texttt{write}^1_\emptyset\ (\texttt{mult}^2_\emptyset\ \langle 21 \rangle\ \langle 2 \rangle \triangleleft) \triangleleft$ to Σ_{peff} using $[\![M]\!]_\iota$. We assume that `mult` can be executed asynchronously (i. e. is orthogonal) and does not reflect a COTOKEN. We can then reduce it until everything is executed and the TOKEN returns as COTOKEN.

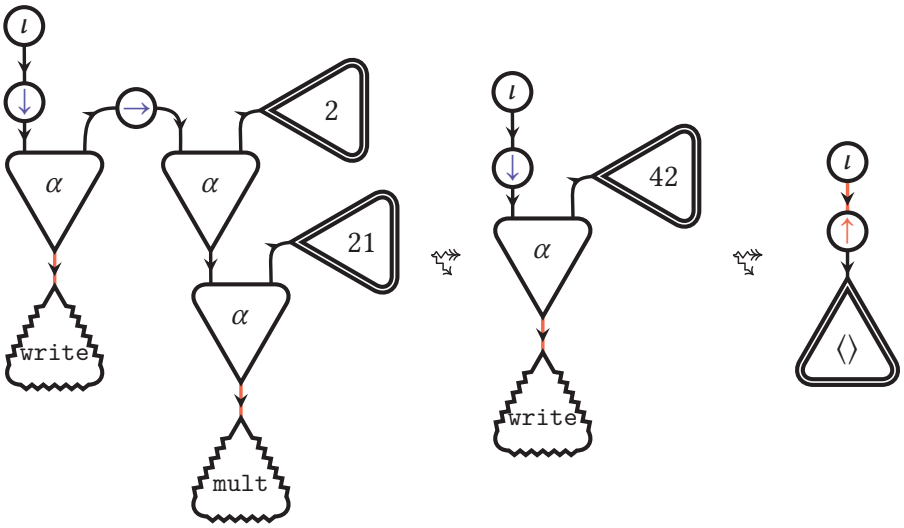


Figure 5.8 Example translation from Λ_{eff} to Σ_{peff} and its reduction.

With all the translations and rules described, we can extend the rules of Σ_{pol} with the rules of Σ_{peff}.

Definition 5.4 (Interaction Rules of Σ_{peff})**.** We extend the interaction rules of Σ_{pol} from definition 1.11:

$$\begin{aligned}
\iota &\bowtie \triangleright[\iota] \\
\xi_{\mathcal{C}}^{i} &\bowtie \alpha[\xi_{\mathcal{C}}^{i,c}(\mathsf{x}), \mathsf{x}] \\
\xi_{\mathcal{C}}^{i,c}[\xi_{\mathcal{C}+\!\!+[\pi]}^{i-1}] &\bowtie \pi \\
\xi_{\mathcal{C}}^{0} &\bowtie \triangleleft[\mathsf{M}] \\
\xi_{\mathcal{C}}^{i} &\bowtie \triangleleft[\triangleright(\xi_{\mathcal{C}}^{i})] \\
\triangleright[\mathsf{x}] &\bowtie \triangleleft[\mathsf{x}] \\
\wedge[\triangleright(j_1(\mathsf{x}, \alpha(\mathsf{y}, \triangleleft(\mathsf{z})))), \triangleright(j_2(\mathsf{x}, \mathsf{y}))] &\bowtie \triangleleft[\mathsf{z}] \\
j_{1,2}[j_{1,2}(\mathsf{x}, \mathsf{y}), \mathsf{x}] &\bowtie \triangleright[\mathsf{y}] \\
j_1[\mathsf{x}, \mathsf{x}] &\bowtie j_2[\mathsf{y}, \mathsf{y}] \\
\vee[\mathsf{y}, \mathsf{z}] &\bowtie \triangleleft[\mathsf{x}], [\triangleleft(\mathsf{y}), \triangleleft(\mathsf{z})]\vee^*[\mathsf{x}] \\
[-, \varepsilon]\vee^*[\triangleright(\mathsf{x})] &\bowtie \triangleright[\mathsf{x}] \\
[\varepsilon, -]\vee^*[\triangleright(\mathsf{x})] &\bowtie \triangleright[\mathsf{x}]
\end{aligned}$$

We do not view the auxiliary agents $\xi_{\mathcal{C}}^{i,c}$, $j_{1,2}$, $j_{1,2}^*$, or $\vee^*$ as part of Σ_{peff} as they are only necessary to conform to the laws of interaction nets and may be exchanged with arbitrary agents.

5.7 Evaluation

In order to derive comparisons between the further variants described in chapter 6 and chapter 7, we evaluate baseline benchmarks of Σ_{peff} using the provided translations from $\mathcal{L}_{\text{sim}}$ and Λ_{eff}.

5.7.1 Approach

In our benchmarks, time by itself is not a relevant metric. We view interactions as being bounded by a constant time ($\mathcal{O}(1)$), so instead only measure the *number* of interactions. Bounds on time may then be derived from this number. Specifically, we differentiate between two counts:

Interactions The total number of active pairs required to be reduced until no active pair is left. We also categorize the number of iterations by the respective interaction rules (duplication, annihilation, erasing, effectful). In a strongly confluent system, these numbers are always constant.

Iterations In every *iteration* of a net, each active pair with a matching interaction rule is reduced *once*, without reducing the potentially resulting active pairs. The measure of iterations is equivalent to the measure of "available parallelism" analyzed in existing work [44, 45]: In contrast to only counting interactions, iterations also measure to which extent the reduction can benefit from parallelism, as every iteration reduces all current active pairs in a single, parallel step.

The parallel reduction is only possible because of locality, as the interactions can not interfere with each other. In a strongly confluent system, this number is always constant.

We further calculate the *ratio* of interactions per iterations, such that larger numbers hint towards more productivity when reducing in parallel.

In our results we pre-reduce all recursive calls as discussed in section 4.3. We do not garbage collect detached nets externally even when it could reduce the interaction counts without changing the correctness. These restrictions can be reproduced as described in appendix A via `cat <file> | opteff-exe -c -t monad`.

We also compare against the Bend language using the Higher-Order Virtual Machine discussed in chapter 2. We use the HVM version 2.0.22 together with the Bend version 0.2.38. We run the programs using
`bend run -Ono-all -Ofloat-combinators -Olinearize-matches -s <file>`.

5.7.2 Results

As our system encodes the *abstract algorithm* (section 1.3.6), we can only evaluate a certain subset of Λ_{eff} [21]. Here we have the further limitation of no token-passing semantics, therefore requiring effectful terms to either be asynchronous or linearly dependent on each other.

We use a very restricted set of programs, which is defined in `bench/nopassing/` (see appendix A). As a TOKEN can not enter APPLICATORS, all recursive calls are tail-recursive.

				Rules				Bend
	Inter.	Iter.	Ratio	Dup.	Ann.	Era.	Eff.	Inter.
`fac10_tail`	284	134	2.119	30	111	28	115	203
`fac20_tail`	554	264	2.098	60	221	48	225	413
`fib10_tail`	304	127	2.394	33	133	32	106	278
`fib20_tail`	564	237	2.380	63	253	52	196	538
`collatz24`	541	148	3.655	44	210	63	224	304
`collatz25`	1152	317	3.631	96	457	128	471	681

Table 5.1 The interaction counts and iterations required to reduce the programs in `bench/nopassing/` to their normal form in Σ_{peff}.

As expected from their related structure and argument sizes, the results in table 5.1 are very similar to each other. Together with the fact that the interactions and iterations appear to scale linearly with more iterations, these preliminary results indicate that our approach has no surprising overhead. This observation is further enforced by the interaction counts of the Bend programming language. The difference in interaction counts to Bend most probably stems from our requirements for partial application and TOKEN interactions. Arithmetic is not seen as effectful in HVM and is implemented via dedicated agents.

In figure 5.9 we plot the available parallelism per iteration of `fac10_tail` against the same program *without pre-reducing* recursive calls. This implies larger spikes in each iteration of the program, as more reduction work is repeated.

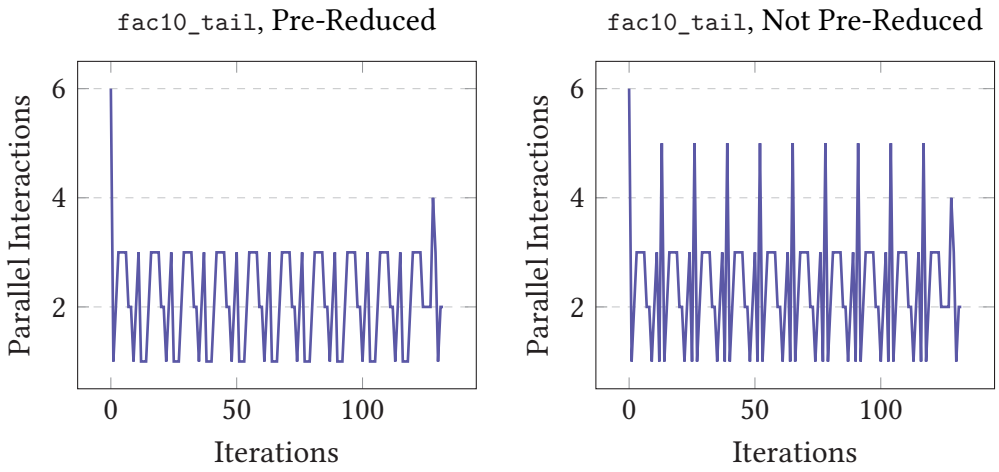


Figure 5.9 Available parallelism per iteration during the reduction of `fac10_tail`, compared to its version without pre-reduced recursive calls.

5.8 Discussion

Aside from the nondeterministic extension for the DISJUNCTIVE FORK, we have not diverged from the laws of interaction nets proposed by Lafont [1]. We therefore arrive at the following as a consequence of theorem 1.7 and corollary 3.7.

Corollary 5.5 (Strong Confluence of Orthogonal $\leftrightarrows$)**.** *Assuming all ACTORS to be orthogonal, the reduction relation* $\leftrightarrows$ *of* $\Sigma_{\text{peff}} \setminus \{\vee\}$ *satisfies the diamond property modulo time of execution.*

The translation to Σ_{peff} preserves the aspects required for (effectful) optimality. For one, by definition the DUPLICATOR agents always also reduce the net iteratively, thus sharing maximally. This sharing includes the execution of asynchronous ACTIONS as they are only duplicated once they have been executed—as required by section 3.7.

Techniques for preventing redundant reductions also remain: Without ACTIONS or TOKENS, strictly sequential reduction may be used in addition to garbage collecting detached nets externally as described in section 1.3.5. With the addition of effectful agents, even parallel reduction may be used since detached, unreduced nets are not generally deemed to be redundant (see section 3.7). In the aspect of minimizing redundant reductions heuristically, any detached net turning out not to contain a TOKEN or COTOKEN can still be garbage collected externally without requiring iterative erasing. By viewing side effects as the only relevant part of a program's evaluation, this heuristic spans general nets encoding a program, as it can effectively be garbage collected once it is incapable of causing side effects.

In comparison to existing solutions of adding side effects to interaction nets discussed in chapter 2, our technique does not have limitations on the number of arguments, does not require halting of reductions in order to execute ACTIONS, is capable of producing arbitrary nets via side effect, and supports reduction in parallel to execution.

Our preliminary results indicate availability of parallelism even when sequentializing the execution by linear dependencies. This parallelism includes asynchronous ACTIONS, β-reduction, partial application of ACTORS, and garbage collection.

Chapter 6

Monadic Style

In this chapter we introduce a token-passing semantics for both the λ-calculus and interaction nets by extending them with monadic constructs.

Monads offer an elegant solution to passing a context along computations sequentially without having to write this passing of context explicitly. The concept has its roots in category theory, but was later applied to abstract programming languages by Moggi [26]. To us, monads may be imagined as carrying not some context, but the Real World in its entirety. Therefore, all operations using a version of the Real World will also be run sequentially. This solution to the problem of sequentiality was proposed for the Haskell programming language by Peyton Jones; Wadler [27]. There, the Real World is implicitly carried via the type system, thus giving a guarantee of sequential execution.

In interaction nets, the use of agents to simulate the behavior of various kinds of monads has first been explored in the PhD thesis of Jiresch [45]. Our agents are related to their "state transformer" agents, but solely interact to redirect the TOKEN agent described in chapter 5 in a way that it represents the implicitly and sequentially passed context of the monad.

In section 6.1 we briefly introduce monads in the context of side effects. In the first part of this chapter we then describe the monadic extension of the effectful λ-calculus alongside its semantics. In the other part we introduce its equivalent representation in interaction nets as well as its translations. We conclude with discussing the monadic style and evaluating its feasibility.

6.1 Monads

For the way in which we make use of monads, we introduce them only as a high-level solution to adding sequentiality to functional programming [27].

A monad consists of two functions `bind` and `return` (or `unit`). A *monadic*

action represents a wrapper M around some value x of type a. This wrapper may be interpreted as carrying the context of the value x. The `bind` function binds a monadic action to a function turning the unwrapped value a into another monadic action $M\ b$—the *continuation*. In order to create such a monadic action from pure terms, the `return` function wraps terms into a monadic action.

In a typed language such as Haskell, these functions can therefore be written as the following:

$$\begin{aligned} &\texttt{bind}\ mx\ f :: M\ a \to (a \to M\ b) \to M\ b \\ &\texttt{return}\ x \quad\ :: M\ a \end{aligned}$$

In order for a system to be considered a monad, it must satisfy a set of *monad laws* (theorem 6.4).

In the case of side effects, we take as inspiration the `IO` monad of Haskell as described by Peyton Jones; Wadler [27]. The core idea behind it is that any ACTION requiring synchronization with the REAL WORLD may be interpreted as a function turning the REAL WORLD into another state of the REAL WORLD attached to a value. For example:

$$\begin{aligned} &\texttt{readInt} \quad\ :: \texttt{RealWorld} \to (\texttt{Int}, \texttt{RealWorld}) \\ &\texttt{writeInt}\ n :: \texttt{Int} \to \texttt{RealWorld} \to ((), \texttt{RealWorld}) \end{aligned}$$

With corresponding definitions of `bind` and `return` for the type $M = \texttt{IO}$, $\texttt{IO}\ a = \texttt{RealWorld} \to (a, \texttt{RealWorld})$, the following then writes an integer it has read:

$$\texttt{bind readInt writeInt} \leftrightsquigarrow \texttt{writeInt}\ \langle 42 \rangle \leftrightsquigarrow \langle\rangle\,,$$

where we hide the implicit passing of `RealWorld` for better readability.

Importantly, this way of passing state *guarantees* a strict sequence of execution, as `RealWorld` may only be passed to the next function if the current execution has finished.

6.2 Language Extension

It is common for a language making use of monads to introduce a dedicated syntax, so programmers do not have to write all `binds` and `returns` explicitly. One of such syntaxes is shown in listing 1 for the Bend language. In an extension to $\mathcal{L}_{\text{sim}}$, we instead opt for a syntax similar to Haskell's *do notation* [28].

Definition 6.1 (Terms of $\mathcal{L}_{\text{mon}}$)**.** We extend the language $\mathcal{L}_{\text{sim}}$ from definition 4.1 with a monadic notation for sequentializing actions.

The terms of the monadic language $\mathcal{L}_{\text{mon}}$ are:

t	$::= \cdots$	Terms of $\mathcal{L}_{\text{sim}}$
	$\mid$ **do** t	Do Expression
	$\mid x \leftarrow t; t$	Monadic Bind
	$\mid$ **return** t	Monadic Return
x	$\in$ x, num, add, ...	Name

Using this extension, sequentialization does not require a custom encoding by data dependencies as before. Instead, the sequence is encoded directly in the monadic binds. As an implication, there is also no need for asynchronous execution via force.

Listing 9 An example program in $\mathcal{L}_{\text{mon}}$. Note how *all* synchronous actions—even `add`—must be written in this notation to be executed.

```
let staticTwo = do (
  return (2 !)
);

do (
  a   ← readInt ⟨⟩;
  b   ← staticTwo;
  res ← add a b;
  writeInt res
)
```

If this specific implementation of `add` is applicable to asynchronous execution as argued in section 3.4, one may still write `writeInt (add a b !)` directly. We assumed such asynchrony for loading the integer 2.

6.3 Semantics

Since our introduced system relies on tokens to execute actions and forks, we replace the concept of passing the Real World implicitly with the concept of passing the token.

The semantics of $\mathcal{L}_{\text{mon}}$ follow the semantics of $\mathcal{L}_{\text{sim}}$ (and therefore Λ) up to this additional token-passing. We argue about its semantics using a monadic extension of Λ_{eff}.

Definition 6.2 (Terms of Λ_{mon})**.** We extend the effectful λ-calculus Λ_{eff} from definition 3.1 with *bind* and *return* terms.

