DEPARTMENT OF
COMPUTER SCIENCE

JOANA SOARES FARIA
BSc in Computer Science and Engineering

AN OPTIMIZING COMPILER FOR A SESSION-TYPED FUNCTIONAL LANGUAGE

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2023

DEPARTMENT OF
COMPUTER SCIENCE

AN OPTIMIZING COMPILER FOR A SESSION-TYPED FUNCTIONAL LANGUAGE

JOANA SOARES FARIA

BSc in Computer Science and Engineering

Adviser: Bernardo Parente Coutinho Fernandes Toninho
Assistant Professor, NOVA School of Science and Technology | FCT NOVA

Examination Committee

Chair: Vítor Manuel Alves Duarte
Assistant Professor, NOVA School of Science and Technology | FCT NOVA

Rapporteur: Alcides Miguel C. Aguiar Fonseca
Associate Professor, Faculdade de Ciências da Universidade de Lisboa

Adviser: Bernardo Parente Coutinho Fernandes Toninho
Assistant Professor, NOVA School of Science and Technology | FCT NOVA

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2023

An Optimizing Compiler for a Session-typed Functional Language

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.0.3) [1].

For all those who believed in me,
and to those who doubted.

Acknowledgements

I’d like to start by thanking my advisor, Bernardo Toninho, for giving me the opportunity to work with him. I’d like to thank him for all his patience with so many doubts and questions, for being unquestionably the best teacher I’ve had during this academic journey. Thank you for going above and beyond, for being an excellent teacher and for being the best mentor anyone could ask for.

I would also like to thank all those who have supported me during my academic journey, without whom none of this would be possible: my mother, Ana, for her unconditional support, for always fighting for me, for all the tears she dried, for all the hugs and cuddles, for all the silliness and playfulness, for being the strongest woman I know, for giving me the strength and courage to follow my dreams, for everything...; my grandmother, Custódia, for her undeniable pride in her granddaughter, even though she has no idea what I’m doing, just computer stuff; my boyfriend, Gonçalo, for being my rock at times when I doubted everything, for us dreaming the same dreams; my friends, Ana, Rosa, Cátia e Inês, for their patience, support and postponed trips; my cats, whose cuddles and purrs are the best remedy for stress; all the others who came into my life and changed it for the better.

Thank you all, for everything!

Abstract

Concurrent programming has grown significantly in popularity thanks to the increased demands for availability and responsiveness placed on software systems to accommodate expanding end-user populations. Nonetheless, dealing with the inherent challenges of concurrent programming is considerably more complex than in traditional sequential programming. To gain a better grasp of concurrency, programming languages should incorporate novel functionalities that offer programmers high-level concurrency primitives.

Session types provide a typing discipline to ensure that channel communication follows specific protocols, enforced during type-checking, preventing communication errors. Additionally, session types based on linear logic, which are foundational to our work, ensure the absence of deadlocks. Session types aid in analyzing concurrency by representing such systems as a set of processes linked by channels, facilitating process coordination.

As a product of prior research, a compiler has been developed that compiles a functional language utilizing session types into Go. Our current study introduces an optimization layer to the compiler, which detects opportunities for optimization, enhancing the efficiency of the code generated, focusing on enhancing synchronization properties and improving message exchange. We have implemented three optimizations: the *forwarding optimization* simplifies the redirection of messages between channels, reducing synchronization points and the creation of channels; *recursion optimization* converts recursive functions into loops to minimize the overhead of recursive function calls; *packing messages optimization* combines multiple messages into a single message to decrease the number of messages exchanged and minimize synchronization costs.

After implementing the optimizations and comparing them to the non-optimized version, we conducted a quantitative evaluation to determine performance gains. We also conducted an informal and experimental validation, arguing that the optimizing transformations preserve program semantics.

We have concluded that the optimizations incorporated in this research enhance the performance of the generated code, while also maintaining program semantics and not hindering compilation times. The most notable optimization is the recursion optimization, which can generate code up to 35 times faster than the non-optimized version.

Keywords: Concurrency, Session Types, Compilation, Go, Functional Language, Compiler optimizations

Resumo

A popularidade da programação concorrente tem aumentado significativamente graças às exigências crescentes de disponibilidade e capacidade de resposta impostas aos sistemas de *software* para acomodar populações de utilizadores finais em expansão. No entanto, lidar com os desafios inerentes à programação concorrente é consideravelmente mais complexo do que na programação sequencial tradicional. Para melhor compreender a concorrência, as linguagens de programação devem incorporar novas funcionalidades que ofereçam aos programadores primitivas de alto nível.

Os tipos de sessão fornecem uma disciplina de tipos para garantir que a comunicação entre canais respeita o protocolo especificado, averiguando essa conformidade durante a fase de verificação de tipos. Estes representam sistemas concorrentes como um conjunto de processos ligados por canais, facilitando a análise da coordenação de processos. Além disso, os tipos de sessão baseados na lógica linear garantem a ausência de *deadlocks*.

Como produto de trabalhos anteriores, foi desenvolvido um compilador que compila para Go uma linguagem funcional que utiliza tipos de sessão. Este trabalho introduz uma camada de otimização no compilador com foco na das propriedades de sincronização e da troca de mensagens. Foram implementadas três otimizações: a *otimização do forwarding* simplifica o redirecionamento de mensagens entre canais, reduzindo os pontos de sincronização e canais criados; a *otimização da recursão* converte funções recursivas em ciclos para minimizar os custos de chamadas de funções recursivas; *otimização do empacotamento de mensagens* combina várias mensagens numa única para diminuir o número de mensagens trocadas e os custos de sincronização.

Depois de implementar as otimizações e compará-las com a versão não otimizada, realizámos uma avaliação quantitativa para determinar os ganhos de desempenho. Também realizámos uma validação informal e experimental, argumentando que as transformações de otimização preservam a semântica do programa.

Concluímos que as otimizações incorporadas nesta investigação melhoram o desempenho do código gerado, mantendo a semântica do programa e não prejudicando os tempos de compilação. A otimização mais notável é a otimização de recursão, capaz de gerar código até 35 vezes mais rápido do que a versão não otimizada.

Palavras-chave: Concurrência, Tipos de sessão, Compilação, Go, Linguagens funcionais, Otimizações de compiladores

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ACP	Algebra of Communicating Processes *(p. 5)*
AST	abstract syntax tree *(pp. 27, 29, 32, 33, 36, 39, 41, 55)*
CC0	Concurrent C0 *(pp. 10–12)*
CCS	Calculus of Communicating System *(p. 5)*
CSP	Communicating Sequential Processes *(pp. 5, 6)*
DAG	directed acyclic graph *(pp. 15, 16)*
SILL	Sessions from Intuitionistic Linear Logic *(pp. 2, 10, 13, 22, 31)*

1 Introduction

Concurrent programming involves running multiple instruction sequences simultaneously. Its evolution has been driven by the needs of modern computing to support the increasing availability and responsiveness demands placed on software systems by an ever-increasing number of end users. When executed properly, it is a powerful tool for harnessing the computational power of modern multi-core CPUs. However, concurrent programming presents inherent challenges that are significantly more arduous than those of traditional sequential programming.

The main challenges of concurrent programming are centered on the shared resources between multiple processes. Safely sharing resources is difficult, and can result in issues like deadlocks, livelocks, and starvation. In addition, concurrent programs are harder to debug and reason about since problems can arise in a non-deterministic manner.

Despite the fast-paced expansion of concurrent programming, most programming languages were not originally developed with concurrency as a fundamental concern. Thus, it can be challenging to tackle the intricacies linked with concurrent programming. Therefore, it is essential to study methods to improve concurrent programming and to develop mechanisms to better handle concurrency. Simple and straightforward paradigms help reduce complexity when reasoning about concurrent programs and avoid common pitfalls.

Shared memory and message-passing are two frequently used paradigms for concurrent programming [2]. In a *shared memory* model processes communicate by reading and writing data to a shared memory location, using mechanisms such as semaphores, monitors, and locks. In a *message-passing* model processes communicate by sending and receiving messages, via some asynchronous or synchronous channel-like abstraction.

The message-passing primitives, being high-level, are easier to implement and less error-prone compared to lower-level mechanisms, like locks, that tend to increase the complexity of programs. Message-passing concurrency is supported by numerous modern programming languages, such as Go, Rust, and Haskell. Nonetheless, these languages offer limited correctness guarantees concerning the use of such concurrency primitives.

Furthermore, the channels in these languages are typed and can only carry fixed payloads but most programs follow a protocol in which different types of values are exchanged during their communication.

To address the lack of mechanisms to specify a communication protocol, *session types* were introduced [3]. Session types offer a typing discipline that guarantees communication through channels adheres to a specific protocol by statically checking it via type-checking. Employing session types enables the prevention of communication errors, and session types associated with linear logic, such as those that form the basis of our work, further guarantee the absence of deadlocks. Therefore, session types aid in reasoning about concurrent programs by modeling such systems as collections of processes connected via channels and also assist in analyzing the coordination of these processes.

Despite the potential and benefits of session types, there is still significant research required to integrate them into general-purpose programming languages. Integrating session types into general-purpose languages presents a challenge because it requires upholding the notion of state-aware or linear typing in a non-linear typing system. The study of natively session-typed languages has resulted in the creation of various academic-flavored programming languages, with a significant emphasis on the algorithmic challenge of type-checking. Due to their academic nature, most of these languages are interpreted [4–8] rather than compiled [9, 10], and their runtime performance is generally neglected.

In previous work, Geraldo [11, 12] designed a general-purpose programming language implementing session types. This language was based on Sessions from Intuitionistic Linear Logic (SILL) [13], modeling concurrency with communication between processes in the style induced by intuitionistic linear logic. Geraldo has developed an interpreter, a bidirectional type-checker, and a compiler to Go code. The type-checker ensures the prevention of communication errors and deadlock freedom.

The compiler that was built was not developed with the performance of the generated code in mind, leading to generally inefficient code. Additionally, utilizing concurrency alone is insufficient to meet modern software's performance requirements. Given the prevalence of type-based optimizations in modern compilers, it is reasonable to assume that session-based concurrency type-driven optimizations can be achieved by leveraging session-type information.

Integrating session-based concurrency with type-driven optimizations can lead to safer and more efficient concurrent systems. So, with compiler optimizations focused on session types, our objective is to enhance the existing compiler by incorporating an optimization phase that identifies opportunities for optimization, resulting in more efficient generated code. The primary focus of the introduced optimization is to improve synchronization properties and optimize message exchange while also preserving the original programs' behavior.

The optimization phase focuses on three main optimizations:

- The **forwarding optimization** simplifies the redirection of messages between channels, reducing synchronization points and the creation of channels;
- **Recursion optimization** converts recursive functions into loops to minimize the overhead of function calls and recursion expenses;
- **Packing messages optimization** combines multiple messages into a single message to decrease the number of messages exchanged and minimize synchronization costs.

This work contributed to enhancing the performance of the code generated by Geraldo's compiler, all the optimizations had positive results and the compilation times were not severely impacted, resulting in advantageous optimizations.

1.1 Document outline

This document begins by introducing the pertinent background in Chapter 2, presenting the concepts of session types and fundamental notions of compilers and compiler optimizations. In Chapter 3 we present the the language that served as the foundation for this work, briefly explaining its syntax, semantics, and type system. The fundamental concepts of Geraldo's compiler are presented in Chapter 4. In Chapter 5, we present the optimizations that were implemented, explaining their purpose and implementation, and highlighting the distinctions between the original and the optimized code. The outcomes of the optimizations are displayed in Chapter 6, demonstrating the improvements in performance and the impact on compilation times, and discussing the results. Finally, Chapter 7 presents our conclusions and discusses potential courses of further research.

1.2 Contributions

The compiled code is accessible for review on a GitHub repository [14]. The main branch of this repository contains the optimizing compiler, in the `sessint` folder, and all the examples used in this work, and respective results, in the `results` folder. The folder `results_scale` contains the chosen examples to scale up and the corresponding results. The compiler includes the `Optimizations` module for the added optimization phase and the `BundleMessages` module for stages and methods related to message packing optimization. The `Compiler` module, which is responsible for code generation, has also undergone additions and alterations. The original compiler is located on the `Original_Compiler` branch.

2 Background and Related Work

Concurrent programming is a programming paradigm for handling the execution of multiple processes simultaneously, these processes cooperate to perform a task [2]. Its goal is to design programs that can efficiently and effectively handle multiple tasks simultaneously, taking advantage of modern multi-core processors and distributed computing environments.

Compilers enhance the performance of concurrent programs by identifying optimization opportunities and exploring the advantages of concurrent programming. When dealing with concurrent programming, compiler optimizations must consider the interactions between threads and how their execution may affect one another. Certain optimizations may need to be restrained or avoided to ensure proper behavior in multi-threaded environments.

This chapter addresses the key challenges of concurrent programming and explains how *session types* can be leveraged to overcome them. Additionally, it provides an introduction to the fundamental concepts of *compilers* and *compiler optimizations* before presenting our research on optimizing a compiler for a session types-based programming language in subsequent chapters.

2.1 Concurrent Programming

With the increasing demand for more powerful and efficient software, concurrent programming has become increasingly relevant in recent years. Although concurrent programming has its benefits, it also poses new challenges. Reasoning about concurrent programs is more difficult than reasoning about sequential ones. Developers must ensure that no execution paths leading to incorrect program states are executed. In addition, they must tackle complexities such as race conditions, deadlocks, and thread safety issues.

The main challenge in concurrent programming lies in coordinating the execution of multiple processes, which may all compete for the same resources, in order to ensure proper program execution. The programs must possess certain properties, such as *safety*, which ensure that a program will never enter a flawed state, and *liveness*, which ensures that a program will eventually enter a desirable state [2]. Demonstrating these properties through an exhaustive case analysis is infeasible due to the possible infinite execution states in concurrent programs. Debugging does not guarantee the absence of errors, and debugging concurrent programs presents its own challenges because of the difficulty of reproducing interactions between different processes and the non-deterministic behavior of concurrent programs.

The complexity of concurrent programming makes it difficult to reason about, leading to the research of new paradigms and programming languages to make it easier. Concurrency theory attempts to solve this by reasoning about concurrent computing using abstract models of concurrency, such as process algebras, allowing for a detailed study of the behavior and interactions of complex concurrent systems.

Process algebras are mathematically rigorous languages that allow specifying and confirming characteristics of concurrent systems [15]. They often use the abstraction of a communication *channel* to model the interaction between different processes. Methods based on process algebras have proven particularly effective in establishing the properties of concurrent systems and providing formal semantics for them. Process algebras have been the subject of extensive research over the past 30 years, beginning with the introduction of Calculus of Communicating System (CCS) [16], Communicating Sequential Processes (CSP) [17] and Algebra of Communicating Processes (ACP) [18]. Built with CCS as its foundation and taking advantage of many key ideas from the λ-calculus, the π-calculus [19] is now a widely used process algebra for reasoning about processes. π–calculus has been extended in several ways to model communication between processes. For example, linear types can be used to control the usage of a resource by limiting the number of times it can be used [20, 21].

Although linear types and other primitives aid in managing communication between multiple processes and provide useful representation for single interactions, they lack the ability to represent and structure a series of reciprocal interactions between two processes. These types are unable to capture a sequence of operations [21]. In order to represent a sequence of communications from a protocol, we must define a set of distinct and unrelated interactions. This increases the burden of programming tasks and the chance of errors. In systems that rely on sophisticated communication patterns between concurrent processes, which are increasingly common today, a lack of organizing strategies can lead to poor readability and bugs.

It is essential to provide a fundamental way to clearly and systematically describe complex interaction behaviors at a high degree of abstraction, along with a formal basis for verification. To achieve this goal, researchers proposed *session types* [3]. These types are utilized to ensure protocol conformance and provide assurances that messages sent and received between concurrent processes adhere to the expected order and type.

2.1.1 Session Types

Session types' main focus is to help structure the communication between processes and to provide a way to characterize the behavior in channel-based models. The fundamental ideas behind session types are [3]:

- The concept of a *session*, which represents a series of interactions between processes. A session allows branching and recursion and acts as a unit of abstraction to describe an interaction. A session is identified by a channel through which the session's interactions are carried out, a fresh channel is generated when initiating each session;
- Three basic primitives for communication: value passing, label branching, and delegation:
  - *Value passing* is the standard message passing found in CSP and π-calculus;
  - *Label branching* is like a method invocation but without value passing;
  - *Delegation* consists in passing a channel to another process.

  With the session structure and the combination of these primitives, it is possible to characterize a complex communication flexibly and clearly;
- A basic type discipline to ensure that two communication processes always interact with compatible communication patterns.

Session types allow type-level coordination between processes, conceptualizing concurrency as a set of processes linked by channels. Static characterization of the temporal behavior of π-calculus is enabled, while some of its risks, such as non-determinism of shared channels and improper data transmission, are eliminated. The type system provides these guarantees, while the language maintains a high level of expressiveness, featuring patterns such as call-return, method invocation, continuous interactions, unbounded interactions, and delegation [3].

In Figure 2.1 we present the syntax proposed by Honda *et al.* [3] for session types.

The construct `request` $a(k)$ `in` P initiates a session over a name a with a fresh channel k, this channel would be used in the process P for communication. The construct `accept` $a(k)$ `in` P is the complementary action to the `request` construct: receives a request to initiate a session via a, generates a fresh channel k that would be used in process P.

As previously mentioned, three interaction types are anticipated: value sending, branching, and passing channels. This is accomplished by the constructs for data sending/receiving, label selection/branching, and channel sending/receiving, respectively.

For *data sending* and *receiving* we use the constructs $k![\tilde{e}]; P$ and $k?[\tilde{x}; P]$, respectively, that allow for standard synchronous message passing, sending the values of the expressions $\tilde{e}$ through the channel k in the first construct and receiving the values $\tilde{x}$ through the channel k in the second.