The terms of the monadic λ-calculus Λ_{mon} are:

$$\begin{array}{lll} t ::= \cdots & & \textsc{Terms of } \Lambda_{\text{eff}} \\ \mid (t \gg\!= t) & & \textsc{Monadic Bind} \\ \mid (\eta\, t) & & \textsc{Monadic Unit} \end{array}$$

We define the monadic token-passing semantics using the *explicit* TOKEN, *implicit* in $\mathcal{L}_{\text{mon}}$. We are not aware of similarly minimal notations for the execution order of IO monads.

Definition 6.3 (Token-Passing Semantics of Λ_{mon})**.** The token-passing $\overset{\triangle}{\rightsquigarrow}$ of Λ_{mon} is described by the following rewrite rules:

$$\begin{aligned} (M \gg\!= N) \triangleleft &\overset{\triangle}{\rightsquigarrow} (M \triangleleft) \gg\!= N \\ (M \triangleright) \gg\!= N &\overset{\triangle}{\rightsquigarrow} N\ M \triangleleft \\ (\eta\, M) \triangleleft \quad &\overset{\triangle}{\rightsquigarrow} M \triangleright \end{aligned}$$

This token-passing semantics indeed matches the behavior of monads exactly: The TOKEN continuing in M when being applied to a bind $(M \gg\!= N)$ can be seen as initiating the unwrapping from $M\ a$ to a. When the TOKEN returns, the resulting a is applied to N with type $a \to M\ b$. Therefore, the resulting $M\ b$ will also have to be applied to a TOKEN to access its wrapped value b, and equivalently for the MONADIC UNIT η.

Notably, the TOKEN only returns as COTOKEN in M if it is either applied to a UNIT, or by being reflected from an ACTOR or FORK. Assuming common typing rules such as Haskell's [27], a TOKEN will never be applied to an ABSTRACTION or APPLICATION in (weak) normal form and can therefore never get "stuck" or cause overlapping rewrite rules. A term such as $(M \triangleright) \gg\!= N \overset{\triangle}{\rightsquigarrow} N\ M \triangleleft$ will necessarily imply that (1) N converges to an ABSTRACTION, and (2) that $N', (N\ M) \downarrow N'$, if existing, is either a MONADIC UNIT, a MONADIC BIND, a FORK, or an ACTION.

Theorem 6.4 (Monad Laws of Λ_{mon})**.** *The token-passing semantics* $\overset{\triangle}{\rightsquigarrow}$ *of* Λ_{mon} *follow the monad laws.*

$$\begin{array}{lll} (\eta\, M) \gg\!= N & \overset{\triangle}{\equiv} N\ M & \textsc{Left Identity} \\ M \gg\!= \lambda x.(\eta\, x) & \overset{\triangle}{\equiv} M & \textsc{Right Identity} \\ (M \gg\!= N) \gg\!= O & \overset{\triangle}{\equiv} M \gg\!= \lambda x.((N\ x) \gg\!= O) & \textsc{Associativity} \end{array}$$

Where $M \overset{\triangle}{\equiv} N$ *iff there exists a common term* C*, such that* $(M \triangleleft) \overset{\triangle}{\rightsquigarrow} C$ *and* $(N \triangleleft) \overset{\triangle}{\rightsquigarrow} C$*.*

Proof. By reduction:

Left Identity	Right Identity
$((\eta\ M) \gg\!= N)\ \triangleleft$	$(M \gg\!= \lambda x.(\eta\ x))\ \triangleleft$
$\overset{\triangle}{\rightsquigarrow}\ ((\eta\ M)\ \triangleleft) \gg\!= N$	$\overset{\triangle}{\rightsquigarrow}\ (M\ \triangleleft) \gg\!= \lambda x.(\eta\ x)$
$\overset{\triangle}{\rightsquigarrow}\ (M\ \triangleright) \gg\!= N$	$\rightsquigarrow (M'\ \triangleright) \gg\!= \lambda x.(\eta\ x)\ \circledast$
$\overset{\triangle}{\rightsquigarrow}\ N\ M\ \triangleleft$	$\overset{\triangle}{\rightsquigarrow}\ (\lambda x.(\eta\ x))\ M'\ \triangleleft$
	$\overset{\beta}{\rightsquigarrow}\ (\eta\ M')\ \triangleleft$
	$\overset{\triangle}{\rightsquigarrow}\ M'\ \triangleright \rightsquigarrow M\ \triangleleft\ \circledast$

Associativity

$((M \gg\!= N) \gg\!= O)\ \triangleleft$	$(M \gg\!= (\lambda x.((N\ x) \gg\!= O)))\ \triangleleft$
$\overset{\triangle}{\rightsquigarrow}\ ((M \gg\!= N)\ \triangleleft) \gg\!= O$	$\overset{\triangle}{\rightsquigarrow}\ (M\ \triangleleft) \gg\!= \lambda x.((N\ x) \gg\!= O)$
$\overset{\triangle}{\rightsquigarrow}\ ((M\ \triangleleft) \gg\!= N) \gg\!= O$	$\rightsquigarrow (M'\ \triangleright) \gg\!= \lambda x.((N\ x) \gg\!= O)\ \circledast$
$\rightsquigarrow ((M'\ \triangleright) \gg\!= N) \gg\!= O\ \circledast$	$\overset{\triangle}{\rightsquigarrow}\ (\lambda x.((N\ x) \gg\!= O))\ M'\ \triangleleft$
$\overset{\triangle}{\rightsquigarrow}\ (N\ M'\ \triangleleft) \gg\!= O$	$\overset{\beta}{\rightsquigarrow}\ ((N\ M') \gg\!= O)\ \triangleleft$
	$\overset{\triangle}{\rightsquigarrow}\ (N\ M'\ \triangleleft) \gg\!= O$

□

In the steps marked with $\circledast$, we assume the TOKEN returns after an arbitrary number of reduction steps. This assumption must not always be true, since ACTIONS are not forced to return a TOKEN after execution, or the TOKEN could be applied to terms for which we have not defined the reduction behavior. Yet it *should* be true for all monadic ACTIONS: Without returning a TOKEN, the monadic chain would be broken and would not reduce further by $\overset{\triangle}{\rightsquigarrow}$. Ultimately, this rule shall also be enforced by a type system or the programmer.

We derive the following based on the results of Sinot [32] and the described semantics of Λ_{mon}:

Proposition 6.1 (Return of the Token)**.** *Assuming that all ACTIONS reflect a TOKEN after execution, a TOKEN-free term $M \in \Lambda_{\text{mon}} \setminus \{\triangleleft, \triangleright\}$ fulfilling the monadic typing rules applied to a TOKEN always either diverges, or reduces to a term applied to a COTOKEN:*

$$M \downarrow M' \iff M \triangleleft \overset{\beta\xi\psi\triangle}{\rightsquigarrow} M' \triangleright$$

6.4 Monadic Agents

The additional term kinds of Λ_{mon} can be equivalently introduced to interaction nets by extending Σ_{eff} and Σ_{peff}.

Definition 6.5 (Agents of Σ_{mon} and Σ_{pmon})**.** The system of Σ_{mon} extends Σ_{eff} from definition 5.1 with agents representing the MONADIC BIND and UNIT of Λ_{mon}:

$$\Sigma_{\text{mon}} = \Sigma_{\text{eff}} \cup \{\gg=_\circ, \eta_\circ\},$$

with $\text{ar}(\gg=_\circ) = 2, \text{ar}(\eta_\circ) = 1$.

Correspondingly, we also extend Σ_{peff} from definition 5.1 to Σ_{pmon} by adding the polarized agents: The MONADIC BIND is polarized as two different agents, with one being its *executive* state:

$$\Sigma_{\text{pmon}} = \Sigma_{\text{peff}} \cup \{\gg=, \gg=^*, \eta\},$$

where $\gg= \equiv \gg=_\circ(\text{arg}_\ominus, \text{cont}_\ominus)^\oplus$, $\gg=^* \equiv \gg=_\circ(\text{cont}_\ominus, \text{ret}_\oplus)^\ominus$, and $\eta = \eta_\circ(\text{cont}_\ominus)^\oplus$.

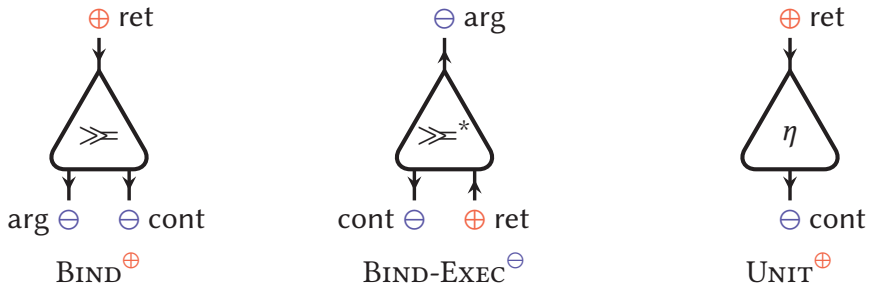


Figure 6.1 The polarized monadic agents of Σ_{pmon}.

The requirement for BIND-EXEC becomes clear with the following observation: When a monadic BIND gets executed, it gets applied to a TOKEN. When this TOKEN consequently continues its traversal in its left auxiliary port (the *argument*), there is no way of interacting with the returning COTOKEN. In contrast, by replacing this BIND agent with a BIND-EXEC agent, the COTOKEN will interact with the principal port.

The rest of the monadic agents interact by extending the rules of Σ_{peff} from definition 5.4 and mirroring the exact behavior of the described monadic token-passing semantics of Λ_{mon}:

$$\gg=[\triangleright(\gg=^*(y, x)), y] \bowtie \triangleleft[x] \qquad (M \gg= N) \triangleleft \overset{\triangle}{\rightsquigarrow} (M \triangleleft) \gg= N$$

$$\gg=^*[\alpha(x, \triangleleft(y)), y] \bowtie \triangleright[x] \qquad (M \triangleright) \gg= N \overset{\triangle}{\rightsquigarrow} N\ M \triangleleft$$

$$\eta[x] \bowtie \triangleleft[\triangleright(x)] \qquad (\eta\ M) \triangleleft \overset{\triangle}{\rightsquigarrow} M \triangleright$$

In figure 6.2 the interactions are annotated with the respective rewrite rules $\overset{\triangle}{\rightsquigarrow}$ of Λ_{mon}.

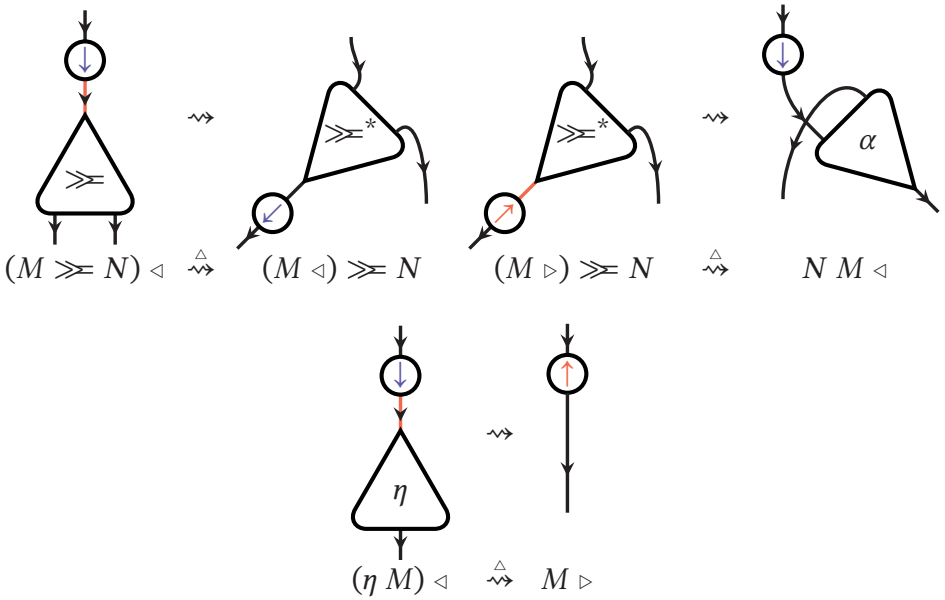


Figure 6.2 Interactions of Σ_{pmon}, resembling the $\overset{\triangle}{\rightsquigarrow}$ reduction of Λ_{mon}.

6.5 Translations

The monadic extension of $\mathcal{L}_{\text{sim}}$ corresponds exactly to Λ_{mon} which can then be translated to Σ_{pmon}.

Definition 6.6 (Translation $\mathcal{L}_{\text{mon}} \to \Lambda_{\text{mon}}$)**.** We extend the translation $\mathcal{L}_{\text{sim}} \to \Lambda_{\text{eff}}$ from definition 4.2 to the additional terms on both sides:

$$\begin{aligned}
[\![\textbf{do}\ M]\!] &= [\![M]\!] \\
[\![x \leftarrow M; N]\!] &= [\![M]\!] \gg\!= \lambda x.[\![N]\!] \\
[\![\textbf{return}\ M]\!] &= \eta\ [\![M]\!]
\end{aligned}$$

As described in definition 5.2, the outermost term M of a program is applied to the TOKEN $M \triangleleft$ in order to trigger the token-passing semantics.

Definition 6.7 (Translation $\Lambda_{\text{mon}} \to \Sigma_{\text{pmon}}$)**.** We extend definition 5.2 to the additional term kinds of Λ_{mon}:

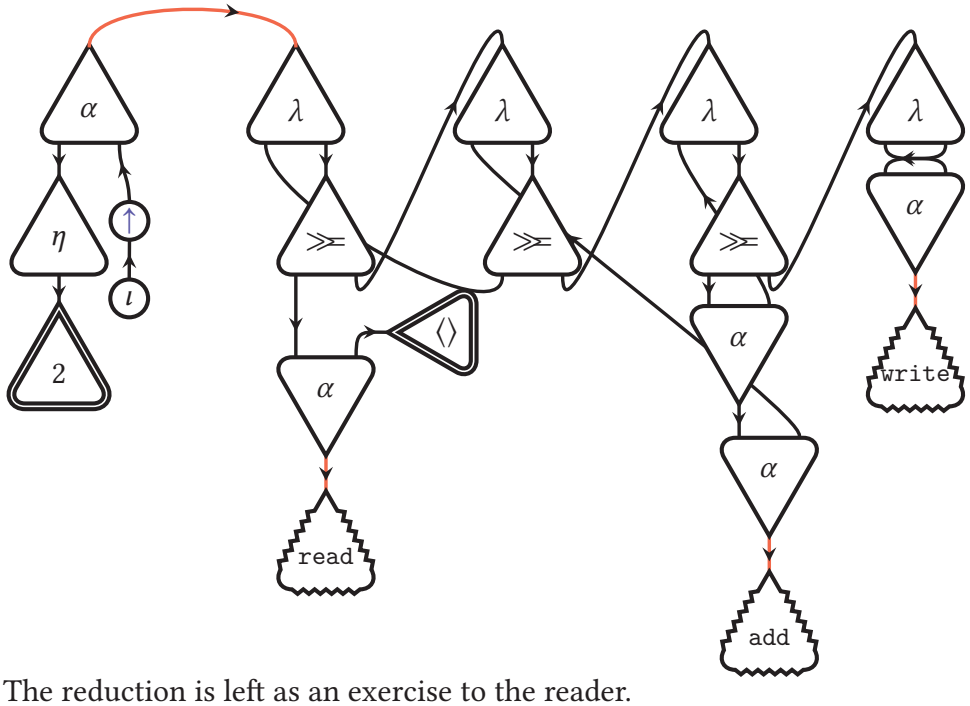
$$\begin{aligned}
[\![M \gg\!= N]\!]_{\mathsf{p}} &= \{\mathsf{p} :== \gg\!=(\mathsf{m}, \mathsf{n})\} \cup [\![M]\!]_{\mathsf{m}} \cup [\![N]\!]_{\mathsf{n}} \\
[\![\eta\ M]\!]_{\mathsf{p}} &= \{\mathsf{p} :== \eta(\mathsf{m})\} \cup [\![M]\!]_{\mathsf{m}}
\end{aligned}$$

Example 6.8 (Example monad translation)**.** We continue listing 9 by translating it to Λ_{mon} and Σ_{pmon}. We have resolved the asynchronous ACTIONS in the interaction net for better readability.

```
let staticTwo = do (
  return (2 !)
);

do (
  a   ← readInt ⟨⟩;
  b   ← staticTwo;
  res ← add a b;
  writeInt res
)
```

$$
\begin{aligned}
&(\lambda staticTwo.\\
&\quad ((\texttt{readInt}^{1}_{\emptyset}\ (\texttt{unit}^{0}_{[\langle\rangle]}\ \triangleleft)) \gg\!= \lambda a.\\
&\qquad (staticTwo \gg\!= \lambda b.\\
&\qquad\quad ((\texttt{add}^{2}_{\emptyset}\ a\ b) \gg\!= \lambda res.\\
&\qquad\qquad \texttt{writeInt}^{1}_{\emptyset}\ res))))\\
&\quad (\eta\ (\texttt{two}^{0}_{[\langle\rangle]}\ \triangleleft))
\end{aligned}
$$



The reduction is left as an exercise to the reader.

6.6 Recursion

The monadic style permits recursion via side effects as defined in section 3.6 and section 4.3. Recursive calls are encoded in ACTIONS producing the term resembling the original term. If we use a recursive call to a monadic term, it will then have to be executed *twice*.

Example 6.9 (REPL in $\mathcal{L}_{\text{mon}}$)**.** `repl` defines a recursive program that continuously prompts the user for two numbers and prints their sum.

```
let repl = do (
  _ ← print ("Enter two numbers!" !);
  x ← readInt ⟨⟩;
  y ← readInt ⟨⟩;
  _ ← writeInt (add x y !);
  r ← repl;
  r
);
repl
```

Note how the term expansion of `repl` first gets loaded into `r` monadically which is then executed again to repeat the loop.

6.7 Evaluation

We continue the evaluation of $\mathcal{L}_{\text{sim}}$ and Σ_{peff} in section 5.7 by running equivalent programs in $\mathcal{L}_{\text{mon}}$ and Σ_{pmon}. In contrast to the previous benchmarks, the use of asynchronous ACTIONS is no longer required. The monadic token-passing semantics therefore eliminates the need for the assumption that certain ACTIONS are able to be executed asynchronously. It further enables more natural implementations, as the ACTIONS do not have to depend on the results of previous ACTIONS. The token-passing allows for non-tail variants that we implement for the factorial function.

We evaluate the programs provided in `bench/monad/`. Of course, the requirement for sequentializing arithmetic does not always exist, and should be seen as a deliberate construction of monadic binds. By using more asynchronous ACTIONS, the results will more and more converge to the results in section 5.7.

In table 6.1, we count the token-passing interactions with the MONADIC BIND and MONADIC UNIT agents as "Token". The execution of FORKS and ACTIONS as well as their intermediate interactions are counted as "Effectful".

				Rules					Previous
	Inter.	Iter.	Ratio	Dup.	Ann.	Era.	Tok.	Eff.	Inter.
`fac10_tail`	371	228	1.627	30	102	49	77	113	284
`fac10`	349	225	1.551	20	92	48	77	112	-
`fac20_tail`	731	458	1.596	60	202	89	157	223	554
`fac20`	689	455	1.515	40	182	88	157	222	-
`fib10_tail`	400	242	1.653	33	123	55	85	104	304
`fib20_tail`	750	462	1.623	63	233	95	165	194	564
`collatz24`	673	366	1.839	44	187	123	129	190	541
`collatz25`	1445	820	1.762	96	408	238	295	408	1152

Table 6.1 The interaction counts and iterations required to reduce the programs in `bench/monad/` to their normal form in Σ_{pmon}. The previous results from table 5.1 are provided in the right column for comparison.

Comparing table 6.1 with section 5.7 provides initial insights into the overhead that sequentialization via the monadic token-passing semantics adds to reduction. Specifically, any asynchronous ACTION is now executed synchronously, while linearly dependent ACTIONS are now sequentialized via token-passing. Still, the average available parallelism per iteration is larger than one.

We also see how the non-tail implementation of the factorial function measures roughly the same as its tail-recursive variant. The practical difference in tail and non-tail lies in whether the TOKEN applies the `mul` function after each time it returns from the recursive call, or if it evaluates the `mul` as an argument to the recursive call. In the first case, the `mul` agents are therefore kept until the TOKEN returned from the final recursive call.

In figure 6.3, we compare the available parallelism without token-passing semantics to the sequentialized version in the monadic style on the basis of `fac10_tail`.

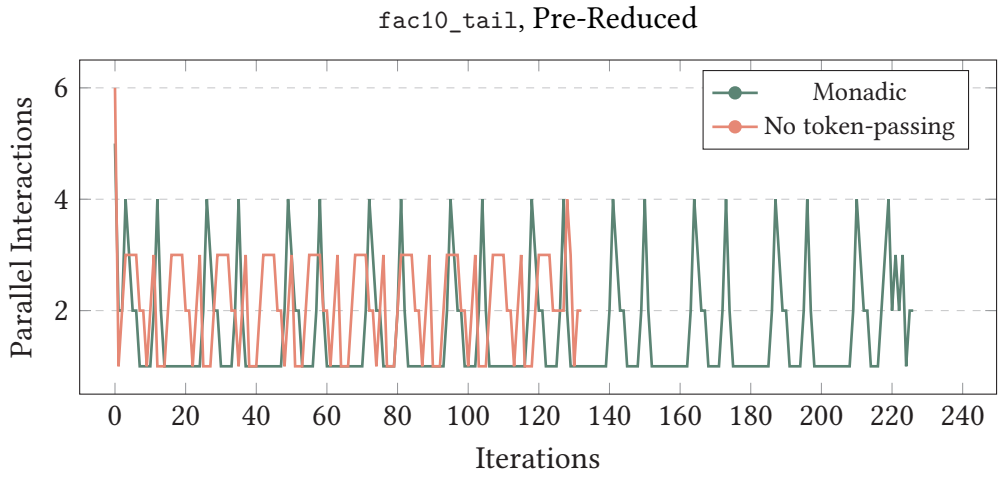


Figure 6.3 Available parallelism per iteration during the reduction of `fac10_tail` in the monadic style vs. no token-passing.

As expected, synchronous execution requires more iterations and interactions than asynchronous execution, which is visible in figure 6.3. As the monadic token-passing semantics does not influence β-reduction or memory management, the available parallelism per iteration remains, albeit with an additional token-passing overhead.

6.8 Discussion

The monadic style creates an elegant separation of effectful and pure terms. We have proven the monadic token-passing semantics to be correct, thus always behaving the same irrelevant of the order in which it is reduced. As every acitve pair of agents still matches at most one well-defined interaction rule, the laws of interaction nets are preserved, thus leading to the following proposition:

Proposition 6.2 (Strong Confluence of $\rightsquigarrow$, Continuation of corollary 5.5)**.** *Assuming all asynchronously executed* ACTORS *to be orthogonal, the reduction relation* $\rightsquigarrow$ *of* $\Sigma_{\text{pmon}} \setminus \{\vee\}$ *satisfies the diamond property modulo time.*

As for corollary 3.7, *modulo time* only refers to the precise time of execution, as there can still exist multiple reduction paths where ACTORS are executed at different points in time *in practice*. Importantly, this difference in execution does not affect the sequentiality requirement, as it is guaranteed by the token-passing semantics.

Our TOKEN-based embedding of the monadic style is based around the idea that the TOKEN itself represents the REAL WORLD implicitly, thus relating to ex-

isting methods of sequentializing side effects such as Haskell's IO Monad. As per the monad laws, the TOKEN can never enter ABSTRACTIONS or APPLICATIONS, therefore never leaving the monadic construct nor interfering with other reductions. In preliminary results we have found this to translate to practical programs: Interactions such as β-reduction remain parallelizable, with the average interactions per iteration staying above 1.5 for all of our tests.

The monadic style forms an elegant extension upon the system of no token-passing semantics, while retaining its properties such as availability for parallelism. This comes with the cost of syntactic constructs surrounding any effectful term supposed to be executed in a sequence.

Chapter 7

Direct Style

We have shown that the monadic approach to sequentializing effects adds syntactic complexity in such a way that any code path able to potentially cause side effects will have to be wrapped in a monad via the `do`-syntax.

In order to preserve the sequentiality previously encoded in the monad, the direct style requires the execution order to be encoded in the token-passing semantics, i. e. in the translation to custom agents of interaction nets.

In the first section we define the direct style and show existing implementations. We then describe a syntactic extension to $\mathcal{L}_{\text{sim}}$ allowing such direct style to be used. In section 7.3, we present an approach similar to Clean and Vine as introduced in chapter 2, where the Real World is passed as an explicit object. In the rest of this chapter we then interpret the world-state passing from the perspective of a token-passing semantics. Finally, we come to the conclusion that the way nets are required to be extended in order to support such direct-style token-passing semantics is indeed almost equivalent to monadic agents.

7.1 Introduction

Imperative programming languages traditionally allow writing effectful functions in a direct and sequential way. Take as example the Python programming language in listing 10. The effectful terms are executed from top to bottom and from left to right[1].

[1] `https://docs.python.org/3/reference/expressions.html#evaluation-order`, 07/13/2025

Listing 10 A direct-style program in Python calculating the factorial of a given positive integer while printing the multiplier in each iteration.

```
print("Enter your number!")
x = int(input())

def fac(n):
  print(n)
  return 1 if n == 0 else n * fac(n - 1)

print(fac(x))
```

The direct style is more intuitive when a program uses many side effects and a strict syntactic separation between the pure and impure code is not desired. This makes it appropriate for imperative programming languages such as C or Python.

On the other hand, functional programming languages tend to prohibit such blending of pure and impure code. We have seen in chapter 6 how a monadic style can be used to accomplish this separation. Yet, supporting side effects in functional programming languages is not unheard of: Among others, OCaml, F#, and Scala all support direct-style execution of side effects while still providing the general purity of functional programming.

The key to supporting sequential execution in functional paradigms is to restrict the evaluation to a certain reduction strategy which reflects the desired execution order. For example, OCaml uses a strict Call-by-Value reduction order, typically right-to-left, which causes effectful terms to be executed in this order as well [35, p. 167].

In the inherently unordered model of interaction nets, forcing a reduction order is not appropriate: Although the ACTORS would then get executed in a specific order, it would also cause memory management, partial application, and β-reduction to be sequentialized needlessly. Instead, the execution order is specified in the token-passing semantics such that it interferes with reduction as little as possible.

7.2 Language Extension

Definition 7.1 (Terms of $\mathcal{L}_{\text{dir}}$)**.** We extend the language $\mathcal{L}_{\text{sim}}$ from definition 4.1. Taking inspiration from OCaml [35], $\mathcal{L}_{\text{dir}}$ allows writing standalone expressions.

The terms of the direct language $\mathcal{L}_{\text{dir}}$ are:

$$\begin{array}{rll} t & ::= \cdots & \text{Terms of } \mathcal{L}_{\text{sim}} \\ & \mid t; t & \text{Sequence} \end{array}$$

A sequence does in fact represent only a syntactic sugar on `let _ = t; t`, where "_" ignores the result. The symbol "_" is special in that it can never be bound by a symbol.

The syntax of $\mathcal{L}_{\text{dir}}$ allows writing programs in an imperative style, where the program is evaluated from top to bottom and right to left. Since standalone terms can contain actions, terms may depend on the reduction of previous terms, even though they are not bound to them by variable. In a fully implicit approach to sequentialization of actions, programs do not require additional syntactic constructs.

We translate listing 10 to $\mathcal{L}_{\text{dir}}$ in listing 11.

Listing 11 A factorial program for positive integers in $\mathcal{L}_{\text{dir}}$. We assume the definitions of token-reflecting `print`, `readInt`, `writeInt`, `isEqual`, `mul`, and `pred`.

```
print "Enter your number!";
let x = readInt ⟨⟩;

let fac n = (
  writeInt n;
  if (isEqual 0 n) then 1
  else mul n (fac (pred n))
);

writeInt (fac x)
```

Note how, finally, there is no need to handle data such as strings or numbers differently than pure terms. Without token-passing semantics, strings have to be loaded synchronously, as in listing 4. In the monadic style, they require embedding in the monadic construct, as in listing 9. Here, impure data can be written anywhere. Yet, the order in which it is received from the Real World is strict.

7.3 World-State Passing

In an initial step towards the desired semantics, we introduce the world-state passing style.

Actions in world-state passing receive the current state of the Real World as an additional argument. An action will produce the new state of the Real

World alongside its pure result whenever it is executed. When threaded through all the actions in a program, such that the object that encodes the Real World is never duplicated or erased, a program will consume an initial state of the world and produce the new state of the world after the program was executed.

Whether or not the object that is being passed along actually encodes data, or exists purely for sequentialization, depends on the specific implementation and requirements. We have discussed this phenomenon for monads in section 6.1, which requires a bind to pass the Real World. The general approach to world-state passing is based on the concept of *environment passing* from the Clean programming language [22]. Further details are described in section 2.4, whereas the concept of world-state passing is also discussed for Ivy and Vine in section 2.2.

The explicit world-passing style diverges from the proposed direct-style semantics in that it requires threading various versions of the Real World linearly through the program.

Definition 7.2 (Terms of the Language $\mathcal{L}_{\oplus}$)**.** We extend $\mathcal{L}_{\text{dir}}$ from definition 7.1 with explicit world-state passing. The terms of the language $\mathcal{L}_{\oplus}$ are:

t	$::= \cdots$	Terms of $\mathcal{L}_{\text{dir}}$
	$\mid \oplus^{(i)}$	Real World
	$\mid (\oplus^{(i)}, t)$	Paired World
	$\mid \textbf{let}\ (\oplus^{(i)}, n)\ n^* = t; t$	Let with World

This threading of the Real World then constructs linear dependencies of terms as in $\mathcal{L}_{\text{sim}}$, only that actions now also depend on the execution of actions even if they do not depend on the produced terms themself.

Listing 12 The factorial program of listing 11 in $\mathcal{L}_{\oplus}$. Every synchronous action receives an additional argument for the Real World as well as produces a potentially modified version of the Real World. For readability, we denote the nested use of $\oplus$ with $\otimes$.

```
let go ⊕ = (
  let (⊕′, _) = print "Enter your number!" ⊕;
  let (⊕″, x) = readInt ⟨⟩ ⊕′;

  let fac n ⊗ = (
    let (⊗′, _) = writeInt n ⊗;
    if (equal? n 0) then (⊗′, 1)
    else (
      let (⊗″, prev) = fac (pred n) ⊗′;
      (⊗″, mul n prev)
    )
  );

  let (⊕‴, res) = (fac x ⊕″);
  print res ⊕‴
);
go
```

Explicit world-state passing may then be translated to interaction nets by giving actors an additional port for passing along the updated version of the Real World. Further, by instead using *reference* agents to track the updates of the Real World as in Vine, the frontend syntax will not require tracking the versions of the Real World produced by actions.

7.4 Token Passing

The application of the Real World in section 7.3 reminds of the token applied to actions. Indeed, the produced modified version of the Real World can be seen as the reflection of a cotoken.

We explore this implicit approach to world-state passing in a token-passing semantics for Λ_{eff} and Σ_{peff}.

7.4.1 Semantics

The token-passing semantics of $\mathcal{L}_{\text{dir}}$ can be described directly via Λ_{eff}, since it does not contain additional syntactic constructs.

At this stage of translation from $\mathcal{L}_{\text{sim}}$ to Λ_{eff}, tokens may appear anywhere in a term. By the force operator "!" in $\mathcal{L}_{\text{sim}}$, the token may either be in asyn-

chronous actions, as described in section 3.4, is *stuck* by invalid use of the TOKEN, or is used to trigger the token-passing and execute the ACTIONS in a term.

Specifically, we take inspiration from the reduction order of common programming languages such as OCaml [35, p. 167], and execute ACTIONS in an order that resembles the Call-by-Value reduction strategy.

In order to argue about the execution order of ACTIONS, we introduce the concept of the *action potential*. Action potential implies that a term may cause side effects to happen. We use as notation an underline as used by Levy [29] to denote *computation types*. In our case, however, arbitrary terms may gain or lose action potential based on interactions with TOKENS.

Definition 7.3 (Action Potential)**.** When a term M has action potential, we write $\underline{M}$. By default, every term has action potential. Terms only lose their action potential by respective rewrite rules. We omit marking the children's or parent's action potential explicitly when it does not influence the reduction behavior.

The β-reduction rule as well as the ξ-reduction rules defined in section 3.2 only apply to terms *without* any action potential. This prevents reduction rules interfering with the rules of the token-passing semantics. Otherwise, reductions could for example duplicate or erase the TOKEN.

The *execution strategy* of Λ_{eff} provides the reductions which yield a term without action potential. The TOKEN reflects from terms unable to cause immediate effects and removes their action potential while executing the ACTIONS in a specific order. Since we mirror the execution of the Call-by-Value evaluation strategy, we translate its behavior to corresponding token-passing rules. We take as inspiration the token-passing semantics by Sinot [32]. Instead of their directed reduction relation $\Downarrow$ and left-to-right reduction order, we use the language's explicit token and a right-to-left execution order.

Definition 7.4 (Token-Passing Semantics of Λ_{eff})**.** The direct style token-passing of Λ_{eff} is described by the following rewrite rules:

$$\underline{M}\ \underline{N}\ \triangleleft \overset{\triangle}{\rightsquigarrow} \underline{M}\ (\underline{N}\ \triangleleft)$$
$$\underline{M}\ (N\ \triangleright) \overset{\triangle}{\rightsquigarrow} \underline{M}\ \triangleleft\ N$$
$$M\ \triangleright\ N \overset{\triangle}{\rightsquigarrow} M\ N\ \triangleleft$$

$$\underline{(\lambda x.M)}\ \triangleleft \overset{\triangle}{\rightsquigarrow} (\lambda x.\underline{M})\ \triangleright$$
$$\underline{\xi^i_{\mathcal{C}}}\ \triangleleft \overset{\triangle}{\rightsquigarrow} \xi^i_{\mathcal{C}}\ \triangleright, \quad i > 0$$
$$\underline{\langle \ldots \rangle}\ \triangleleft \overset{\triangle}{\rightsquigarrow} \langle \ldots \rangle\ \triangleright$$
$$\underline{x}\ \triangleleft \overset{\triangle}{\rightsquigarrow} x\ \triangleright$$

As a more high-level interpretation it can be observed that when an APPLICATION is to be executed (indicated by a TOKEN), the argument N is executed next. After this execution is done (indicated by a COTOKEN), the term M itself

is executed. Finally, when this execution is done as well, the APPLICATION may finally be reduced as normal, with the resulting term being executed next.