P	$::=$	$\texttt{request}\ a(k)\ \texttt{in}\ P$	session request
	$\mid$	$\texttt{accept}\ a(k)\ \texttt{in}\ P$	session acceptance
	$\mid$	$k![\tilde{e}];P$	data sending
	$\mid$	$k?[\tilde{x}]\ \texttt{in}\ P$	data reception
	$\mid$	$k \lhd l; P$	label selection
	$\mid$	$k \rhd \{l_1 : P_1 \mid \dots \mid l_n : P_n\}$	label branching
	$\mid$	$\texttt{throw}\ k[k']\ \texttt{in}\ P$	channel sending
	$\mid$	$\texttt{catch}\ k(k')\ \texttt{in}\ P$	channel reception
	$\mid$	$\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$	conditional branch
	$\mid$	$P \mid Q$	parallel composition
	$\mid$	$\texttt{inact}$	inaction
	$\mid$	$(\nu u)P$	name/channel hiding
	$\mid$	$\texttt{def}\ D\ \texttt{in}\ P$	recursion
	$\mid$	$X[\tilde{e}\tilde{k}]$	process variables

Figure 2.1: Session types syntax of processes

For *branching*, we use the constructs $k \rhd \{l_1 : P_1 | \dots | l_n : P_n\}$ and $k \lhd l; P$. The first construct branches on the label l that is received through the channel k and continues as P_i if l_i is selected. The second selects a label l from the set of labels of the session and continues as P.

For *channel passing*, we pass a channel from one session to another process. For this, we use the constructs $\texttt{throw}\ k[k']\ \texttt{in}\ P$ and $\texttt{catch}\ k(k')\ \texttt{in}\ P$, which allow for the sending and receiving of a channel k' through the channel k, respectively.

Following these constructs associated with session types, we have more standard constructs in concurrent programming: concurrent composition, name/channel hiding, conditional, and recursion. The construct $P \mid Q$ executes P and Q in parallel. The construct $(\nu u)P$ hides the name u in the process P. The construct $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$ branches on the boolean expression e and continues as P if e is true and Q otherwise. The construct $\texttt{def}\ D\ \texttt{in}\ P$ defines a recursive process D in the process P.

The type syntax [3] is presented in Figure 2.2, where $(t, t', \dots)$ are type variables, $(s, s', \dots)$ sort variables, $(S, S', \dots)$ represent sorts (basic data types) and $(\alpha, \beta, \dots)$ are types of interactions over channels.

$$
\begin{array}{lllllllllll}
S & ::= & \mathbf{nat} & \mid & \mathbf{bool} & \mid & \langle \alpha, \bar{\alpha} \rangle & \mid & s & \mid & \mu s.S \\
\\
\alpha & ::= & \downarrow [\tilde{S}]; \alpha & \mid & \downarrow [\alpha]; \beta & \mid & \&\{l_1 : \alpha_1, \dots, l_n : \alpha_n\} & \mid & \mathbf{1} & \mid & \bot \\
& \mid & \uparrow [\tilde{S}]; \alpha & \mid & \uparrow [\alpha]; \beta & \mid & \oplus\{l_1 : \alpha_1, \dots, l_n : \alpha_n\} & \mid & t & \mid & \mu t.\alpha
\end{array}
$$

Figure 2.2: Session types syntax of types

We emphasize the sort $\langle \alpha, \bar{\alpha} \rangle$ that represents two complementary interaction structures, associated with a name, one starts with `accept` and the other with `request`. The type $\downarrow [\tilde{S}]; \alpha$ correspond to inputting the values of sorts $\tilde{S}$ and continue with the actions of type α, $\uparrow [\tilde{S}]; \alpha$ is its complementary type, receiving values instead of sending them. The constructs $\downarrow [\alpha];\beta$ and $\uparrow [\alpha];\beta$ are similar but instead of values, represent the send and receive of channels. The type $\&\{l_1 : \alpha_1, \dots, l_n : \alpha_n\}$ expresses the branching of the type α_i if the label l_i is selected. The type $\oplus\{l_1 : \alpha_1, \dots, l_n : \alpha_n\}$ is its complementary type, selecting a label l_i and continuing with the type α_i. The type $\mu t.\alpha$ is a recursive type, where the type variable t is bound in the type α. The type $\mathbf{1}$ represents the inaction of a session, and $\perp$ indicates that no further connection is possible.

The type system of session types follows the judgment $\Theta; \Gamma \vdash P \rhd \Delta$, stating that under the environment Θ and Γ the process P has type Δ. If $\Theta; \Gamma \vdash P \rhd \Delta$ is derivable in the system then P is typable under $\Theta; \Gamma$ with Δ. From the type system rules we highlight the rule for sending and receiving values, presented in Figure 2.3 and Figure 2.4, respectively [3].

$$\frac{\Gamma \vdash \tilde{e} \rhd \tilde{S} \qquad \Theta; \Gamma \vdash P \rhd \Delta \cdot k : \alpha}{\Theta; \Gamma \vdash k![\tilde{e}]; P \rhd \Delta \cdot k : \uparrow [\tilde{S}]; \alpha} \ \textsc{Send}$$

Figure 2.3: Typing rule for input

In the typing rule for sending if, in the environment context Θ, the typing context Γ can derive that the expressions $\tilde{e}$ have session types $\tilde{S}$, and if the continuation process P has session type α in the environment context Δ, then the entire expression $k![\tilde{e}]; P$ is well-typed with the session type $\uparrow [\tilde{S}]; \alpha$. In simpler terms, the rule ensures that before sending a message over a channel, the message's values and the continuation process are properly typed according to the specified session types.

$$\frac{\Theta; \Gamma \cdot \tilde{x} : \tilde{S} \vdash P \rhd \Delta \cdot k : \alpha}{\Theta; \Gamma \vdash k?[\tilde{x}] \ \texttt{in} \ P \rhd \Delta \cdot k : \downarrow [\tilde{S}]; \alpha} \ \textsc{Rcv}$$

Figure 2.4: Typing rule for output

In the typing rule for receiving values, if in the environment context Θ, the typing context Γ can derive that the variables $\tilde{x}$ have session types $\tilde{S}$, and if the continuation process P has session type α in the environment context Δ, then the entire expression $k?[\tilde{x}]$ `in` P is well-typed with the session type $\downarrow [\tilde{S}]; \alpha$. In simpler terms, the rule ensures that before receiving a message over a channel, the variables used to store the received values are properly typed according to the specified session types.

Session types provide a means to express our intentions in clear terms for cross-channel communication. Furthermore, it provides us with the ability to assess compliance with a communication protocol through type-checking [3]. To ensure compliance with a protocol, session types rely on the complementary or dual communication patterns between the caller and the callee. Consequently, the type-checker can dismiss non-complementary patterns [3], posing a significant advantage that facilitates the development of correct programs. The duality of types was presented before and is presented in Figure 2.5.

$$\begin{array}{rcl}
\overline{\uparrow [\tilde{S}]; \alpha} & = & \downarrow [\tilde{S}]; \overline{\alpha} \\
\overline{\downarrow [\tilde{S}]; \alpha} & = & \uparrow [\tilde{S}]; \overline{\alpha} \\
\overline{\&\{l_i : \alpha_i\}} & = & \oplus\{l_i : \overline{\alpha_i}\} \\
\overline{\oplus\{l_i : \alpha_i\}} & = & \&\{l_i : \overline{\alpha_i}\} \\
\overline{\uparrow [\alpha]; \beta} & = & \downarrow [\alpha]; \overline{\beta} \\
\overline{\downarrow [\alpha]; \beta} & = & \uparrow [\alpha]; \overline{\beta} \\
\overline{\mu t.\alpha} & = & \mu t.\overline{\alpha} \\
\overline{\mathbf{1}} & = & \mathbf{1} \\
\overline{t} & = & t
\end{array}$$

Figure 2.5: Duality of session types

One benefit of session types is their ability to define and specify complex interactions in a structured, simple, and clear manner as a unit of abstraction, rather than being limited to a pre-defined set of basic primitives. Session types are a powerful tool for structuring programs that involve message-passing and concurrent processes. They can also be utilized to convey common communication patterns, which Example 2.1 [3] illustrates.

Example 2.1 (ATM — a session type example with continuous interactions)**.** The traditional call-return can encapsulate a sequence of communications as a single abstract unit. We can extend this idea to more complex communication patterns. For example, the following session type describes the interaction between a customer and an ATM:

$$\begin{array}{rl}
ATM(a,b) = & \texttt{accept}\ a(k)\ \texttt{in}\ k![id]; \\
& \quad k \rhd \{\texttt{deposit} : \texttt{request}\ b(h)\ \texttt{in}\ k?(amt)\ \texttt{in} \\
& \qquad h \lhd \texttt{deposit}; h![id, amt]; ATM[a,b] \\
& \quad \mid \texttt{whitdraw} : \texttt{request}\ b(h)\ \texttt{in}\ k?(amt)\ \texttt{in} \\
& \qquad h \lhd \texttt{withdraw}; h![id, amt]; \\
& \qquad k \rhd \{\texttt{success} : k \lhd \texttt{dispense}; k![amt]; ATM[a,b] \\
& \qquad\quad \mid \texttt{failure} : k \lhd \texttt{overdraft}; ATM[a,b]\}\} \\
& \quad \mid \texttt{balance} : \texttt{request}\ b(h)\texttt{in}\ h \lhd \texttt{balance}; h?(amt)\ \texttt{amt in} \\
& \qquad k![amt]; ATM[a,b]\}
\end{array}$$

The program begins by accepting session a with the user and presenting three options: `deposit`, `withdraw`, and `balance`. If the user chooses to `withdraw` funds, the program prompts them to enter the amount *amt*. The program then interacts with the bank via session b, using the user's code and the specified amount. The withdrawal operation may either succeed or fail, and the program responds accordingly with two options. If the operation is successful, the bank will respond with `success` and pay out the amount *amt* to the user. If the operation is unsuccessful, the bank will provide a message of failure and result in an `overdraft` message. Regardless of the case, the program proceeds to the initial waiting mode.