By a term being executed, any terms within this term are executed recursively in this order as well. The only rule preventing endless recursion are the reflection rules which make the execution end without recursing deeper. Specifically, the reflection from ABSTRACTIONS implies that we never execute ACTIONS within an ABSTRACTION, but only when it was applied to an already executed term. This behavior is to be expected in traditional, *weakly evaluated* programming languages.

Since the translation of $\mathcal{L}_{\text{sim}}$ defined in section 4.2 translates LET EXPRESSIONS to APPLICATIONS, multiple LET EXPRESSIONS will be executed from top to bottom, as the APPLICATIONS are executed from right to left.

The recursive entering and returning of the TOKEN and COTOKEN can be seen in the following example.

Example 7.5. We continue example 3.4. The obvious overhead of our token-passing semantics is discussed in section 7.4.3.

$$
\begin{aligned}
&\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ (\underline{\texttt{readFile}^{1}_{\emptyset}\ \langle\text{“example.txt”}\rangle})\ \triangleleft \\
\overset{\triangle}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ (\underline{\texttt{readFile}^{1}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ \triangleleft) \\
\overset{\triangle}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ (\underline{\texttt{readFile}^{1}_{\emptyset}}\ (\underline{\langle\text{“example.txt”}\rangle}\ \triangleleft)) \\
\overset{\triangle}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ (\underline{\texttt{readFile}^{1}_{\emptyset}}\ (\langle\text{“example.txt”}\rangle\ \triangleright)) \\
\overset{\triangle}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ (\underline{\texttt{readFile}^{1}_{\emptyset}}\ \triangleleft\ \langle\text{“example.txt”}\rangle) \\
\overset{\xi}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ (\texttt{readFile}^{1}_{\emptyset}\ \triangleright\ \langle\text{“example.txt”}\rangle) \\
\overset{\triangle}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ (\texttt{readFile}^{1}_{\emptyset}\ \langle\text{“example.txt”}\rangle\ \triangleleft) \\
\overset{\xi}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ (\texttt{readFile}^{0}_{[\langle\text{“example.txt”}\rangle]}\ \triangleleft) \\
\overset{\xi}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ (\langle\text{“hello, world”}\rangle\ \triangleright) \\
\overset{\triangle}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ \triangleleft\ \langle\text{“hello, world”}\rangle \\
\overset{\triangle}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ (\underline{\langle\text{“example.txt”}\rangle}\ \triangleleft)\ \langle\text{“hello, world”}\rangle \\
\overset{\triangle}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ (\langle\text{“example.txt”}\rangle\ \triangleright)\ \langle\text{“hello, world”}\rangle \\
\overset{\triangle}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \triangleleft\ \langle\text{“example.txt”}\rangle\ \langle\text{“hello, world”}\rangle \\
\overset{\xi}{\rightsquigarrow}\ &\texttt{writeFile}^{2}_{\emptyset}\ \triangleright\ \langle\text{“example.txt”}\rangle\ \langle\text{“hello, world”}\rangle \\
\overset{\triangle}{\rightsquigarrow}\ &\texttt{writeFile}^{2}_{\emptyset}\ \langle\text{“example.txt”}\rangle\ \triangleleft\ \langle\text{“hello, world”}\rangle \\
\overset{\xi}{\rightsquigarrow}\ &\underline{\texttt{writeFile}^{1}_{[\langle\text{“example.txt”}\rangle]}}\ \triangleleft\ \langle\text{“hello, world”}\rangle \\
\overset{\xi}{\rightsquigarrow}\ &\texttt{writeFile}^{1}_{[\langle\text{“example.txt”}\rangle]}\ \triangleright\ \langle\text{“hello, world”}\rangle
\end{aligned}
$$

$\overset{\triangle}{\rightsquigarrow}$ $\texttt{writeFile}^{1}_{[\langle\text{“example.txt”}\rangle]}$ $\langle\text{“hello, world”}\rangle$ $\triangleleft$

$\overset{\xi}{\rightsquigarrow}$ $\underline{\texttt{writeFile}}^{0}_{[\langle\text{“example.txt”}\rangle,\langle\text{“hello, world”}\rangle]}$ $\triangleleft$

$\overset{\xi}{\rightsquigarrow}$ $\langle\rangle$ $\triangleright$

Note how the reflecting ACTIONS produce terms without action potential. If the resulting terms contained immediate ACTIONS, the implementation would be responsible for bouncing the TOKEN.

By only allowing β-reductions on executed terms, only as many β-reductions can happen in parallel as there are TOKENS in a term. We improve upon this limitation in section 7.4.4.

We derive the following based on the results of Sinot [32] and the described semantics of Λ_{eff}:

Proposition 7.1 (Return of the Token)**.** *Assuming that all ACTIONS reflect a TOKEN after execution, a TOKEN-free term $\underline{M} \in \Lambda_{\text{eff}} \setminus \{\triangleleft, \triangleright\}$ applied to a TOKEN always either diverges, or reduces to a term applied to a COTOKEN:*

$$M \downarrow M' \iff \underline{M} \triangleleft \overset{\beta\xi\psi\triangle}{\rightsquigarrow} M' \triangleright$$

7.4.2 Redirector Agents

The presented token-passing semantics via decreasing action potential adds a TOKEN traversal so general it can not be embedded into Σ_{peff} as defined. This is because in order for an agent to *redirect* a TOKEN to one of its ports, it has to interact with it first. Such interaction must happen at an auxiliary port for APPLICATORS, as this is the port where an APPLICATION is connected to its surrounding term.

Once a COTOKEN returns from the port where a TOKEN has been redirected to, it must in turn interact with this returning agent. Therefore, the APPLICATOR has to be *rotatable* such that it may interact with any of its ports.

This rotation has been described by Sinot [32] in a similarly token-based approach, encoding certain sequential *reduction strategies* into the net. Their approach linearizes the entire reduction, while we aim to embed the sequential *execution* without blocking pure reductions.

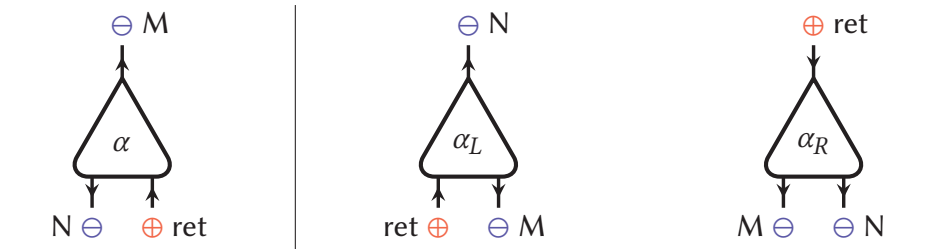
Similar rotations have been discussed in the work on *hard combinators* [46], which describe a variant of interaction nets invariant in their geometry. There, rotating combinators are described as *clocks* where the principal port is iteratively moved around the agent.

We call the rotating APPLICATOR the *redirector*. Where the token-passing execution strategy in Λ_{eff} is encoded using APPLICATIONS and action potentials as described in definition 7.4, it is now encoded in interactions with REDIRECTORS.

Definition 7.6 (Agents of Σ_{pdir})**.** The system of Σ_{pdir} extends Σ_{peff} from definition 5.1 by polarizing the APPLICATOR as two additional *redirector agents*:

$$\Sigma_{\text{pdir}} = \Sigma_{\text{peff}} \cup \{\alpha_L, \alpha_R\},$$

where $\alpha_L \equiv \zeta_\circ(\text{ret}_\oplus, \text{M}_\ominus)^\ominus$ and $\alpha_R \equiv \zeta_\circ(\text{M}_\ominus, \text{N}_\ominus)^\oplus$.



APPLICATOR (TOP) | REDIRECTOR (LEFT) | REDIRECTOR (RIGHT)

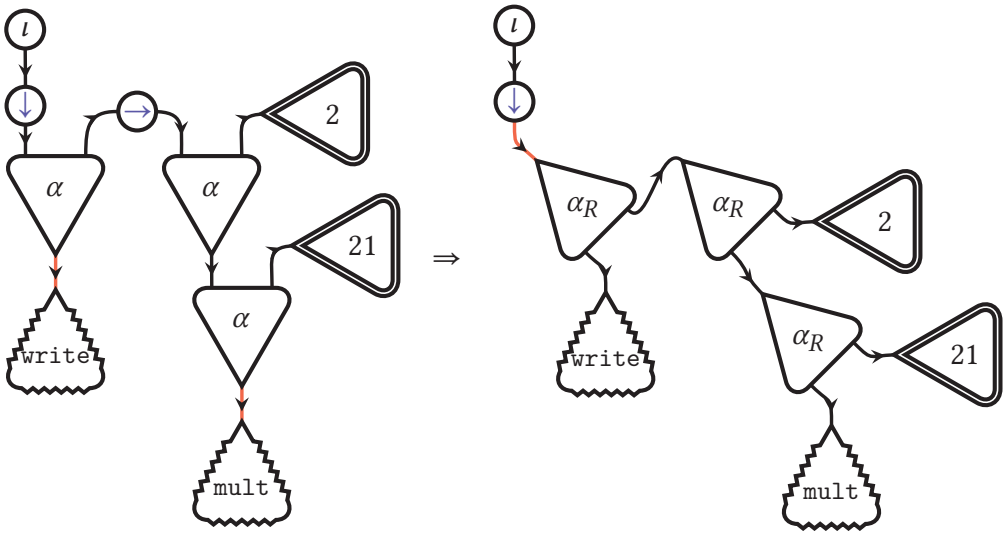
Figure 7.1 The polarized redirector agents of Σ_{pdir}.

In the translation from Λ_{eff} to Σ_{peff}, the TOKENS will be applied to the right auxiliary port of the top APPLICATOR. In order to trigger the token-passing semantics, all APPLICATORS will have to be rotated to right REDIRECTORS, such that they interact with the TOKEN.

Definition 7.7 (Translation $\Lambda_{\text{eff}} \to \Sigma_{\text{pdir}}$)**.** We extend the translation from definition 5.2 by overwriting the translation of APPLICATIONS provided in definition 1.12, such that it incorporates the rotation required for the direct-style token-passing semantics.

$$[\![M\ N]\!]_{\text{p}} = \{\text{p} == \alpha_R(\text{m}, \text{n})\} \cup [\![M]\!]_{\text{m}} \cup [\![N]\!]_{\text{n}}$$

Example 7.8. As in example 5.3, we translate $M = \texttt{write}^1_\emptyset\ (\texttt{mult}^2_\emptyset\ \langle 21\rangle\ \langle 2\rangle)$ to Σ_{pdir}. This time, no TOKENS aside from the initiating TOKEN are required.



The token-passing semantics for Λ_{eff} can then be translated to Σ_{pdir} so that the redirection of the TOKEN agent happens in accordance with the TOKEN term entering and leaving APPLICATIONS. In the following extension of the rules of Σ_{peff} from definition 5.4, the redirection rules mirror the exact behavior of the described direct token-passing semantics:

$$
\begin{aligned}
\alpha_R[x, \triangleleft(\alpha_L(y, x))] &\bowtie \triangleleft[y] \\
\alpha_L[x, \triangleleft(\alpha(y, x))] &\bowtie \triangleright[y] \\
\alpha[x, y] &\bowtie \triangleright[\alpha(x, \triangleleft(y))] \\
\lambda[x, y] &\bowtie \triangleleft[\triangleright(\lambda(x, y))] \\
\xi^i_{\mathcal{C}} &\bowtie \triangleleft[\triangleright(\xi^i_{\mathcal{C}})], \quad i > 0 \\
\pi &\bowtie \triangleleft[\triangleright(\pi)]
\end{aligned}
\qquad\qquad
\begin{aligned}
\underline{M}\ \underline{N} \triangleleft &\overset{\triangle}{\rightsquigarrow} \underline{M}\ (\underline{N} \triangleleft) \\
\underline{M}\ (N \triangleright) &\overset{\triangle}{\rightsquigarrow} \underline{M} \triangleleft N \\
M \triangleright N &\overset{\triangle}{\rightsquigarrow} M\ N \triangleleft \\
\underline{(\lambda x.M)} \triangleleft &\overset{\triangle}{\rightsquigarrow} (\lambda x.\underline{M}) \triangleright \\
\underline{\xi^i_{\mathcal{C}}} \triangleleft &\overset{\triangle}{\rightsquigarrow} \xi^i_{\mathcal{C}} \triangleright, \quad i > 0 \\
\underline{\langle \ldots \rangle} \triangleleft &\overset{\triangle}{\rightsquigarrow} \langle \ldots \rangle \triangleright \\
\underline{x} \triangleleft &\overset{\triangle}{\rightsquigarrow} x \triangleright
\end{aligned}
$$

In figure 7.2 the interactions are annotated with the respective rewrite rules $\overset{\triangle}{\rightsquigarrow}$ of Λ_{eff}.

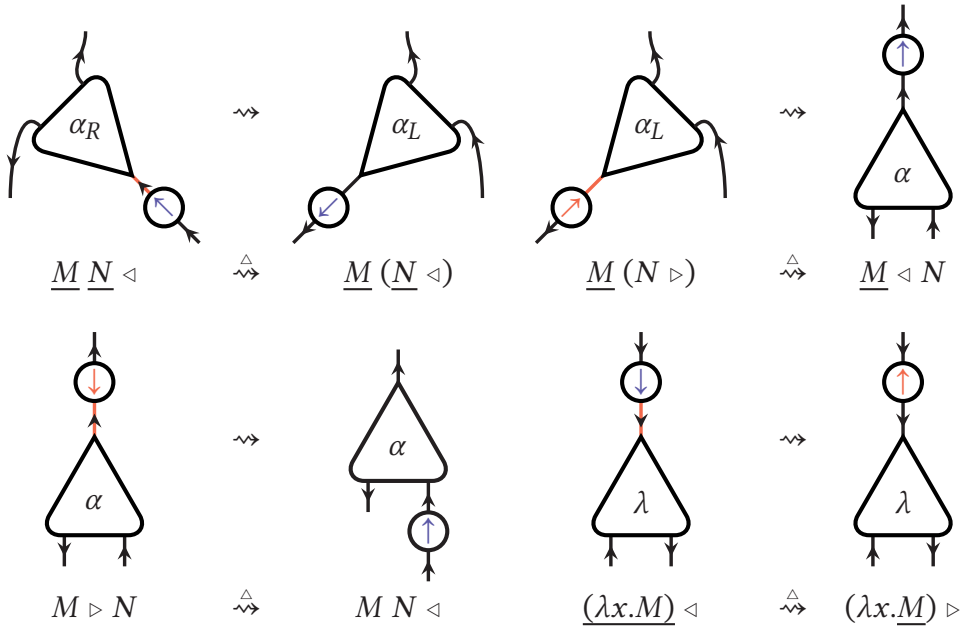


Figure 7.2 Interactions of the redirector agents, annotated with the respective token-passing rules of definition 7.4. Further reflections other than ABSTRACTORS are omitted.

The mentioned restriction of redexes only β-reducing when the APPLICATION has a TOKEN emerges naturally from the use of REDIRECTORS: β-reduction requires interaction with the *top* APPLICATOR, which always sends a TOKEN after rotation from the *left* REDIRECTOR. An interaction with an ABSTRACTOR can then only occur when it reflected a COTOKEN, as seen in figure 7.3.

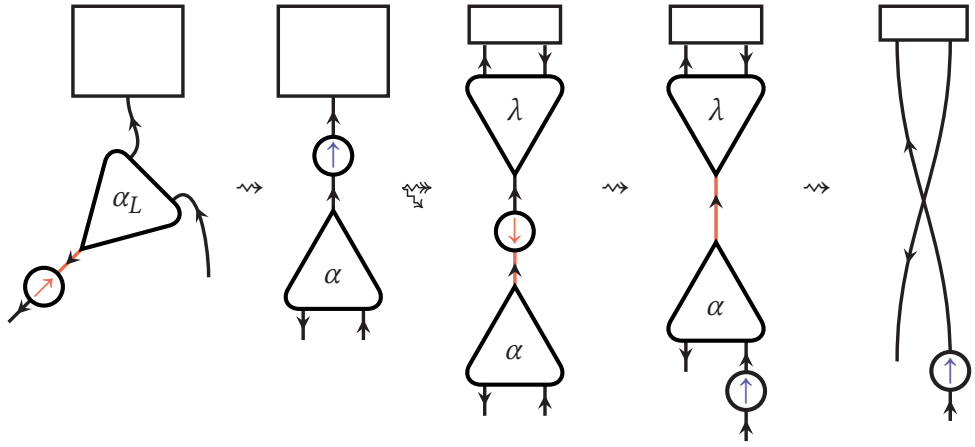


Figure 7.3 β-reduction under token-passing where a $M \twoheadrightarrow \lambda x.M'$. Annotated with the respective token-passing rules of definition 7.4.