A user for this program may be specified in the following way:

$$\texttt{request}\ a(k)\ \texttt{in}\ k![myId]; k \lhd \texttt{withdraw}; k![58];$$
$$k \rhd \{\texttt{dispense} : k?[amt]\ \texttt{in}\ P\ |\texttt{overdraft} : Q\}.$$

We highlight that the interactions are reciprocal. This program demonstrates interactions among three parties: the user, ATM, and the bank. △

Session types' systems are aimed at achieving session fidelity. While it cannot guarantee the absence of deadlocks, a running process will behave as expected, in accordance with the specified protocol. Furthermore, this typing system guarantees the absence of runtime errors, that is, a typable program will never reduce to an error [3].

2.1.2 Languages featuring session types

Investigations aimed at achieving better and safer forms of communication have long focused on the design of concurrent programming languages. These investigations led to the creation of session types, which provide a better approach to structuring and reasoning about complex interactions. The SILL [13, 22, 23] family of languages is a culmination of this work, demonstrating the compatibility of a variety of practical language features with a logical foundation. This brings us closer to the development of new languages with better tools for concurrent program development.

In addition to standard functional programming features, the SILL language family is designed to enable concurrent communication between processes through the use of channels [13]. The language uses the perspective of propositions-as-types interpretation of intuitionistic linear logic as a session-typed language, and it aims to enable desirable properties such as deadlock prevention, session fidelity, and progress, while maintaining practical usability.

The SILL family is a complete functional calculus that uses session types to model concurrency and is designed to support higher-order, message-passing concurrent computation [13]. Since our work is based on SILL family, we will discuss the language in more detail in Chapter 3.

Session types have undergone extensive research to model concurrency, leading to numerous minimalist programming language implementations. Essentially academic, these languages aim to improve type-checker implementations and are mostly interpreted, as shown by the work of Pfenning and Griffith [22], and Rocha and Caires [24, 25]. Nevertheless, there are also some compiled languages that include FreeST [9], Concurrent C0 (CC0) [10], and our work. Since our work is based on an existing compiler, to better understand different approaches to the compilation of languages with session types we highlight two compiled languages that use session types: FreeST [9] and CC0 [10].

Concurrent C0 CC0 [10] is derived from C0, an imperative language that bears resemblances to C. It offers communication via channels using session types, providing tools for safe and effective concurrent programming and eliminating the burden of manually synchronizing bidirectional communication. Communication is asynchronous, rendering all sends as non-blocking procedures, but the receiving end has to block until a message is available.

CC0's typing system ensures safe concurrency via linearity and session fidelity enforcement. Session types enable CC0's implementation to use fewer and smaller buffers compared to alternative message-passing strategies. Although important, the type-checker in CC0 is not a novelty since Griffith [26] has also worked on type-checking session-typed and linear languages. After type-checking, the CC0 compiler inserts annotations with information regarding the communication structure to the runtime. Then it compiles the source to a target language (C or Go), linking it with a runtime implementation written in the same target language.

The CC0 compiler infers widths, which represent a concept associated with specific session types to determine the number of values that can be buffered at a time. As a result, this feature empowers the CC0 runtime to implement small, fixed-length circular buffers as queues without any need for resizing.

The CC0 implementation includes a feature called *forwarding*, which enables a process to terminate before its child and remove itself from the process tree. A forward operation contracts a node with exactly one child, establishing direct communication between its parent and its child. Session fidelity is preserved through forwarding since a process can only forward channels of the same session type. In this implementation, forwarding is regarded as a "special kind of message". The typing system infers communication directions based on the forwarding process. Therefore, a forward message includes a reference to the new channel and tries to obtain the value it originally anticipated through that new channel. When a forward is received, the receiving process destroys the original channel and replaces its reference with the new one.

The compiler generates either C or Go code, resulting in different runtimes with different threading models and synchronization strategies. However, the general structure of channels is shared among both runtimes. A channel consists of a message queue, a channel direction, a mutex that safeguards the channel's state, and a condition variable used by receivers to wait for message arrival or a change in the queue's direction.

The runtime has four main functions to provide the functionality for spawning processes and message passing:

- `NewChannel` creates a new concurrent process and the channel that it will use, then returns a reference to that channel to the caller. It receives the function and arguments of the provider, and the information inferred by the compiler regarding the type width and initial direction of the channel;

- `Send` sends a given message over a given channel, also receiving the message type and inferred direction. It locks the channel to enqueue the message and its type and to set the direction of the channel, after which it unlocks the channel;

- `Recv` receives a message from the given channel and also takes the message type and inferred direction. It locks the channel to retrieve the message and if the buffer is empty it will unlock and wait for the condition variable for the sender.

- `Forward` takes the two channels involved and the inferred directions of the communications. The forwarding process sends a forward message in the inferred direction using regular message passing and then terminates.

Highly concurrent programming is encouraged by CC0, and most programs use several processes which makes C runtimes not perform as well as Go runtimes, since C uses heavier system threads while Go uses green threads, more lightweight.

The CC0 programming language is similar to C and lacks features such as polymorphism and higher-order functions. Additionally, due to its imperative nature, it differs from typical session-type implementations, which are typically functional. The runtime of CC0 differs significantly from our compiler, which we will introduce in Chapter 4.

FreeST FreeST [9] is a typed programming language for concurrent computing. It employs message-passing to enable communication between processes, using bidirectional channels to exchange messages.

The FreeST type system manages communication over channels using polymorphic context-free session types. Context-free session types allow for nested protocols that are not limited to tail recursion and extend regular session types [27]. The language has primitives for constructing and interacting on channels and forking new threads, both of which are based on a core linear functional programming language. The compiler relies on a novel algorithm for determining the equivalents of context-free session types, based on previous work by Vasconcelos *et al.* [28].

FreeST is developed in Haskell and generates Haskell code. It uses the `MVar t` type provided by Haskell for implementing communication channels. A `MVar t` refers to a mutable location that can either be empty or have a value of type `t`. A channel is implemented using two `MVars`. Writing to a channel requires the second `MVar` and the `putMVar` primitive. If the `MVar` is full, it waits until it empties. Reading from a channel uses the first `MVar` and the `takeMVar` primitive, which returns the contents of `MVar` or waits if the variable is empty.

Session types imply that the types passed through a channel may change over time, so the implementation of FreeST uses `unsafeCoerce` to bypass the Haskell type system. This makes it possible to avoid type errors when compiling the target Haskell code. The use of `unsafeCoerce`, in this case, is safe since the compiler does type-checking prior to its use. The use of `unsafeCoerce` is limited to the reading and writing from `MVars`.

The call-by-value semantics used by context-free session types conflict with Haskell's lazy evaluation. To overcome this problem, FreeST implements the desired call-by-value semantics by using *bang patterns*, which force parameter evaluation before executing the function body.

When compared to FreeST, SILL exhibits stronger properties such as progress and deadlock freedom. One possible limitation of FreeST is that the threads provided by Haskell are operating system threads, which are more resource-intensive compared to lightweight threads such as those used by Go, used in our work, as will be presented in Chapter 4. This issue is further compounded by the fact that programs using session types are typically concurrent and tend to launch multiple threads, exacerbating the problem.

2.2 Compilers

A compiler is a program that maps a source program into a semantically equivalent target program, from a high-level language to a lower-level language. This process comprises two steps: analysis and synthesis [29].

During the *analysis stage* [29], the source program is broken down into its individual elements, and a grammatical structure is applied to create an intermediate representation of the program. When the analysis detects syntax errors or semantic flaws in the source program, it must provide informative warnings. The data collected in the analysis phase is sent, along with the intermediate representations, to the synthesis phase.

The analysis stage has several sub-stages. Initially, the compiler conducts lexical analysis on the character stream of the source program to generate a series of tokens. The resulting tokens are fed to the next stage, the syntax analysis, which creates a syntax tree as an intermediate representation, where each node represents an operation and its children represent the arguments. Finally, the compiler performs semantic analysis, wherein type-checking plays a vital role.

Type-checking is a technique used to prevent type inconsistencies in programs. It operates based on the rules defined by the type system to analyze program behavior and ensures that the types of operands match the expected types.

Type-checking can be performed in two ways: type synthesis, which derives the type of an expression from the types of its sub-expressions and requires type annotations, and type inference, which determines the type of an expression based on the way it is used and does not rely on type annotations. Bidirectional type-checking is a technique that combines type synthesis and type inference to achieve a more efficient type-checking process, and it will be discussed in Subsection 2.2.1.