7.4.3 Token-Passing Overhead

Without further information, the TOKEN must interact with every term once in order to determine their action potential and potentially execute the ACTIONS within. ACTIONS without enough arguments cause *reflections*, which require additional interactions with the TOKEN equivalent to its arity. Terms produced by the execution of ACTIONS also have to be interacted with at least once.

In general, this overhead can be reduced by acquiring more information about terms. In our simple model we assume that every term has action potential; in reality, this assumption can be improved by *inferring* a term's action potential based on the action potential of its sub-terms. This problem is in fact closely related to the work on *effect systems* [34] which provide a type-based understanding of effects in programs. Aside from types, the syntax of a user-facing language could be extended with syntactic constructs marking certain terms as *pure* or *effectful* in order to determine the action potential of terms in Λ_{eff}.

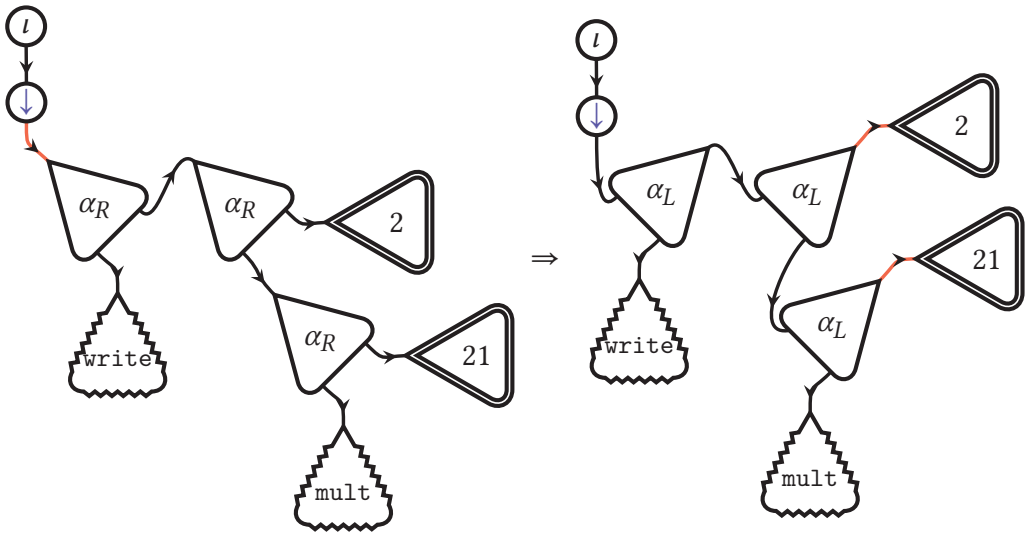
Effectively, such information in Λ_{eff} translates to Σ_{pdir} by rotating certain REDIRECTORS before the TOKEN has reached them. We provide a minimal proof of concept in the following section, which essentially infers the action potential of terms *at runtime*.

7.4.4 Inference of Action Potential

In the inference procedure, every REDIRECTOR is by default rotated to the *left* α_L instead of the right α_R. TOKENS thus cannot pass, as the LEFT REDIRECTOR has its

negative principal port connected to the argument of the APPLICATION.

Example 7.9. As in example 7.8, we translate $M = \texttt{write}^1_\emptyset\ (\texttt{mult}^2_\emptyset\ \langle 21\rangle\ \langle 2\rangle)$ to Σ_{pdir}. This time, the REDIRECTORS are rotated to the left.



It is clear from example 7.9 that additional interaction rules with non-effectful agents are required. We define these generically for agents depending on whether they have *immediate action potential* or not.

Definition 7.10 (Immediate Action Potential)**.** The RIGHT REDIRECTOR, FORKS, and ACTORS with arity 0 have immediate action potential:

$$\Sigma_{\text{im}} = \{\alpha_R, \wedge, \vee, \xi^0_{\mathcal{C}}\}$$

Any other agent aside from agents responsible for memory management or execution do not have immediate action potential:

$$\Sigma_{!\text{im}} = \Sigma_{\text{peff}} \setminus \Sigma_{\text{im}} \setminus \{\delta, \sigma, \varepsilon, \nu, \triangleleft, \triangleright\}$$

The RIGHT REDIRECTOR agent α_R is used to (1) track the inferred action potential, and (2) pave the path for the token-passing semantics, as a TOKEN can only pass through α_R. This works only because α_R can by construction not occur in a net translated by Λ_{eff}.

The inference aims to spread immediate action potential such that its surrounding REDIRECTORS are also considered to have action potential. If an agent does not have immediate action potential, its surrounding REDIRECTORS should turn into APPLICATORS such that they can interact immediately without requiring a TOKEN.

From the perspective of Λ_{eff}, we can activate the inference rules by *assuming* that a COTOKEN has returned from either a term with or without action potential. Then, the resulting term should reflect the next or previous state of the token-passing rule such that all action potential will eventually be eliminated. We can derive the following inference rules by introducing the rewrite rule $\overset{\triangledown}{\rightsquigarrow}$:

- $\underline{M}\ (N\ \triangleright) \overset{\triangledown}{\rightsquigarrow} \underline{M}\ \triangleleft N$, as in definition 7.4.
- $\underline{M}\ (\underline{N}\ \triangleright) \overset{\triangledown}{\rightsquigarrow} \underline{M}\ \underline{N}\ \triangleleft$, in order to trigger $\underline{M}\ (\underline{N}\ \triangleleft)$.
- $M\ \triangleright\ \underline{N} \overset{\triangledown}{\rightsquigarrow} M\ \underline{N}\ \triangleleft$, as in definition 7.4.
- $\underline{M}\ \triangleright\ \underline{N} \overset{\triangledown}{\rightsquigarrow} \underline{M}\ \underline{N}\ \triangleleft$, in order to trigger $\underline{M}\ (\underline{N}\ \triangleleft)$ and then $(\underline{M}\ \triangleleft)\ N$.

Here we write $\underline{M}$ instead of $\underline{M}$ or M, as inference runs during reduction and the terms might already be inferred as not having action potential.

In interaction nets, these TOKENS will not be passed explicitly. Instead, the inferred rotation of REDIRECTORS depends on whether the connected agents are in Σ_{im} or $\Sigma_{!\text{im}}$, thus resembling an implicitly returned COTOKEN.

We derive the following interaction rules for $e \in \Sigma_{\text{im}}$ and $u \in \Sigma_{!\text{im}}$:

$$\alpha_L[\mathsf{x}, \alpha(u(\mathsf{y}_1, \dots, \mathsf{y}_m), \mathsf{x})] \bowtie e[\mathsf{y}_1, \dots, \mathsf{y}_n] \qquad \underline{M}\ (N\ \triangleright) \overset{\triangledown}{\rightsquigarrow} \underline{M}\ \triangleleft N$$

$$\alpha_L[\alpha_R(\mathsf{x}, e(\mathsf{y}_1, \dots, \mathsf{y}_n))] \bowtie u[\mathsf{y}_1, \dots, \mathsf{y}_m] \qquad \underline{M}\ (\underline{N}\ \triangleright) \overset{\triangledown}{\rightsquigarrow} \underline{M}\ \underline{N}\ \triangleleft$$

$$M\ \triangleright\ \underline{N} \overset{\triangledown}{\rightsquigarrow} M\ \underline{N}\ \triangleleft$$

$$\alpha[\mathsf{x}, \alpha_R(e(\mathsf{y}_1, \dots, \mathsf{y}_n), \mathsf{x})] \bowtie e[\mathsf{y}_1, \dots, \mathsf{y}_n] \qquad \underline{M}\ \triangleright\ \underline{N} \overset{\triangledown}{\rightsquigarrow} \underline{M}\ \underline{N}\ \triangleleft$$

In figure 7.4 the interactions are annotated with the implicit (CO-)TOKENS and the respective inference rules $\overset{\triangledown}{\rightsquigarrow}$ of Λ_{eff}. The resulting TOKENS indicate where the actual token-passing semantics will traverse.

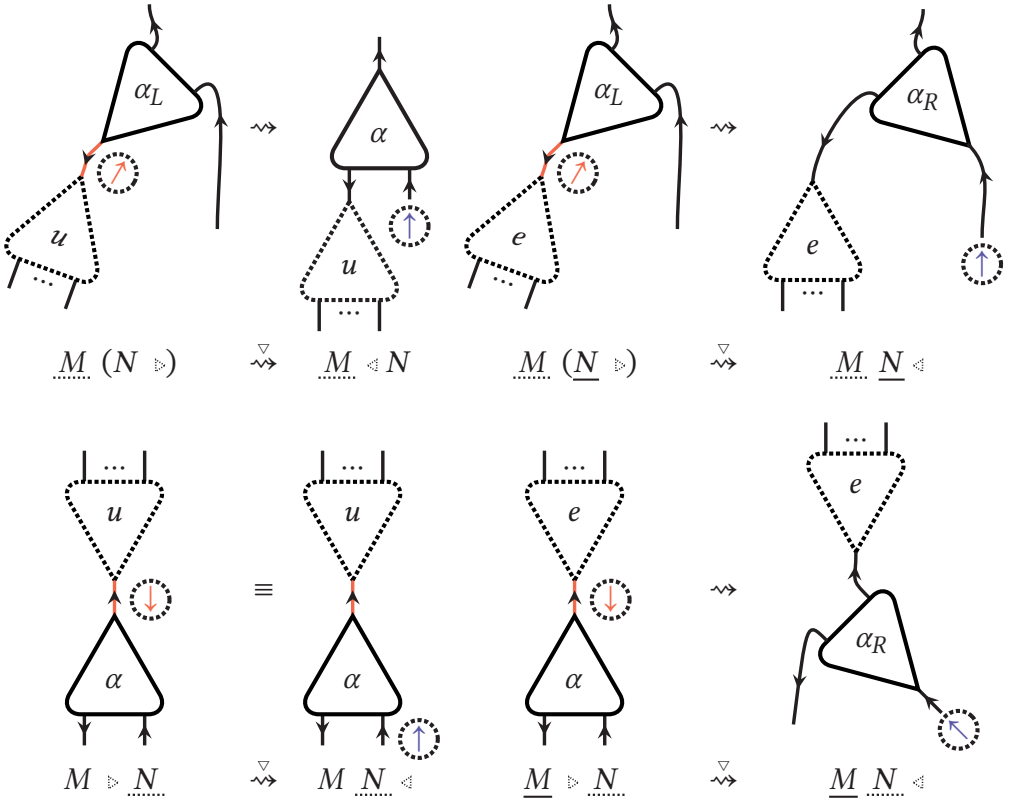
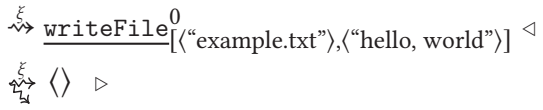


Figure 7.4 Runtime inference of Σ_{pdir}, annotated with the inference rules of Λ_{eff}.

Using the presented interactions, reductions can happen in parallel to execution and token-passing. Any agent in $\Sigma_{!\text{im}}$ spreads its purity during reduction such that partial application, memory management, and β-reduction may happen before a TOKEN has reached them.

Example 7.11. Coming back to example 7.5, the reduction may now be described as:

$$
\begin{aligned}
&\underline{\texttt{writeFile}^{2}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle}\ (\underline{\texttt{readFile}^{1}_{\emptyset}}\ \underline{\langle\text{“example.txt”}\rangle})\ \triangleleft \\
\overset{\triangledown}{\rightsquigarrow}\ &\texttt{writeFile}^{2}_{\emptyset}\ \langle\text{“example.txt”}\rangle\ (\texttt{readFile}^{1}_{\emptyset}\ \langle\text{“example.txt”}\rangle)\ \triangleleft \\
\overset{\xi}{\rightsquigarrow}\ &\underline{\texttt{writeFile}}^{1}_{[\langle\text{“example.txt”}\rangle]}\ \underline{\texttt{readFile}}^{0}_{[\langle\text{“example.txt”}\rangle]}\ \triangleleft \\
\overset{\triangledown}{\rightsquigarrow}\ &\texttt{writeFile}^{1}_{[\langle\text{“example.txt”}\rangle]}\ \underline{\texttt{readFile}}^{0}_{[\langle\text{“example.txt”}\rangle]}\ \triangleleft \\
\overset{\triangledown}{\rightsquigarrow}\ &\texttt{writeFile}^{1}_{[\langle\text{“example.txt”}\rangle]}\ (\underline{\texttt{readFile}}^{0}_{[\langle\text{“example.txt”}\rangle]}\ \triangleleft) \\
\overset{\xi}{\rightsquigarrow}\ &\texttt{writeFile}^{1}_{[\langle\text{“example.txt”}\rangle]}\ (\langle\text{“hello, world”}\rangle\ \triangleright) \\
\overset{\triangledown}{\rightsquigarrow}\ &\texttt{writeFile}^{1}_{[\langle\text{“example.txt”}\rangle]}\ \langle\text{“hello, world”}\rangle\ \triangleleft
\end{aligned}
$$



7.4.5 Recursion

Recursion in the direct-style semantics works similar to the monadic style (section 6.6). However, here the explicit “loading” of the recursive call is not required. Instead, the TOKEN executes the function an argument is applied to first before applying the argument, thus performing this loading implicitly. Consequently, any recursive call requires an argument (say, a UNIT) in order to execute the call twice.

Another key difference is the way closures are to be constructed. As described in section 4.3, the closure consists of APPLICATIONS of the terms a definition closes over. As such, any argument—even ones not bound in the body—will be executed by the presented token-passing semantics. This includes top-level definitions such as `let x = readInt <>`, which would get executed in every iteration of the recursion.

A simple solution for closing over such terms is the additional abstraction (*thunking*) of any definition without arguments. This way, their effects can be repeated by applying arbitrary values without them being caused by the Call-by-Value semantics. More reasonably however, definitions that are expected to run only *once* should be handled separately in more sophisticated transformations, which are beyond the scope of this thesis.

Another approach would be to use an explicit RECURSOR agent which would eliminate the need for closures to be constructed. The token-based agent responsible for recursion presented in the work of Almeida; Pinto; Vilaça [47] would be an appropriate choice.

7.4.6 Redirectors Generalize Monads

The monadic style has strong similarities to the presented direct style. Multiple effectful terms can be chained together using a SEQUENCE, which in a way corresponds to the `do`-syntax. However, the direct style also supports *inlining* effectful terms as they are executed in a Call-by-Value fashion from right to left. This difference is shown by example in figure 7.5.

```
let repl = do (
  s ← "Enter two numbers!";
  _ ← print s;
  x ← readInt ⟨⟩;
  y ← readInt ⟨⟩;
  s ← add x y;
  _ ← writeInt s;
  r ← repl;
  r
);
repl
```

```
let repl _ = (
  print "Enter two numbers!";
  x = readInt ⟨⟩;
  y = readInt ⟨⟩;
  writeInt (add x y);
  repl <>
);
repl <>
```

Figure 7.5 REPL in monadic style from example 6.9 vs. direct style. The asynchronous ACTIONS are made synchronous to match the direct style.

In the monadic style, the execution of ACTIONS can be deferred by wrapping it inside a UNIT such that it has to be executed *twice* in order to be effectful. ABSTRACTIONS can be used similarly in the direct style: By arguing with a token-passing semantics based on action potential, abstracted terms will also require two executions (and an arbitrary application) in order to be effectful.

The agents of Σ_{pdir} and Σ_{pmon} have further similarities: The MONADIC BIND has an executive state which sends a TOKEN and interacts with the returning COTOKEN by turning into an APPLICATOR. In comparison, the LEFT REDIRECTOR also sends a TOKEN, and as well rotates further into an APPLICATOR once the COTOKEN returns.

Both monadic and direct styles have a third agent which is only used to interact with the initiating TOKEN of the token-passing semantics: The non-executive MONADIC BIND in the monadic style, and the RIGHT REDIRECTOR in the direct style.

Indeed, the direct style emerges merely from a substitution of the MONADIC BIND into every RIGHT REDIRECTOR (or APPLICATOR), where the MONADIC UNIT is then implicit in reflecting agents. The inference interactions in turn gain back the potential for reduction lost during the substitution of TOP APPLICATORS.

We can see in figure 7.6 how the only interaction not existing in the monadic style is the traversal of the TOKEN into the continuation of the bind—i. e. the left side of an APPLICATION. This interaction does not make sense in the monadic style, as the continuation must always start with an ABSTRACTOR and the TOKEN would not interact with ABSTRACTORS—nor APPLICATORS for that matter.