Following the analysis stage, we proceed to the synthesis stage. The *synthesis stage* [29], builds the intended target program using the intermediate representation and data from the analysis stage. To produce a better target program, most compilers perform transformations on the intermediate representation during an *optimizing phase*, which comes after the analysis stage and before the synthesis stage.

2.2.1 Bidirectional type-checking

Type-checking ensures that the operands' types match the expected types for an operator by using logical principles to analyze a program's behavior [29]. The type-checker allows the programmer to confirm that a developed program is compatible with the intended behavior, and it also allows the compiler to reject faulty programs.

In a pure type inference system, there's no need to declare types, because the compiler will infer this information. However, type inference becomes undecidable when dealing with more advanced type systems, such as those that statically check array accesses or invariants of data structures. Furthermore, in the absence of type annotations, error messages often fail to accurately identify the true cause of the error [30]. Despite these limitations, type inference has its advantages. For instance, in a system without type inference, everything has to be annotated, as is the case in languages like Java. Such extensive annotation can quickly become cumbersome and burdensome.

Bidirectional type-checking allows programmers to utilize advanced typing features while maintaining a reasonable number of type annotations, which are only required for certain expressions. Annotations must be lightweight and non-obvious, constituting only a fraction of the program's text [30, 31]. Functions and high-level methods must be annotated, but not their intermediate results.

In bidirectional type systems, a type alternates between being treated as an input and an output. The algorithm alternates between synthesizing the type and checking it against the types already known, instead of trying to infer the type of all expressions [30, 31].

Bidirectional typing replaces the standard typing judgment $\Gamma \vdash e : \tau$, which is read "under assumptions in the Γ, the expression e synthesizes type τ", with two different judgments [30]:

- $\Gamma \vdash e \Rightarrow \tau$ that reads "under assumptions in Γ, the expression e synthesizes type τ". This means that it is only known Γ and e and it is necessary to figure out the type τ from e, as in type inference.

- $\Gamma \vdash e \Leftarrow \tau$ that reads "under assumptions in Γ, the expression e checks against type τ". In this situation, the type τ is already known and it's necessary to make sure e conforms to that type (check against).

To implement this technique, it is necessary to determine which premises or conclusions synthesize a type and which ones check against a known type. One approach is to cover all possibilities with rules, but this results in an excessive number of rules. To avoid having to guess types, wherever we synthesize a type, the type should come from know information [31].

The "recipe" for bidirectional type-checking, begins by identifying the principal judgment in the rule. Next, we bidirectionalize this judgment by synthesizing if it is an elimination rule or by checking if it is an introduction rule. An introduction rule produces a term with a specific type, while an elimination rule explains how to use a term of a given type [31]. The third step involves bidirectionalizing the remaining judgments in the rule by leveraging available information: if the type of a judgment is known, it should be checked [31].

Bidirectional typing, therefore, allows programmers to benefit from both a type system with more advanced and sophisticated features as well as an acceptable amount of annotation providing greater flexibility to the type system. Bidirectional typing reduces the type annotation burden by inwardly propagating the type information. Several research projects [32–35] have incorporated type systems that take advantage of the flexibility provided by bidirectional type-checking.

2.2.2 Code generation

Code generation is the final stage of the compiling process. The compiler will take as input the intermediate representation created in the analysis phase and will map this representation to the target language [29]. This mapping must preserve the original semantics, must efficiently use the available resources of the machine on which the program will run, and the generated code must be correct with respect to the original.

Code generation is heuristics-based and may not always generate optimal code, since the generation of optimal code is an undecidable problem, and many of its subproblems, such as evaluation order, are NP-complete. When the efficiency of the generated code is crucial, an optimization phase is added before code generation. Subsection 2.2.3 will discuss compiler optimization topics.

The code generation stage has an *instruction selection* phase, where instructions from the intermediate representation are translated into a code sequence with instructions for the target machine. When conducting this phase, it is crucial to consider instruction speed and machine idioms. Although a naive instruction translation may lead to correct code, it can produce inefficient target code if efficiency is not considered. Knowing the cost of instructions is critical and it is possible to generate optimal code sequences considering these costs and taking advantage of dynamic programming [29].

Instruction ordering is another phase of code generation. It determines the execution order for instructions, which can impact the efficiency of the target code [29].

The majority of code generators divide the received intermediate representation into "basic blocks", which consist of a sequence of operations that can always be performed together. The building of a flow graph, representing the flow of control, is accomplished by using these basic blocks and further gains can be achieved by examining the local optimizations in each basic block [29].

The optimization of basic blocks is often achieved by building an directed acyclic graph (DAG), a node of the DAG is assigned to each of the initial values of the variables and statements in the basic block. The DAG representation allows for [29]:

- elimination of local common subexpressions, avoiding recomputation of values by finding nodes in the DAG that have the same children in the same order as others;

- elimination of dead code, identifying in the DAG any node with no ancestors;
- the reordering of independent statements;
- application of algebraic law to reorder operands and simplify the computation, like replacing a more expensive operator with a cheaper one (for example, replace x^2 by $x * x$) and replacing constant variables by their values.

Code generation is thus an important phase in the compilation process, and a well-thought-out strategy for its execution is needed to ensure that the generated code is correct and efficient.

2.2.3 Compiler optimizations

As previously stated, the naive translation of high-level language constructs into machine code often results in inefficient code. When generating code, the compiler aims to optimize it by eliminating unnecessary instructions or replacing them with more efficient alternatives in the target code. The optimization process can involve global optimizations between basic blocks or local optimizations within each block [29].

One additional goal of compiler optimizations is to increase the level of parallelism in tasks, thereby leveraging the multiple cores found in modern CPUs. Considering the aforementioned factors, it becomes clear that compiler optimizations are an essential component of the compilation process but also an arduous task. To guide the optimization process, we must consider the following four primary design objectives [29]:

- Correctness, in the way that must preserve the meaning of the source program;
- Performance gain in several programs, not just in very specific cases;
- Reasonable compilation time;
- Manageable effort in engineering.

In the following sections, we describe the most commonly implemented compiler optimizations.

Loop transformations There are several types of loop transformations and this type of optimization is the most common. The first strategy is *loop unrolling*, which unrolls the loop in successive computations. This reduces the number of executed instructions and in more instructions available for parallel execution [36].

The next examples illustrate the optimization using a simple syntax similar to pseudocode that utilizes the `DO` instruction to identify loops. The following example is presented to illustrate a *loop unrolling* optimization:

```
DO i = 1 TO 100 BY 1;
A(i) = A(i) + B(i);
END;
```

```
DO i = 1 TO 100 BY 2;
A(i) = A(i) + B(i);
A(i+1) = A(i+1) + B(i+1);
END;
```

On the left side, we can see an unoptimized version of a loop that starts at 1 and goes to 100, with increments by 1, performing the code `A(i) = A(i) + B(i);` at each iteration. On the right side, is its optimized version with a loop unfolding by 2. This optimization reduces the number of increments and tests for the loop control by half.

Another loop transformation technique is called *loop fusion*, which involves merging two separate loops into a single loop to reduce loop overhead and code space [36]. Below we present an example demonstrating an optimized version (on the right) compared to an unoptimized version (on the left):

```
DO i = 1 TO 100 BY 1;
A(i) = 0;
END;
DO i = 1 TO 100 BY 1;
B(i) = X(i) + Y;
END;
```

```
DO i = 1 TO 100 BY 1;
A(i) = 0;
B(i) = X(i) + Y;
END;
```

In order to perform loop fusion, certain conditions must be verified: both loops must be executed, computations in one loop must not affect the other, and both loops must execute the same number of times [36].

Redundant subexpression elimination Redundant subexpression elimination removes computations that are already available. As a result, executing fewer instructions and using less instruction space is possible [36].

The code below shows the unoptimized version on the left side. Notice that `C` and `D` have the same expression. This redundancy allows the optimization seen on the right side.

```
A = 10;
B = A;
C = A + 2 * 3;
D = 2 * 3 + B;
E = D + A;
```

```
A = 10;
B = A;
C = A + 2 * 3;
D = C;
E = D + A;
```

Code motion The majority of programs' execution time is spent executing instructions in inner loops. Reducing the number of these instructions can result in better execution times [29]. The code motion strategy can be used to attack this program by reducing the number of instructions in inner loops. By using the code motion strategy, a subexpression can be moved to a less frequently executed area of the program, such as outside the inner loop. However, the value of the subexpression should not be changed by the move [29, 36]. The goal of this optimization is to reduce the number of executed instructions.

The following example demonstrates an application of this technique: on the left side, in the non-optimized version, variable `A` is computed in each iteration of the loop, even though it's unnecessary since it does not depend on any of the results calculated in the loop. Therefore it can be moved outside the loop, resulting in the optimized version on the right side.

```
DO i = 1 TO 100 BY 1;
A = X * Y * W;
B = C(i) + A;
END;
```

```
A = X * Y * W;
DO i = 1 TO 100 BY 1;
B = C(i) + A;
END;
```

Constant folding Constant folding, which replaces variable usage of constants with the constants themselves, is another potential optimization.