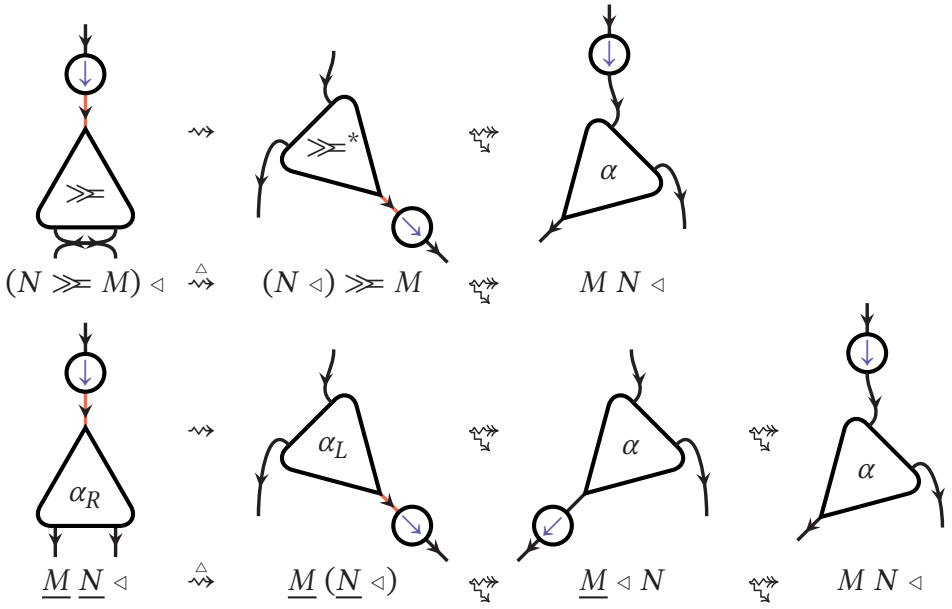


Figure 7.6 Symmetry of monadic agents and redirectors. For visual purposes, the auxiliary ports of the MONADIC BIND are swapped. In both styles, the terms M, N are connected to the bottom left and right auxiliary ports, respectively.

7.5 Evaluation

We continue the evaluations of section 6.7 and section 5.7.

7.5.1 Comparison

We compare the inferred and non-inferred direct style semantics to the monadic style. In table 7.1, we compare the direct style semantics to the results of section 6.7 by evaluating the equivalent programs in the provided `bench/direct/` directory.

Similar to section 6.7, we count token-passing steps such as rotating REDIRECTORS as "Token" interactions. This also includes inference interactions if there are any.

				Rules					Monad
	Inter.	Iter.	Ratio	Dup.	Ann.	Era.	Tok.	Eff.	Inter.
No Inference									
`fac10_tail`	723	644	1.123	30	61	40	458	134	371
`fac20_tail`	1443	1294	1.115	60	121	70	918	274	731
`fib10_tail`	863	775	1.114	33	78	45	559	148	400
`fib20_tail`	1643	1485	1.106	63	148	75	1069	288	750
`collatz24`	1400	1183	1.183	44	109	151	842	254	673
`collatz25`	3102	2661	1.166	96	239	297	1889	581	1445
Inference									
`fac10_tail`	571	364	1.569	30	61	34	306	140	371
`fac20_tail`	1131	724	1.562	60	121	64	606	280	731
`fib10_tail`	693	408	1.699	33	78	39	409	134	400
`fib20_tail`	1313	778	1.688	63	148	69	779	254	750
`collatz24`	1095	684	1.601	44	109	95	597	250	673
`collatz25`	2394	1538	1.557	96	239	189	1319	551	1445

Table 7.1 The interaction counts and iterations required to reduce the programs in `bench/direct/` to their normal form in Σ_{pdir}. The previous results for the monadic style from table 6.1 are provided in the right column for comparison.

As clear from table 7.1, the direct style with inference still requires more interactions than the monadic style, thus extending the hypothesis of section 7.4.6 to practical observations: Inferring the direct style aims to match the parallelism and semantics of the monadic style, but requires an interaction overhead during the runtime inference in order to reach this goal.

Without inference, the direct style has barely any available parallelism per iteration while at the same time requiring around twice as many interactions as the monadic style. This is to be expected, as the right rotations of the REDIRECTORS linearize the entire reduction path. Therefore, the only parallel interactions are execution, partial application, and garbage collection, but never β-reduction.

We visualize this relation of interaction and iteration counts in figure 7.7.

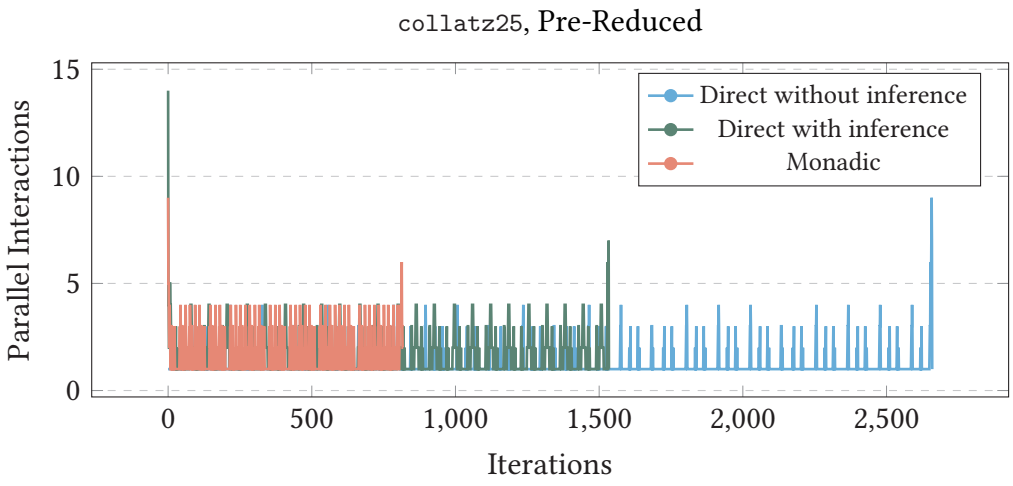


Figure 7.7 Comparison of the monadic style and the inferred and non-inferred direct style using the example of `collatz25`.

7.5.2 Join

As described in section 4.4, the CONJUNCTIVE FORK works best with token-passing semantics allowing the TOKEN to enter APPLICATIONS. We evaluate its performance only in the direct style.

We compare two implementations of $f(n,k) = \sum_{i=0}^{n} i^k$: An iterative version and a parallel one. The first one calculates the sum iteratively using a loop in a literal translation of f. The second one uses a DISJUNCTIVE FORK via a divide-and-conquer algorithm.

Both implementations in table 7.2 use pre-reduced recursion and are run with inference enabled.

				Rules				
	Inter.	Iter.	Ratio	Dup.	Ann.	Era.	Tok.	Eff.
Iterative	7680	4640	1.655	404	708	311	4341	1916
Parallel	28654	724	39.577	1891	2489	3594	15213	5467

Table 7.2 Comparison of reducing `sum_of_powers100_2_iterative` and `sum_of_powers100_2_parallel`.

It can be seen in table 7.2 how the parallel implementation requires fewer iterations, while in turn requiring more iterations. We visualize the potential for parallelism per iteration in figure 7.8.

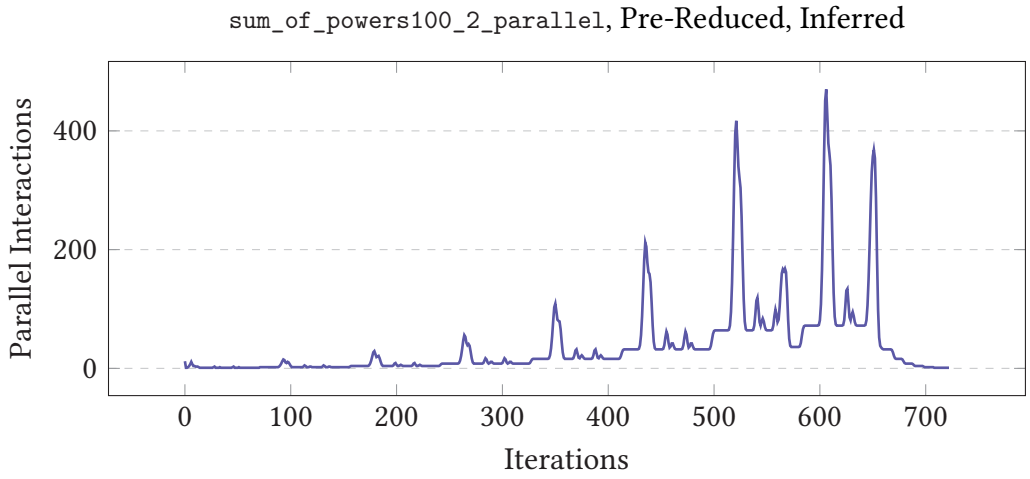


Figure 7.8 Available parallelism per iteration during the reduction of `sum_of_powers100_2_parallel`. The maximal parallel interactions of 470 occur in iteration 606.

With an average interaction count of 39.577 per iteration, the program will execute optimally with at least 40 parallel computing threads. Of course, this number would increase with larger n, as the divide and conquer algorithm creates new threads exponentially. However, from the nature of interaction nets it follows that the total *interaction* count will stay the same regardless, even if the program was run with fewer threads. We conjecture these results on parallelism to translate to any divide-and-conquer algorithm.

7.6 Discussion

The direct-style token-passing semantics extends Σ_{peff} with agents and interaction rules following the laws of interaction nets. As in the monadic style (proposition 6.2), we can therefore derive a general notion of strong confluence:

Proposition 7.2 (Strong Confluence of $\leftrightarrows$, Continuation of corollary 5.5)**.** *Assuming all asynchronously executed* ACTORS *to be orthogonal, the reduction relation* $\leftrightarrows$ *of* $\Sigma_{\text{pdir}} \setminus \{\vee\}$ *satisfies the diamond property modulo time.*

The presented token-passing allows embedding an execution order mirroring the Call-by-Value strategy into a λ-calculus without any strategy. The simplicity of the presented semantics should allow for embeddings of arbitrary execution strategies such as left-to-right Call-by-Value, Call-by-Name, or even *strong* execution (i. e. executing within ABSTRACTIONS). Existing work on token-passing semantics by Sinot [32, 38] describe potential ways of doing so.

By the Call-by-Value strategy we also achieve results on effectful optimality: As any ACTION of arity 0—even synchronous—is executed before substitution, their resulting terms are also shared maximally. This goes contrary to, say, a Call-by-Name strategy, where synchronous ACTIONS could be executed multiple times if they were substituted into multiple symbols.

Our strategy of embedding this semantics uses the concept of action potential with the aim of eliminating action potential in order to make existing reduction rules applicable. This idea is related to CBPV as introduced in section 3.1, where "forcing" a term corresponds to turning a term with action potential into a term without any. "Thunking" a term then corresponds to wrapping a term in action potential. Whereas this thunking would correspond more to the MONADIC UNIT, any TOKEN-reflecting term can also be seen as being thunked.

The relation between the direct and monadic style has been further enforced by the realization that, abstractly, the agents of both semantics interact the same up to a single interaction.

In our preliminary results, the inference reduces the interactions while increasing the available parallelism per iteration. With the arguably more elegant syntax of the direct style, the presented language and translations form an appropriate target for parallel yet effectful languages based on interaction nets, while at the same time preserving properties of optimality. Future work should investigate methods of further approaching the performance of the monadic-style semantics.

Chapter 8

Conclusion

In chapter 1, we described the pure λ-calculus, the polarized interaction combinators, and their relations. We went on to describe existing projects supporting side effects in interaction nets, as well as projects with related goals (chapter 2). Using both of these chapters as foundation, we then introduced an effectful extension of the λ-calculus in chapter 3, while also defining its properties and limitations. In chapter 4 we provided a simple language compiling back to the effectful λ-calculus. In the rest of this thesis we looked at solutions to the presented problems by introducing effectful agents in chapter 5, the corresponding monadic-style semantics (chapter 6), and finally the direct-style semantics (chapter 7). We backed the work on interaction nets with the parallel development of equivalent semantics for the λ-calculus.

In this chapter, we summarize the chapters of this thesis, discuss potential future work, and provide a final conclusion.

8.1 Summary

Optimal Effects In this chapter we presented the effectful λ-calculus Λ_{eff}. We presented a minimal extension without notions of reduction or execution order, such that the translation to interaction nets follows trivially. We described the problem of this minimalism using examples with side effects that influence each other. On the other hand, we showed the benefits of not prescribing an execution order using asynchronous ACTIONS, which may be used for ACTIONS commuting with all other ACTIONS ("orthogonality"). As Λ_{eff} can interact with the REAL WORLD, we have also introduced how recursion can be implemented via side effects. Finally, we linked back to the core of this thesis—optimality. In Λ_{eff}, TOKENS can be used to share orthogonal and idempotent ACTIONS. We have sketched how FORKS may be used to construct a hierarchy on such ACTIONS,

which stagger parallel execution iteratively. Importantly, we have argued how parallelism in the context of optimality becomes more appropriate with the addition of side effects.

A Simple Language Using the introduced Λ_{eff}, we could now derive a frontend language $\mathcal{L}_{\text{sim}}$ with support for definitions, branches, common data types, and *n*ary FORKS. We used these features to introduce various applications in the domain of distributed programming and foreign functions. Ultimately, we argue that $\mathcal{L}_{\text{sim}}$ increases the value of all presented encodings, as it makes its underlying theoretical results more obvious to programmers.

Effectful Agents To our knowledge, we presented the first formalization of side effects and their interactions in the context of interaction nets. We also used the work of Sinot [32] in a novel way by repurposing his tokens used for reduction to cause side effects by executing actions. We described all features introduced for Λ_{eff} for interaction nets, with the disjunctive FORK requiring a non-deterministic extension. We have further argued how FORKS could provide a communication mechanism not directly implementable in the λ-calculus but via channels of the π-calculus. Finally, preliminary results on interaction counts indicate no surprising overhead due to our approach of handling partial application and execution via TOKENS.

Monadic Style The monadic style represents the first approach to the token-passing semantics required for sequentializing side effects in Λ_{eff} and interaction nets. In order to back the work on interaction nets up with formalizations in the λ-calculus, we described the extended languages $\mathcal{L}_{\text{mon}}$ and Λ_{mon}. To our knowledge, the use of tokens to describe the execution order of monads is novel. We proved its correctness in the sense of fulfilling the monad laws. We concluded the formalization by providing translations from the λ-calculus to interaction nets. Finally, we gave an initial numeric estimate of the performance using several example programs. Since TOKENS may never leave the monadic construct, the token-passing semantics does not interfere with reduction. Our benchmarks confirmed this theory, as the potential for parallelism remains high for our set of test programs.

Direct Style The second approach to the token-passing semantics aimed for a more intuitive syntax by orienting on imperative programming languages. We argued how token-passing can also be seen as an implicit world-state passing style. Since the direct style does not have syntactic constructs restricting the

potential for side effects, all terms were then assumed to be effectful. The presented token-passing semantics correspondingly visits every term once, leading to a token-passing overhead. We derived a runtime inference of action potential by making use of the rotating REDIRECTOR agents. We furthermore argued how agents make the relation between the direct and monadic style obvious, as the agents behave almost equivalently. Our preliminary practical results confirmed its properties: The direct style without inference effectively linearizes the entire reduction, thus barely having any available parallelism. The inference then regains some parallelism via inference interactions, which result in less total interactions and iterations. Still, in general the direct-style semantics requires more interactions than the monadic style, as the inference procedure by itself results in a token-passing overhead.

8.2 Future Work

Formalizations In this initial work we were unable to provide formal proofs of most discussed properties. In the aspect of correctness, proofs of our token-passing semantics and inference semantics would allow further use in production code. This comes in addition to the requirement of general estimates of efficiency and interaction overhead. Those are especially relevant for formalizations of effectful optimality and whether our proposed solutions fulfill these formalizations.

Type Systems We have argued several times how type systems could be used to improve upon our work. In Λ_{eff}, assigning types to DATA could provide a verification beyond only checking whether the arity of an action is respected. Types could further allow overloading ACTIONS to different behavior. In the monadic style, types are technically required for the fulfillment of the monad laws, as otherwise the return of a COTOKEN may not be guaranteed. It further requires investigation how such necessity for returning TOKENS may be described in a type system. In the direct style, we have provided a proof-of-concept runtime inference algorithm. With the addition of a typed *effect system* [34] such inference could be accomplished more elegantly, possibly yielding even more potential for parallel reductions.

Communication Contrary to the λ-calculus, interaction nets allow connections between arbitrary parts of a net. In the context of FORK agents, we argued how such connections may be used to establish communications between threads. Developments in the area of the π-calculus and other process calculi seem like a promising area of research which could benefit from interaction nets.

Specifically projects such as λ(fut) [48] aiming to merge features of process calculi into the λ-calculus, seem relevant. As we have introduced ACTIONS also as a way of communicating between different computers—say by receiving functions as terms from the internet—future research could investigate how wires of interaction nets may be established between remote connections such that they can keep interacting along this wire.