In the following example, the expressions' values for `B` and `C`, found on the left side of the non-optimized version, can be computed at compile time since they depend solely on constants, resulting in the optimized version on the right side.

```
A = 1;
B = 2;
T = A + B;
IF (T = 3) F();
ELSE G();
```

```
A = 1;
B = 2;
T = 3;
IF (3 = 3) F();
ELSE G();
```

Dead code elimination Constant folding can result in dead code, which means code that cannot be executed because that area of the program is unreachable. These situations can be optimized by performing dead code elimination [36].

The presented example for constant folding illustrates the presence of dead code in which the `G()` function will never execute. Therefore, the following optimization can be performed:

```
A = 1;
B = 2;
T = 3;
F();
```

Parsing technique The performance of the generated code might be affected by the parsing technique. It is feasible to associate source expressions to create an instruction sequence that minimizes hardware delays by conducting a minimal depth parse [36]. For instance, the expression `A+B+C+D` should be computed as `(A+B)+(C+D)` instead of the left-to-right association `(((A+B)+C)+D)`. Thus, in Figure 2.6 its preferred the tree on the left rather than the one on the right.

This allows you to take advantage of pipelined units that can execute independent instructions simultaneously.

Peephole optimization Peephole optimization is a local technique that compilers use to optimize the final code. The technique involves scanning and examining a window of target instructions for potential transformations [29, 36]. This approach is not only restricted to

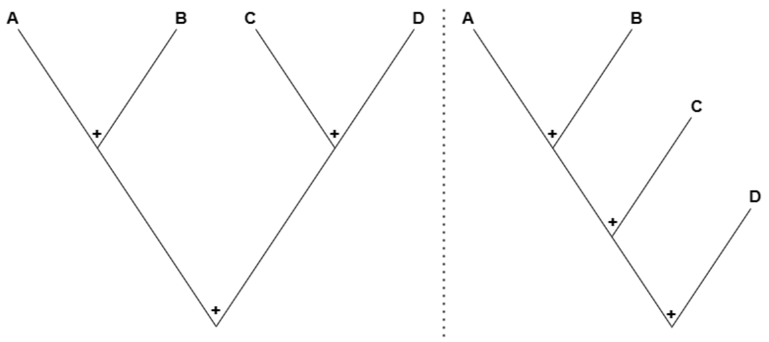

Figure 2.6: Two possible parsing trees of the expression

the final code, but it can also enhance the generated intermediate code, thus improving the resulting intermediate representation. Peephole optimization can be perceived as a small sliding window on a program, and optimal results may need several iterations of this technique, as one optimization can create opportunities for further improvements. The main peephole optimizations include [29]:

- *Redundant-instruction elimination*: naive code generation algorithms may produce redundant instructions, which can be eliminated to decrease the number of instructions executed and improve the efficiency of the generated code;
- *Flow-of-control optimizations*: often during code generation unnecessary jumps are produced (such as jumps to other jumps, jumps to conditionals, ...). These jumps can be simplified leading to simpler and better-performing code;
- *Algebraic simplifications* and reduction in strength can be made by replacing expensive operations with equivalent cheaper ones on the target machine;
- *Use of machine idioms* can take advantage of the fact that the target machine may have hardware instructions to implement specific operations more efficiently.

Tail-call optimizations Functional languages use functional calls to express loops and other control flows, which makes tail-call optimizations particularly important. A function f is considered to be in a tail position if it appears in the body of another function g and is the last operation that g executes before returning. During a tail-call optimization, instead of creating a new return address for f to return to, g can assign f the return address that was given to g, allowing f to directly return to it [37]. This optimization is particularly valuable when dealing with recursive calls.

The forwarding optimization and the recursion optimization of our compiler, which are described in Section 5.1 and Section 5.2, respectively, are heavily based on tail-call optimizations.

Optimizations for functional languages While all the mentioned optimizations can apply to both functional and imperative languages, functional languages have their specific optimizations.

Equational reasoning is a benefit of pure functional languages. It means that, since a purely functional language has no side effects, you can prove that one expression is equivalent to another (if $a = f(x)$ then $g(f(x), f(x))$ is equivalent to $g(a, a)$) [37].

In pure functional languages, variables are immutable since there are no variable assignments. Without variable assignments, developing programs requires creating new values instead of updating old ones. Immutable values allow a compiler to perform substitutions that are not possible otherwise, as shown in Example 2.2 [37].

Example 2.2. In this example, the compiler knows that the function call `f(r)` does not have side effects, meaning that it does not modify the variable `r`. So the compiler knows the variable `y` is computed using the values `a1` and `b1`.

```
type recrd = {a: ..., b: ...}
var a1 := 5
var b1 := 7
var r := recrd{a := a1, b := b1}
var x := f(r)
var y := r.a + r.b
```

△

Inline expansion is an optimization that is particularly significant in functional languages as these languages tend to use many small functions. This optimization involves copying the function body in place of a function call.

Deforestation [38] is a technique for transforming programs that aims to remove intermediate lists or tree structures that are created and immediately consumed by a program. This refers to the optimization of functional programs through the elimination of redundant data structures, resulting in the enhancement of both memory usage and system performance. Intermediate trees and lists are the foundations of specific functional programming styles. They come up naturally as a way to permit concise, modular, and declarative code, as shown in Example 2.3 [39].

Example 2.3. The following function accepts two integers denoting an interval and returns the sum of squared odd integers within that interval. This example assumes Haskell syntax. The function `f` is a composition of several operations that use the output of one function as intermediate value for subsequent operations.

```
f : (Integer, Integer) -> Integer
f = sum . map sq . filter odd . between
```

This writing style is founded on intermediate lists. It is possible to produce code that doesn't generate intermediate lists, but this would sacrifice the code's conciseness and modularity, making it longer and therefore more difficult to comprehend, as demonstrated below:

```
f(m, n) = go m
      where go m   m > n = 0
                   otherwise = go (m + 1) +
                         if odd m then sq m else 0
```

△

To maintain the modular and consistent style that is characteristic of functional languages, and to avoid the performance burden associated with intermediate representations, some compilers apply the deforestation technique by removing intermediate structures and converting multiple successive transformations into a single transformation.

The deforestation technique and peephole optimization served as primary inspirations for designing our compiler's packing messages optimization. This optimization is described in detail in Section 5.3.

Source-to-source compilation frequently allows for particular optimizations that leverage the mechanisms provided by the target language. As demonstrated in the constant folding example, applying one optimization can create new opportunities for improvement to emerge. The order in which optimizations are executed can create opportunities for new optimizations to arise, making it crucial to determine the optimal order for executing the optimizations.

3 Language

Our compiler takes a functional language that incorporates session types following the style of the SILL family of languages, heavily based on Toninho's [13] previous work. In this chapter, we introduce this language.

The SILL family of languages was devised to express concurrent communication between processes using channels alongside standard functional programming features [13]. It uses a perspective of propositions-as-types interpretation of intuitionistic linear logic as a session-typed language and it aims to achieve certain desirable qualities, such as deadlock freedom, session fidelity, and progress, while also ensuring practical usability.

The SILL family emerged from work [13] to build support for higher-order, massage-passing concurrent computation, building a complete functional calculus that takes advantage of session types to model concurrency. The core idea of SILL is to isolate linearity, instead of forcing everything to be linear. Forcing the whole language to be linear is troublesome since it hinders the integration with languages that ate not inherently linear.

Similar to SILL, our language is divided in two interdependent layers: the process layer and the functional layer. The *process layer* contains the concurrency primitives and can use the terms of the functional layer freely. The *functional layer* interacts with the process layer in a controlled manner, processes are embedded in the functional layer.

3.1 Syntax

The syntax of our language is presented in Figure 3.1. Here M, N are terms, and P, Q are process expressions of the language. The functional layer of the language comprises typical λ-abstraction, showed as $\texttt{fun}\ \bar{x} \rightarrow M\ \texttt{end}$, application as MN, recursion as $\texttt{let}\ x = M\ \texttt{in}\ N$, and constructs for sum, product, and recursive types. We emphasize the *contextual monad* construct $a \leftarrow \{P\} \leftarrow a_1, \ldots, a_n$ that expresses that a process P uses channels $a_1, \ldots, a_n$ to provide a service along channel a.

The constructs for the process layers express the behavior of sessions. The construct

M, N	$::=$	`fun` $\bar{x}\ \rightarrow M$ `end`	typical λ-abstraction
	$\mid$	`let` $x = M$ `in` N	recursion
	$\mid$	$M\,N$	function application
	$\mid$	`if` M `then` N_1 `else` N_2 `endif`	branching
	$\mid$	$\ldots$	basic types and their operations
	$\mid$	$a \leftarrow \{P_{a,\overline{a_i}}\} \leftarrow a_1, \ldots, a_n$	contextual monad
P, Q	$::=$	`send` $c\ M;\ P$	send a value M along channel c
	$\mid$	$x : T \leftarrow$ `recv` $c;\ P$	receives along channel c
	$\mid$	`send` $c\ d;\ P$	send a channel d along channel c
	$\mid$	`close` c	close channel c and terminate
	$\mid$	`wait` $c\ ;\ P$	wait for closure of c
	$\mid$	`fwd` $c_1\ c_2$	forward between c_1 and c_2
	$\mid$	$c \leftarrow$ `spawn` $M\ \overline{d};\ P$	spawn process M and bind channel c
	$\mid$	`case` c `of` $\overline{l_j : P_j}$	branch on selection of l_j along c
	$\mid$	$c.l_j\ ;\ P$	select label l_j along c
	$\mid$	`if` M `then` N_1 `else` N_2 `endif`	branching
	$\mid$	`print` $M\ ;\ P$	print

Figure 3.1: Syntax of expressions and processes in SILL

`send` $c\ M;\ P$ denotes that the functional term M is sent on channel c, and continues as P. The construct $x : T \leftarrow$ `recv` $c;\ P$ is its dual, it receives a value on channel c and binds it to x, of type T, continuing as P. The syntax for higher-order communication of channels is similar to the send and receive operations of functional terms.

To signal that there will be no further communication on channel c, we use the construct `close` c. When waiting for the closure of a session is needed, we use the wait construct `wait` $c;\ P$. The construct `fwd` $c_1\ c_2$ forwards the communication between channels c_1 and c_2, redirecting the inputs and outputs on c_1 to c_2 and vice-versa.

The construct for spawning of a process `spawn` $M\ \overline{d};\ P$ is complementary to the contextual monad construct of the functional layer. This construct indicates the launch of a new M process running in parallel with the process P, using the channels $\overline{d}$, and offering a fresh channel c. The monadic value can be used to refer to processes in the functional layer as well as to communicate with processes in the process layer.

Furthermore, the language has an idiom to offer a choice of alternative behaviors. This feature is implemented through the external choice construct `case` c `of` $\overline{l_j : P_j}$, which branches based on the selection of l_j along c and continues with P_j. Dually, the construct $c.l_j\ ;\ P$ selects the label l_j on the channel c, and then continues as P.

The process can branch itself by using the `if` M `then` N_1 `else` N_2 `endif` construct. If the term M evaluates to true, it continues as N_1, or as N_2 otherwise.

Lastly, the `print` $M\ ;\ P$ construct prints the value of the term M and proceeds as P.

Example 3.1 presents several features of our language, such as recursion, using a combination of the constructs `spawn` and `fwd`, and choice.

Example 3.1. This example demonstrates how to build a recursive function to calculate the next number in the Fibonacci sequence. The first line defines the `IntCStream` recursive type which gives the option to either request the `next` number in the sequence or end the communication, using the `stop` keyword.

In the second line, we define our function `fib`, which takes two arguments: `n`, the current number of the sequence, and `acc`, which is an accumulator, that represents the next number of the sequence. This function provides the service `IntCStream` through the channel `c`. Upon requesting the next number, using the `next` label, `n` is sent through channel `c`, on line 6, and a new process spawned, linked to channel `d` to compute the next number of the sequence, on line 7. The newly created channel `d` is connected to the offering channel `c` through the `fwd` construct, on line 8. By combining the `spawn` and `fwd` constructs, a recursive process is created. To terminate the communication the `stop` label is used, closing the channel `c`, on line 10.

```
 1  stype IntCStream rec x. &{next: int^x, stop: @};
 2  fib : int -> int -> {IntCStream}
 3  fun n acc -> c <- {
 4      case c of
 5          next: (
 6              send c n;
 7              d <- spawn (fib acc (n + acc));
 8              fwd d c
 9          ) stop: (
10              close c
11          )
12  } end;
```

Below we present a main program to utilize our `fib` function, which requests the first three numbers in the Fibonacci sequence:

```
14      f <- spawn (fib 0 1);
15      f.next;
16      x1:int <- recv f;
17      print x1;
18      f.next;
19      x2:int <- recv f;
20      print x2;
21      f.next;
22      x3:int <- recv f;
23      print x3;
24      f.stop;
25      wait f;
26      close m
27  } ;;
```

The function `fib` is called by spawning a new process that is associated with channel `f`. Next, we request the first three numbers in the sequence, sending the `next` label to channel `f`, on lines 15, 18 and 21, and receiving the result on this channel, on lines 16, 19 and 22. At the end we send the `stop` label through channel `f` to end the communication, on line 24 and wait for its termination, on line 25. The final step is to close the channel `m`, which is associated with the main process, on line 26.

$\triangle$

3.2 The type system

The types of the language are divided into two layers: the *process* layer and the *functional* layer, and these layers are interdependent [13].

The types for the functional part and for the concurrent part are presented in Figure 3.2 and Figure 3.3, respectively [13].

In the functional part of the language, we consider the context Ψ that holds the functional variables making the typing judgment for this part of the language $\Psi \Vdash M : \tau$, meaning that the term M has the type τ under the typing assumptions for variables in Ψ.

τ, σ	$::=$	$\tau \to \sigma \mid \ldots \mid \forall t.\, \tau \mid \mu t.\, \tau \mid t$	(ordinary functional types)
	$\mid$	$\{a : A \leftarrow \overline{a_i : A_i}\}$	process offering A along channel a, using channels a_i offering A_i

Figure 3.2: Functional types of SILL

The functional types include function types, recursive types, and other standard constructs like base types, products, and sums. We highlight the contextual monadic type $\{a : A \leftarrow \overline{a_i : A_i}\}$ that represents the type of a process offering a session A in channel a using channels a_i with types A_i, embedded as a functional value.

The typing judgment for the process layer can be expressed as $\Psi, \Gamma, \Delta \vdash P :: c : A$. It indicates that the session process P provides a given service A along a channel c, which means that the channel c follows the communication pattern specified by the session type A under the typing assumptions for ordinary variables in Ψ, the linear channels in Δ, and the shared channels in Γ. The process P can use channels c_i offering services A_i to provide the service A, leading to the linear sequent $c_1{:}A_1, \ldots, c_n{:}A_n \vdash P :: c{:}A$ [13].

A, B, C	$::=$	$\tau \supset A$	input value of type τ and continue as A
	$\mid$	$\tau \wedge A$	output value of type τ and continue as A
	$\mid$	$A \multimap B$	input channel of type A and continue as B
	$\mid$	$A \otimes B$	output fresh channel of type A and continue as B
	$\mid$	$\&\{\overline{l_j : A_j}\}$	offer choice between l_j and continue as A_j
	$\mid$	$\oplus\{\overline{l_j : A_j}\}$	provide one of the l_j and continue as A_j
	$\mid$	$\mu X.A \mid X$	recursive session type
	$\mid$	$\mathbf{1}$	terminate

Figure 3.3: Process types of SILL

In the process layer, the session types input $\tau \supset A$ and output $\tau \wedge A$ refer to the functional layer, corresponding to offer to send and receive a value of type τ, respectively. The concrete syntax for this types is τ => A and τ ^ A, respectively.

The type $A \multimap B$ represents the input of a channel of type A and continues as the session type B, and its dual type $A \otimes B$ represents the output of a channel of type A and continue as B.

The complementary types $\&\{\overline{l_j : A_j}\}$ and $\oplus\{\overline{l_j : A_j}\}$ represent the choice between l_j and continue as A_j and the selection of the label l_j and continue as A_j, respectively.

The type $\mu X.A$ represents a recursive session type where X is a type variable. Finally, the type $\mathbf{1}$ represents an inactive session.

Given the above presentation of the types in our language, we now present the typing rules for the functional and process layers [13].

We highlight the typing rules associated with the monad, the remaining typing rules are in Appendix A. The monadic value can be used to refer to processes in the functional layer as well as to communicate with processes in the process layer, it defines a runnable process that offers along channel c and uses channel a_i. All the sessions in Δ must be used.

$$\frac{\Delta = a_i{:}A_i \quad \Psi ; \Delta \vdash P :: c{:}A}{\Psi \Vdash c \leftarrow \{P\} \leftarrow \overline{a_i{:}A_i} : \{c : A \leftarrow \overline{a_i{:}A_i}\}} \; \{\,\}I$$

This rule states that if the process P is well-typed and uses the channels $\overline{a_i}$, with types A_i, to offer type A along channel c, then the functional term $c \leftarrow \{P\} \leftarrow \overline{a_i}$ is of type $\{c : A \leftarrow \overline{a_i{:}A_i}\}$.

Dually, we can create a new process M to run in parallel with the process Q using the channels $\overline{a_i}$, with types A_i, which will be consumed by the process. To achieve this we isolate these channels in Δ' so that they are no longer available to the process Q. Process Q will interact will the newly created process M through the channel c, which has type A, and will offer channel d. This rule is presented bellow:

$$\frac{\Psi \Vdash M : \{c{:}A \leftarrow \overline{a_i{:}A_i}\} \quad \Psi \; ; \Gamma ; \; \Delta', c{:}A \vdash Q :: d{:}D}{\Psi ; \Gamma ; \Delta, \Delta' \vdash c \leftarrow \texttt{spawn} \; M \; \overline{a_i} \; ; \; Q :: d{:}D} \; \{\,\}E$$

The typing system of our language ensures the properties of type preservation and progress [13]. Type preservation refers to preserving typing through all computation steps. Progress states that either the communication and computation processes must be completed and terminated or there are still active processes that are undergoing communication and computation. These two properties guarantee *deadlock freedom* and session *fidelity*.