Abstract Machines Abstract machines model the execution of computational models such that they can be analyzed easier, while also allowing translations to hardware or simulations. As abstract machines sometimes use evaluation "tokens" to track the reduction state [30], integration of our work could allow elegant notions of executing actions in such machines. Future work could further investigate reduction and execution via existing parallel and concurrent machines for interaction nets [49, 50]. The relation to the work on Geometry of Interaction (GoI) [39] should also be investigated, as it also uses token-like traversal on a graph-like model of computation. GoI allows for abstract machines even with embedded Call-by-Value semantics [40, 41].

As the linear substitution calculus is also deemed to be an optimal reduction system [51], it would be interesting to see whether our notion of effects could be integrated in its abstract machines [52] as well.

Power of Actors As described, ACTORS provide an elegant way of interacting with the REAL WORLD. We have only used this functionality for embedding side effects. Yet, we believe that ACTORS may be useful beyond that. For one, the agents proposed by Salikhmetov [23] used for "embedded read-back" could be translated to ACTORS. This way, the direct-style token-passing semantics could yield the read-back normal form of a λ-term directly via side effect, with multiple TOKENS requiring an additional kind of output synchronization. In fact, any related process implemented via a term-traversing evaluation function may be translated to ACTORS with a matching token-passing semantics.

Efficiency The goal of our implementation was not efficiency but accuracy. Future work should investigate the efficient implementation of our proposed agents and interactions. Current work on efficiency is dominated by the HVM [4] as it targets the GPU. It uses a small set of agents allowing sophisticated optimizations. As we have argued about the irrelevance of built-in agents using the power of ACTORS, efficient integration of such ACTORS could provide further optimizations. We imagine an implementation where each ACTOR resembles raw machine code such that the execution initiated by a TOKEN is only a matter of assigning the program pointer of the attached thread to the respective code snippets. The

strict sequentiality required to execute multiple snippets of machine code can only be guaranteed by our very strict separation of pure and impure code.

Real-Time We have argued in our propositions of strong confluence how it only holds generally when the precise point in time of execution is ignored. This is because without prescribing a certain reduction order, reductions of non-effectful active pairs (such as garbage collection) can be prioritized—thus delaying the execution of ACTORS. For tasks requiring real-time interaction with the REAL WORLD, the latency of execution in parallel reduction should be as minimal as using a sequential reduction method. Future research should go into regaining some guarantee of real-time execution, for example by prioritizing reductions maximizing the passing of the TOKEN agent in a deterministic manner.

Oracle As discussed in section 1.3.6, an oracle is required to support all terms of the λ-calculus. Although we do not see any reason in how the addition of such an oracle may conflict with our addition of side effects and TOKENS, we have not demonstrated as such. The addition of delimiter agents and labels as proposed by van Oostrom; van de Looij; Zwitserlood [12] would be the straightforward way of doing so. However, our use of TOKENS enables the applicability of a recent approach by Salikhmetov [23]: There, an EVAL agent resembling Sinot's token is used in combination with a "waiting construct" to provide β-optimality.

8.3 Conclusion

In this thesis we explored the question of embedding sequential execution in the otherwise non-sequential system of interaction nets, while also considering the optimality property. We set the focus on an effectful extension of the λ-calculus that we translated to interaction nets. We further aimed for realistic use cases by developing a minimal programming language.

While doing so, we provided a token-based understanding of the execution of effectful terms. This token-based approach allows parallel as well as sequential execution independent of reduction. We have presented ways of explicitly initiating parallel execution, corresponding to either racing or joining threads. In comparison to existing work, our approach is flexible in the number of arguments, implementations, reduction strategies, and available parallelism.

We derived two token-passing semantics which may be used for executing effectful terms in a specific order. Both of the approaches have respective limitations and benefits. The monadic style comes with a syntactic overhead, while the direct style initially results in more interactions and less parallelism. By translation to interaction nets we have found strong similarities between both

approaches, effectively differing in only one interaction rule. The direct style emerges from generalizing the monadic agents to all token-redirecting agents. The lost parallelism can then be regained by inferring the rotation of agents connected to pure or effectful agents. We demonstrated this behavior in practice by evaluating and comparing the individual approaches.

Regarding optimality, we have not left the paradigm of the interaction nets—thus maintaining existing results of β-optimality. We further discussed an initial notion of optimality in the context of side effects, therefore establishing the concept of sharing effectful terms when they are asynchronous, idempotent, and orthogonal.

In summary, our method of sequentializing execution is considered feasible and merits further investigation.

Bibliography

1. LAFONT, Yves. Interaction Nets. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '90*. San Francisco, California, United States, 1990, pp. 95–108. Available from DOI: 10.1145/96709.96718.
2. ASPERTI, Andrea; GUERRINI, Stefano. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1999. ISBN 978-0-06-081542-4.
3. LAWALL, Julia L.; MAIRSON, Harry G. Optimality and Inefficiency: What Isn't a Cost Model of the Lambda Calculus? *SIGPLAN Not.* 1996, vol. 31, no. 6, pp. 92–101. Available from DOI: 10.1145/232629.232639.
4. TAELIN, Victor. *HVM2: A Parallel Evaluator For Interaction Combinators* **online**2024. [visited on 2025-07-31]. Available from: https://raw.githubusercontent.com/HigherOrderCO/HVM/main/paper/HVM2.pdf.
5. BARENDREGT, Hendrik Pieter. *The Lambda Calculus: Its Syntax and Semantics*. Vol. 103. Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co., 1984. ISBN 0-444-86748-1.
6. HUET, Gérard. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *J. ACM*. 1980, vol. 27, no. 4, pp. 797–821. Available from DOI: 10.1145/322217.322230.
7. LAFONT, Yves. Interaction Combinators. *Information and Computation*. 1997, vol. 137, no. 1, pp. 69–101. Available from DOI: 10.1006/inco.1997.2643.
8. FERNÁNDEZ, Maribel; MACKIE, Ian. A Calculus for Interaction Nets. In: NADATHUR, Gopalan (ed.). *Principles and Practice of Declarative Programming*. Berlin, Heidelberg: Springer, 1999, pp. 170–187. Available from DOI: 10.1007/10704567_10.
9. MAZZA, Damiano. *Interaction Nets: Semantics and Concurrent Extensions*. 2006. Université Aix-Marseille II/Università degli Studi Roma Tre.

10. GONTHIER, Georges; ABADI, Martín; LÉVY, Jean-Jacques. The Geometry of Optimal Lambda Reduction. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1992, pp. 15–26. POPL '92. Available from DOI: 10.1145/143165.143172.

11. [SW] ROCHEL, Jan, Graph-Rewriting-Lambdascope version 0.5.11, 2024. URL: https://hackage.haskell.org/package/graph-rewriting-lambdascope.

12. Van OOSTROM, Vincent; van de LOOIJ, Kees-Jan; ZWITSERLOOD, Marijn. Lambdascope: Another Optimal Implementation of the Lambda-Calculus. In: *Workshop on Algebra and Logic on Programming Systems (ALPS)*. 2004.

13. LAMPING, John. An Algorithm for Optimal Lambda Calculus Reduction. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '90*. San Francisco, California, United States: ACM Press, 1990, pp. 16–30. Available from DOI: 10.1145/96709.96711.

14. STAPLES, John. *Efficient Evaluation of Lambda Expressions: A New Strategy*. University of Queensland. Department of Computer Science, 1980.

15. ASPERTI, Andrea; GIOVANNETTI, Cecilia; NALETTO, Andrea. The Bologna Optimal Higher-Order Machine. *Journal of Functional Programming*. 1996, vol. 6, no. 6, pp. 763–810. Available from DOI: 10.1017/S0956796800001994.

16. MACKIE, Ian. YALE: Yet Another Lambda Evaluator Based on Interaction Nets. *ACM SIGPLAN Notices*. 1998, vol. 34, no. 1, pp. 117–128. Available from DOI: 10.1145/291251.289434.

17. MACKIE, Ian. Efficient λ-Evaluation with Interaction Nets. In: van OOSTROM, Vincent (ed.). *Rewriting Techniques and Applications*. Berlin, Heidelberg: Springer, 2004, pp. 155–169. Available from DOI: 10.1007/978-3-540-25979-4_11.

18. FERNÁNDEZ, Maribel; MACKIE, Ian. Interaction Nets and Term-Rewriting Systems. *Theoretical Computer Science*. 1998, vol. 190, no. 1, pp. 3–39. Available from DOI: 10.1016/S0304-3975(97)00082-0.

19. ASPERTI, Andrea; LANEVE, Cosimo. Interaction Systems I: The Theory of Optimal Reductions. *Mathematical Structures in Computer Science*. 1994, vol. 4, pp. 457–504. Available from DOI: 10.1017/S0960129500000566.

20. LÉVY, Jean-Jacques. Optimal Reductions in the Lambda Calculus. *To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. 1980, pp. 159–191.

21. ASPERTI, Andrea; COPPOLA, Paolo; MARTINI, Simone. (Optimal) Duplication Is Not Elementary Recursive. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2000, pp. 96–107. POPL '00. Available from DOI: 10.1145/325694.325707.

22. ACHTEN, Peter; PLASMEIJER, Rinus. The Ins and Outs of Clean I/O. *Journal of Functional Programming*. 1995, vol. 5, no. 1, pp. 81–110. Available from DOI: 10.1017/S0956796800001258.

23. SALIKHMETOV, Anton. Token-Passing Optimal Reduction with Embedded Read-Back. *Electronic Proceedings in Theoretical Computer Science*. 2016, vol. 225, pp. 45–54. Available from DOI: 10.4204/EPTCS.225.7.

24. GAY, Simon J. *Interaction Nets*. 1991. University of Cambridge Computer Laboratory.

25. ACHTEN, Peter; van GRONINGEN, John; PLASMEIJER, Rinus. High Level Specification of I/O in Functional Languages. In: LAUNCHBURY, John; SANSOM, Patrick (eds.). *Functional Programming, Glasgow 1992*. London: Springer, 1993, pp. 1–17. Available from DOI: 10.1007/978-1-4471-3215-8_1.

26. MOGGI, E. Computational Lambda-Calculus and Monads. In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*. Pacific Grove, CA, USA: IEEE Comput. Soc. Press, 1989, pp. 14–23. Available from DOI: 10.1109/LICS.1989.39155.

27. PEYTON JONES, Simon L.; WADLER, Philip. Imperative Functional Programming. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1993, pp. 71–84. POPL '93. Available from DOI: 10.1145/158511.158524.

28. JONES, Simon Peyton. Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. *Engineering Theories of Software Construction, Marktoberdorf Summer School*. 2001, pp. 47–96.

29. LEVY, Paul Blain. *Call-by-Push-Value*. 2001. Queen Mary and Westfield College, University of London.

30. ACCATTOLI, Beniamino; LAGO, Ugo Dal; VANONI, Gabriele. *The Abstract Machinery of Interaction (Long Version)*. 2020-07-09. Available from DOI: 10.48550/arXiv.2002.05649.

31. AHMAN, Danel; PRETNAR, Matija. Asynchronous Effects. *Proc. ACM Program. Lang.* 2021, vol. 5. Available from DOI: 10.1145/3434305.

32. SINOT, François-Régis. Call-by-Name and Call-by-Value as Token-Passing Interaction Nets. In: URZYCZYN, Paweł (ed.). *Typed Lambda Calculi and Applications*. Berlin, Heidelberg: Springer, 2005, pp. 386–400. Available from DOI: 10.1007/11417170_28.

33. [SW] MARLOW, Simon, Async version 2.2.5, 2023. URL: https://hackage.haskell.org/package/async-2.2.5/[visited on 2025-05-09].

34. LUCASSEN, J. M.; GIFFORD, D. K. Polymorphic Effect Systems. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1988, pp. 47–57. POPL '88. Available from DOI: 10.1145/73560.73564.

35. MINSKY, Yaron; MADHAVAPEDDY, Anil; HICKEY, Jason. *Real World OCaml: Functional Programming for the Masses*. 1st ed. O'Reilly Media, Inc., 2013. ISBN 978-1-4493-2391-2.

36. FERNÁNDEZ, Maribel; KHALIL, Lionel. Interaction Nets with McCarthy's Amb. *Electronic Notes in Theoretical Computer Science*. 2002, vol. 68, no. 2, pp. 51–68. Available from DOI: 10.1016/S1571-0661(05)80363-9.

37. MCCARTHY, John. A Basis for a Mathematical Theory of Computation. In: *Studies in Logic and the Foundations of Mathematics*. 1959, vol. 26, pp. 33–70. ISBN 978-0-444-53391-3.

38. SINOT, François-Régis. Token-Passing Nets: Call-by-Need for Free. *Electronic Notes in Theoretical Computer Science*. 2006, vol. 135, no. 3, pp. 129–139. Available from DOI: 10.1016/j.entcs.2005.09.027.

39. GIRARD, Jean-Yves. Geometry of Interaction 1: Interpretation of System F. In: FERRO, R.; BONOTTO, C.; VALENTINI, S.; ZANARDO, A. (eds.). *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1989, vol. 127, pp. 221–260. Logic Colloquium '88. Available from DOI: 10.1016/S0049-237X(08)70271-4.

40. MACKIE, Ian. The Geometry of Interaction Machine. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1995, pp. 198–208. POPL '95. Available from DOI: 10.1145/199448.199483.

41. FERNÁNDEZ, Maribel; MACKIE, Ian. Call-by-Value λ-Graph Rewriting without Rewriting. In: *International Conference on Graph Transformation*. Springer, 2002, pp. 75–89. Available from DOI: 10.1007/3-540-45832-8_8.

42. ALEXIEV, Vladimir. *Non-Deterministic Interaction Nets*. 1999. Available from DOI: 10.7939/R3MW28R67. University of Alberta.

43. JACOBS, Jules. *Guarantees by Construction*. 2024. Available from DOI: 10.54195/9789493296541. Radboud University Press.

44. MÉNDEZ-LOJO, Mario; NGUYEN, Donald; PROUNTZOS, Dimitrios; SUI, Xin; HASSAAN, M. Amber; KULKARNI, Milind; BURTSCHER, Martin; PINGALI, Keshav. Structure-Driven Optimizations for Amorphous Data-Parallel Programs. In: *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: Association for Computing Machinery, 2010, pp. 3–14. PPoPP '10. Available from DOI: 10.1145/1693453.1693457.

45. JIRESCH, Eugen Robert Winfried. *Extending Interaction Nets towards the Real World*. 2012. Technische Universität Wien.

46. BÉCHET, Denis; LIPPI, Sylvain. Hard Combinators. *Electronic Notes in Theoretical Computer Science*. 2008, vol. 203, no. 1, pp. 31–48. Available from DOI: 10.1016/j.entcs.2008.03.032.

47. ALMEIDA, José Bacelar; PINTO, Jorge Sousa; VILAÇA, Miguel. Token-Passing Nets for Functional Languages. *Electronic Notes in Theoretical Computer Science*. 2008, vol. 204, pp. 181–198. Available from DOI: 10.1016/j.entcs.2008.03.061.

48. NIEHREN, Joachim; SCHWINGHAMMER, Jan; SMOLKA, Gert. A Concurrent Lambda Calculus with Futures. In: GRAMLICH, Bernhard (ed.). *Frontiers of Combining Systems*. Berlin, Heidelberg: Springer, 2005, pp. 248–263. Available from DOI: 10.1007/11559306_14.

49. PEDICINI, Marco; PELLITTA, Giulio; PIAZZA, Mario. Sequential and Parallel Abstract Machines for Optimal Reduction. In: *Preproceedings of the 15th Symposium on Trends in Functional Programming (TFP2014)*. 2014.

50. PINTO, Jorge Sousa. Sequential and Concurrent Abstract Machines for Interaction Nets. In: TIURYN, Jerzy (ed.). *Foundations of Software Science and Computation Structures*. Berlin, Heidelberg: Springer, 2000, pp. 267–282. Available from DOI: 10.1007/3-540-46432-8_18.

51. BARENBAUM, Pablo; BONELLI, Eduardo. Optimality and the Linear Substitution Calculus. In: MILLER, Dale (ed.). *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017, vol. 84, 9:1–9:16. ISSN 1868-8969. Available from DOI: 10.4230/LIPIcs.FSCD.2017.9.

52. ACCATTOLI, Beniamino; BARENBAUM, Pablo; MAZZA, Damiano. Distilling Abstract Machines. *SIGPLAN Not.* 2014, vol. 49, no. 9, pp. 363–376. Available from DOI: 10.1145/2628136.2628154.

53. ROCHEL, Jan. *Port Graph Rewriting in Haskell*. 2011-02-06. URL: http://rochel.info/docs/graph-rewriting.pdf.

Appendix A

Implementation & Usage

In this chapter we describe our implementation of parts of the presented languages from a high-level perspective. We further provide information on how to use the included tools and reproduce our results.

The full implementation can be found on:

- `https://github.com/se-tuebingen/thesis-optimal-effects/tree/main/impl` (submitted)
- `https://github.com/marvinborner/optimal-effects/tree/bachelor` (public)

A.1 Usage

The implementation is written in Haskell and uses the Cabal package manager. Therefore, the installation of GHC and Cabal is required. We have used GHC 9.10.1 and Cabal 3.10.3.0 for development and benchmarking. The project requires a GHC version greater than 9.8.1. In order to use the visualizations, the libraries of the OpenGL utility toolkit are required (e. g. `freeglut3-dev`).

To install, type in a POSIX shell:

```
cabal build                       # to build the project
cabal run opteff-exe -- --help    # to show the help screen
cabal install                     # to install `opteff-exe`
```

Programs written in $\mathcal{L}_{\text{sim}}$ can be parsed by piping into `stdin`:

```
cat samples/direct/fac.front | cabal run opteff-exe
cat samples/direct/fac.front | opteff-exe
```

There exist three possible modes:

```
-v,--visualize    Visualize reduction interactively (default).
-b,--benchmark    Benchmark reduction until normal form.
-c,--count        Count iterations until normal form.
```

The *target* language can be set using `-t,--target` and must either be "direct" or "monad" (defaults to "direct"). If the direct-style semantics are chosen, the `-i,--infer` flag specifies whether runtime inference of action potential should be used (section 7.4.4).

As the token-passing semantics must be specified, the samples from chapter 4 may be tested using the "monad" target. As long as the monadic constructs are not used, the "monad" target corresponds to the effectful agents in chapter 5 and does not enforce any token-passing semantics.

The benchmark mode allows the use of two additional flags:

```
-r,--random    Apply rules in random order.
               Should result in constant interaction counts.
-p,--parallel  Simulate parallel reduction by applying rules
               exhaustively instead of once.
               Counts all possible parallel reductions as 1.
               Will not have constant interaction counts with -r.
```

A.2 User Interface

We use the same user interface as used by Rochel [11], extended with our additional agents and some visual enhancements. The window opening when `--visualize` is used consists of:

Rule Tree at the top left. All nested rules will be applied upon right-clicking an element of the tree. By left-clicking, all matching active pairs will be highlighted in red.

Canvas at the rest of the window. You can zoom in or out using the scroll-wheel. You can move the canvas by left-clicking while moving the mouse. Any agent can be moved around by left-clicking. Any highlighted active pair can be reduced by right-clicking.

Pausing Pressing `space` or right-clicking pauses the animation.

Agents As presented in section 1.2. Principal ports are marked with a circle. They can be at any port instead of only at the top.

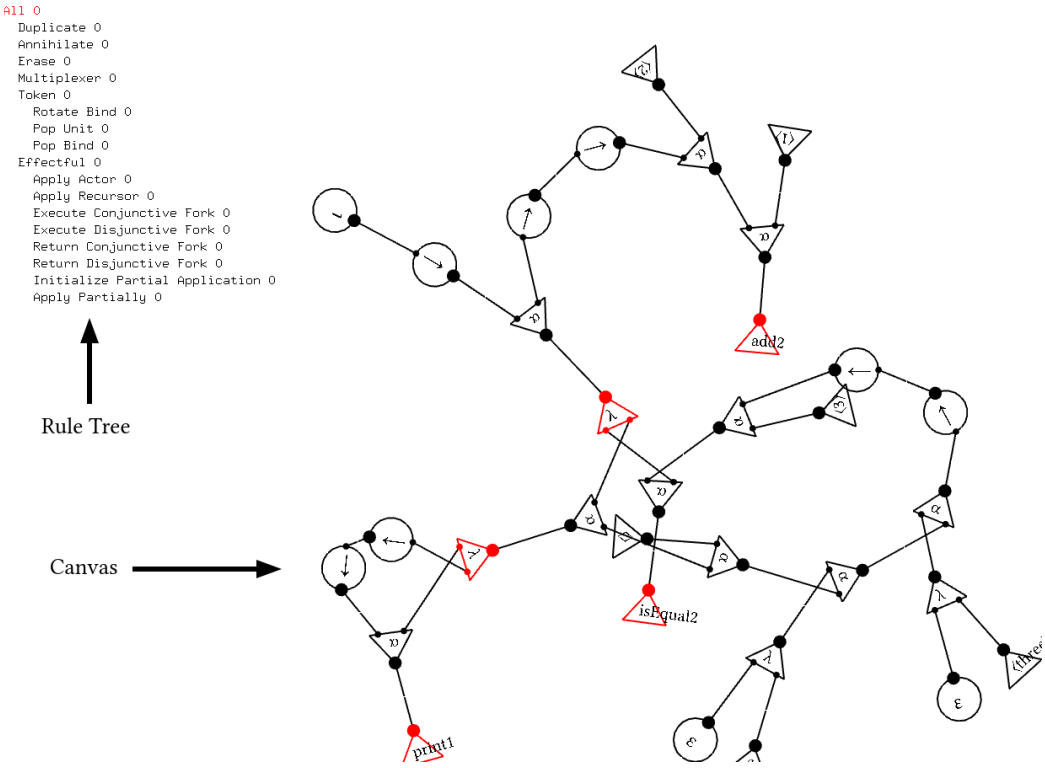


Figure A.1 The user interface, visualizing listing 4.

A.3 Structure

We have forked the `graph-rewriting` library by Rochel [53]—also used by Lambdascope [11]. The code related to the pure, deterministic parts of interaction nets and the pure λ-calculus, as well as the graph rewriting and UI code were in part written by Rochel [11] for Lambdascope. Such files or code snippets are marked accordingly.

The modules in the `src/` directory are structured the following way:

`GraphRewriting` The fork of `graph-rewriting` [53].

`Data.View` An internal data structure of the `GraphRewriting` library.

`Data.Lambda` The effectful λ-calculus defined in definition 3.1 and definition 6.2.

`Data.Direct` The direct-style interaction net Σ_{pdir} defined in definition 7.6.

`Data.Monad` The monadic-style interaction net Σ_{pmon} defined in definition 6.5.

`Data.Front` The frontend language defined in definition 4.1.

`Data.Effects` The definitions and data structures for built-in effects.

`Language.Direct.GL` The graphics definitions for the visualizer of Σ_{pdir}.

`Language.Direct.Reducer` The high-level reduction functions of Σ_{pdir}.

`Language.Direct.Rules` The interaction rules of Σ_{pdir} defined in section 7.4.2.

`Language.Monad.GL` The graphics definitions for the visualizer of Σ_{pmon}.

`Language.Monad.Reducer` The high-level reduction functions of Σ_{pmon}.

`Language.Monad.Rules` The interaction rules of Σ_{pmon} defined in section 6.4.

`Language.Generic.Effects` The handling of the built-in side effects, mocked by returning constant data.

`Language.Generic.Nodes` The declarations of generic agents existing in both the direct and monadic style.

`Language.Generic.Rules` The generic interaction rules of Σ_{pol} defined in definition 1.11.

`Language.Lambda.Transformer.Direct` The translation of Λ_{eff} to Σ_{pdir} defined in definition 7.7.

`Language.Lambda.Transformer.Monad` The translation of Λ_{mon} to Σ_{pmon} defined in definition 6.7.

`Language.Front.Transformer.Lambda` The translation of $\mathcal{L}_{\text{sim}}$ (and $\mathcal{L}_{\text{mon}}$) to Λ_{eff} (and Λ_{mon}) defined in definition 4.2 (and definition 6.6).

`Language.Front.Parser` The parser for the syntax of $\mathcal{L}_{\text{sim}}$ defined in definition 4.1.

We have not implemented token-passing semantics for Λ_{eff} or Λ_{mon}.

A.4 Tests & Samples

The presented code has corresponding implementations in `samples/`, as shown in table A.1.

listing 4	`nopassing/dependent.front`
listing 5	`nopassing/angelic_pong.front`
listing 6	`direct/read_join.front`
listing 7	(not implemented)
listing 8	`nopassing/ffi_local.front`
listing 9	`monad/static_two.front`
example 6.9	`monad/repl.front`
listing 11	`direct/fac_interactive.front`
figure 7.5	`direct/repl.front`

Table A.1 Code listings and their corresponding implementations in `samples/`.

We provide `<file>.check` files for most samples. Their content corresponds to running `opteff-exe --bench [--infer] --target <direct|monad>` respectively. The `./test` shell scripts in the `samples/*` directories run the samples and compare the outputs. Additionally, the samples get executed repeatedly using the `--random` flag in order to verify strong confluence.

The benchmarks used in section 5.7, section 6.7, and section 7.5 are found in the `bench/` directories.

A.5 Differences

The implementation diverges from our specifications in several ways:

Semicolons The $\mathcal{L}_{\text{sim}}$ language (`.front`) does not support semicolons as statement delimiter. It uses newlines instead.

Mocking All side effects are mocked and deterministic. `readInt` will always return the same number, for example. The interpreters of the `evalLang` ACTION are executed using `unsafePerformIO` and are therefore expected to behave deterministically as well.

Pure Data All constant data in $\mathcal{L}_{\text{sim}}$ is translated directly to DATA instead of (asynchronous) ACTIONS. This is possible because of mocking and our high-level interpretation. Code for $\mathcal{L}_{\text{sim}}$ in `.front` files does not require the use of "!". We therefore do not count the additional interactions in our benchmarks.

Determinism As our entire implementation is deterministic, `race` will always return the leftmost thread returning a COTOKEN during benchmarking. Even when interaction rules are applied in random order (`--random`), the active pairs will still be reduced in a fixed order.

Polarization The implementation does not use the concept of polarization. Instead, agents of different polarization are encoded as different agents entirely, or by having an additional property (e. g. `direction :: AppDir` for REDIRECTORS, or `exec :: Bool` for MONADIC BINDS)

Forks Our FORKS do not use auxiliary agents as proposed in section 5.5. Instead, we exploit the flexibility of the `graph-rewriting` library and reduce the CONJUNCTIVE FORK only when two COTOKENS are connected—which is not otherwise a valid interaction rule.

Recursor The implementation uses different agents for ACTORS expanding to recursive nets—the RECURSOR. This is because RECURSORS hold a boxed λ-term, which is not compatible with the way we store side effect instructions in ACTORS. We have not implemented the additional transformations for the direct style discussed in section 7.4.5. In our implementation, the recursor always expands to a pre-reduced net, as defined by `prereduceRules` in `irect,Monad`src/Language/D/Rules.hs.

Anonymous Functions The `.front` files support a syntax for anonymous functions. Abstractions are written as $[M]$. The abstraction can be referred to by de Bruijn index, written as $\$n$. For example: $[\![$`[[[$2 ($1 $0)]]]`$]\!] = \lambda f g x.f\ (g\ x)$.

Force The FORCE symbol "!" always translates to two TOKENS, even if an asynchronous ACTION does not reflect any TOKEN.

Asynchronous If By default, all IF EXPRESSIONS are thunked and applied with a UNIT as discussed in section 4.2. In the direct style, this prevents both sides of the branch being evaluated regardless of whether it is true or false. If instead both sides should be executed, `.front` files support `if!` expressions, which translate directly to Church conditionals. We only have this difference in the implementation as we translate $\mathcal{L}_{\text{sim}}$ to Λ_{eff} independent of the selected semantics.

Parts of this work were submitted and accepted to the 30th ACM International Conference on Functional Programming 2025 Student Research Competition (ICFP 2025 SRC) in form of an extended abstract.

Erklärung

Laut Beschlüssen der Prüfungsausschüsse Bioinformatik, Informatik, Informatik Lehramt, Kognitionswissenschaft, Machine Learning, Medieninformatik und Medizininformatik der Universität Tübingen vom 05.02.2025. Gültig für Abschlussarbeiten (B.Sc./M.Sc./B.Ed./M.Ed.) in den zugehörigen Fächern. Bei Studienarbeiten und Hausarbeiten bitte nach Maßgabe des/der jeweiligen Prüfers/Prüferin.

1. Allgemeine Erklärungen

Hiermit erkläre ich:

- Ich habe die vorgelegte Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.
- Ich habe alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet.
- Die Arbeit war weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens.
- Falls ich ein elektronisches Exemplar und eines oder mehrere gedruckte und gebundene Exemplare eingereicht habe (z.B., weil der/die Prüfer/in(nen) dies wünschen): Das elektronisch eingereichte Exemplar stimmt exakt mit dem bzw. den von mir eingereichten gedruckten und gebundenen Exemplar(en) überein.

2. Erklärung bezüglich Veröffentlichungen

Eine Veröffentlichung ist häufig ein Qualitätsmerkmal (z.B. bei Veröffentlichung in Fachzeitschrift, Konferenz, Preprint, etc.). Sie muss aber korrekt angegeben werden. Bitte kreuzen Sie die für Ihre Arbeit zutreffende Variante an:

☐ Die Arbeit wurde bisher weder vollständig noch in Teilen veröffentlicht.

☒ Die Arbeit wurde in Teilen oder vollständig schon veröffentlicht. Hierfür findet sich im Anhang eine vollständige Tabelle mit bibliographischen Angaben.

3. Nutzung von Methoden der künstlichen Intelligenz (KI, z.B. chatGPT, DeepL, etc.)

Die Nutzung von KI kann sinnvoll sein. Sie muss aber korrekt angegeben werden und kann die Schwerpunkte bei der Bewertung der Arbeit beeinflussen. Bitte kreuzen Sie alle für Ihre Arbeit zutreffenden Varianten an und beachten Sie, dass die Varianten 3.4 - 3.6 eine vorherige Absprache mit dem/der Betreuer/in voraussetzen:

☒ 3.1. Keine Nutzung: Ich habe zur Erstellung meiner Arbeit keine KI benutzt.

☐ 3.2. Korrektur Rechtschreibung & Grammatik: Ich habe KI für Korrekturen der Rechtschreibung und Grammatik genutzt, ohne dass es dabei zu inhaltlich relevanter Textgeneration oder Übersetzungen kam. Das heißt, ich habe von mir verfasste Texte in derselben Sprache korrigieren lassen. Es handelt sich um rein sprachliche Korrekturen, sodass die von mir ursprünglich intendierte Bedeutung nicht wesentlich verändert oder erweitert wurde. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

☐ 3.3. Unterstützung bei der Softwareentwicklung: Ich habe KI als Unterstützung beim Schreiben von Code in der Softwareentwicklung genutzt. Es handelt sich hierbei lediglich um Unterstützung und nicht um die automatische Generierung von größeren Programm-Teilen. Im Zweifelsfall habe ich mich mit meinem/r Betreuer/in besprochen. Alle genutzten Programme mit Versionsnummer sind im Anhang meiner Arbeit in einer Tabelle aufgelistet.

☐ 3.4. Übersetzung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Übersetzung von mir in einer anderen Sprache geschriebenen Texte genutzt. Jede derartige Übersetzung ist im laufenden Text gekennzeichnet und der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller übersetzten Textstellen und der verwendeten Programme mit Versionsnummer.

☐ 3.5. Code-Generierung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Erzeugung von Code in der Softwareentwicklung genutzt. Der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller derartigen Nutzungen, der verwendeten Programme mit Versionsnummer und der verwendeten Prompts.

☐ 3.6. Text-Generierung: Ich habe *nach vorheriger Absprache und mit Erlaubnis meines/r Betreuer/in* KI zur Erzeugung von Text in meiner Arbeit genutzt. Jede derartige Verwendung von KI ist im laufenden Text gekennzeichnet und der Anhang meiner Arbeit enthält eine Tabelle mit einem vollständigen Nachweis aller derartigen Nutzungen, der verwendeten Programme mit Versionsnummer und der verwendeten Prompts.

Falls ich in irgendeiner Form KI genutzt haben (siehe oben), dann erkläre ich:

Mir ist bewusst, dass ich die Verantwortung trage, falls es durch die Verwendung von KI zu fehlerhaften Inhalten, zu Verstößen gegen das Datenschutzrecht, Urheberrecht oder zu wissenschaftlichem Fehlverhalten (z.B. Plagiaten) kommt.

4. Abschluss und Unterschrift(en)

Mir ist bekannt, dass ein Verstoß gegen diese Erklärung prüfungsrechtliche Konsequenzen haben und insbesondere dazu führen kann, dass die Prüfungsleistung mit „nicht ausreichend" bzw. die Studienleistung mit „nicht bestanden" bewertet wird und bei mehrfachem oder schwerwiegendem Täuschungsversuch eine Exmatrikulation erfolgen bzw. ein Verfahren zur Entziehung eines eventuell verliehenen akademischen Titels eingeleitet werden kann.

Marvin Borner	30.07.2025, Tübingen	M. Bam
Vorname, Nachname Student/in	Ort, Datum	Unterschrift

Die Punkte 3.4 - 3.6 erfordern eine Zustimmung des/r Betreuer/in. Sollten Sie einen dieser Punkte angekreuzt haben, dann sollte der/die Betreuer/in bitte hier unterschreiben:

Ich habe der oben genannten Nutzung von KI zur Erstellung der Arbeit zugestimmt.

-	-	
Vorname, Nachname Betreuer/in	Ort, Datum	Unterschrift