Our language introduces a simpler approach to integrating session types in a general-purpose programming language. The integration of higher-order communication is achieved more straightforwardly and cleanly by using the contextual monad. The functional part of the language is built on λ-calculus and enriched with the linear contextual monad. This approach isolates communication and linear typing. [13].

4 The Compiler

In previous work, Geraldo developed a compiler [11, 12] for the session-typed functional language presented in Chapter 3 and our work aims to alter this compiler to improve the performance of the generated code. In this chapter, we will present the compiler developed by Geraldo.

Geraldo's compiler, written in OCaml, translates our language into valid Go code, using Go's green threads to avoid using a less efficient custom runtime. The compiler provides compile-time guarantees for the absence of communication errors and deadlocks.

The compiler is divided into two main stages: the type-checking and the compilation to the executable code.

The *type-checking* stage uses a novelty algorithm for bidirectional type-checking enabling the omission of most of the type annotations. Additionally, the compiler annotates the abstract syntax tree (AST) with typing information during this stage, including information not found in the user-level syntax. The type-checker guarantees the absence of communication errors and deadlock freedom.

In the *compilation stage*, we distinguish two substages: the first stage is the *preamble generation*, wherein each session type is transformed into a set of types and associated methods, and each type represents a step in the session. The second step of the compilation stage is the *code generation*, where the instructions of the given program are translated to Go code.

4.1 Type-Checking

The type-check phase applies the bidirectional type-check presented in Subsection 2.2.1, and for this it was necessary to reformulate the typing judgments $\Psi \Vdash M : \tau$ and $\Psi, \Gamma, \Delta \vdash P :: c : A$ to fit the bidirectional type-checking algorithm for checking and synthesis judgments [12].

For the functional part of our language, we apply the judgments $\Psi \Vdash M \Rightarrow T$ and

$\Psi \Vdash P \Leftarrow T$. The former denotes that M synthesizes a type T under the context Ψ, and the latter denotes that P checks against a type T under the context Ψ [12]. In the declarative system, rules fall into two categories: synthesis rules and checking rules. This makes it possible for the omission of most of the type annotations, which are only required for top-level definitions.

The process layer must consider the linear context Δ, and all sessions within this context should be fully utilized. To deal with this separating the context for input and output channels, resulting in the judgments $\Psi, \Delta_I/\Delta_O \vdash P \Rightarrow c : A$ and $\Psi, \Delta_I/\Delta_O \vdash P \Leftarrow c : A$. In the context Δ_I, all channels must be used by process P, and in the context Δ_O are all the channels that remain unused after the execution of P, and must still be used.

The bidirectional type-checking technique is implemented in both layers of the language. The functional layer has two functions: one for checking functional values against a given type and one for synthesizing a type from a functional value. These functions receive the linear context to keep track of the available channels. In the process layer only a synthesis function is needed, the checking is done by combining the synthesis function with the subtyping function to compare the session types. Our language employs subtyping to manage the folding and unfolding of recursive session types.

4.2 Compilation

Rather than using a custom runtime, our compiler compiles our language to Go so that we could leverage its efficient channel-based concurrency primitives and lightweight threads.

A process in the form of $c \leftarrow \{P\} \leftarrow \tilde{d}$ is compiled into a Go function that takes the channels c and $\tilde{d}$ as arguments. The compilation of functions is a straightforward translation to Go functions.

The typing discipline of Go channels locks the channel type upon creation. However, in our session-typed languages, the payload of a channel changes over time, as the session is carried out. A simple Go channel cannot implement a term such as `send c 4; send true; close c`, that first sends an integer an and then a boolean. This problem is addressed in two ways: the multichannel approach and the single-channel approach.

In the *multichannel* approach, the type information gathered during the type-checking phase is utilized to link a session to several Go channels, one for every step of the session. Moreover, each operation of the session sends the channel for the next operation in addition to the intended message.

The *single-channel* approach creates only one channel. To overcome the problem of types passed through that channel, its type is set to `interface{}`, indicating that the channel can use any type. This approach involves the constant casting of data during channel communication, which may lead to safety issues. However, the type safety of the data is ensured by the type-checking phase performed prior to code generation. This approach proved itself to be more efficient than the multichannel approach, as it avoids the creation of multiple channels, an operation that has some costs associated with it.

Regardless of the chosen strategy, session types need their inherently stateful behavior to be encoded, and the change in their state must be captured within the session. To accomplish this, each session type is represented as a type in Go, which has functions to perform essential communications and move the session to the succeeding state, creating a state machine. The custom Go types save information regarding the type name, the type of the exchanged data, and details about the next state of the session. For a choice type, in addition to the information mentioned above, the type also contains the mapping of the labels to their respective next states.

The initial stage of code generation involves creating the *preamble*, which consists of generating the types and methods that represent the session types. These types are gathered from the AST, and the Go types are produced for each session state, along with the necessary methods to progress the session to the following state. The Example 4.1 demonstrates the compilation of the session type $int \wedge bool \wedge \mathbf{1}$.

Example 4.1. For session type $int \wedge bool \wedge \mathbf{1}$ the following Go code is generated, using the single-channel approach:

```go
type send_int struct {
    c chan interface{}
    next *send_bool
}
func init_send_int (c chan interface{}) *send_int {
    return &send_int{ c, nil }
}
func (x *send_int) Send(v int) *send_bool {
    //...
    return x.next
}
func (x *send_int) Recv() (int, *send_bool) {
    //...
    return (<-x.c).(int),x.next
}
```

First, the struct type `send_int` is defined, containing the channel `c` and a pointer to the next state of the session, `next`. The `Send` and `Recv` methods return the next state of the session, and the `Recv` method also returns the received value.

△

When a session is initiated only the outer state is initialized, the `Send` and `Recv` methods initiate the next state of the session, as needed. This *lazy* initialization is done to avoid the creation of unnecessary channels.

After generating the preamble, the next step is compiling the program's declarations. This involves translating the AST into Go code and utilizing the types and methods created in the preamble to represent the session types.

The compilation of the forward operation is also worth mentioning. The compilation of the forwarding construct `fwd` d c requires that channels d and c have the same type and synthesizes a process expression by analyzing the forwarded type. This process redirects messages from the ambient channel to the offered channel and vice-versa, for example, if the forwarded type is an input, the forwarder first receives on c and then sends along d. If the forward type is recursive, the forwarder code is simply embedded inside an infinite loop, ensuring proper variable usage throughout all loop iterations and loop termination. If the type is not recursive, the compilation outcome is just a channel redirection, receiving in d and sending in c.

5 Optimizations

The SILL compiler, described in Chapter 4, was not designed to prioritize code performance, hence generating generally inefficient code. Therefore, our work aims to improve the efficiency of generated code by enhancing the current compiler with type-based optimizations . To achieve this goal we added an *optimization phase* between the type-checking phase and the code generation phase.

Prior to the addition of the optimization layer, we began by improving the performance of the compiler itself. In the code generation phase, a large string with the generated code is produced by using multiple concatenation operations. Nevertheless, this strategy has some performance drawbacks as strings in Ocaml are immutable [40] and concatenating strings requires creating a new string. To address this, we use a buffer [41] to save the result of code generation, and then output the contents of the buffer to a file.

Previous performance analyses, undertaken by Geraldo, showed two crucial points to optimize: the minimization of the number of messages exchanged, by packing several messages into one communication, and optimizing the implementation of the `fwd` construct to not spawn an additional goroutine, if possible. Considering these opportunities we developed three optimizations: forwarding optimization, optimization of recursion, and packing messages. These optimizations will be presented in detail in the following sections.

The optimization layer is inserted between the type-checking phase and the code generation phase, as illustrated in Figure 5.1, depicting all phases of our compiler.

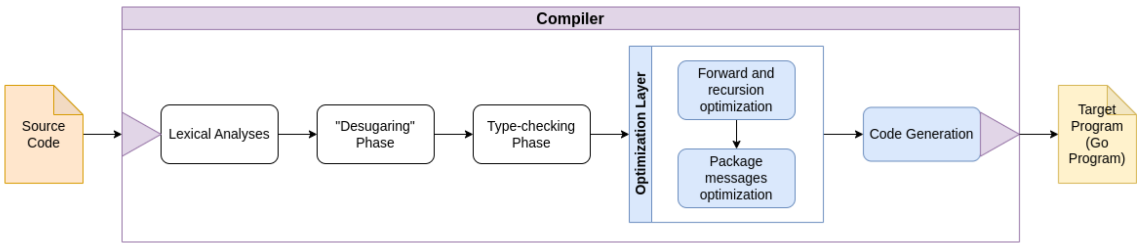

Figure 5.1: Compiler phases

The program is inputted into the compiler and undergoes a lexical analysis. If any syntax errors are found, the analysis fails. If it succeeds, a preliminary AST of our program is built. Then, a simple desugaring is performed to enhance the representation. This is followed by the type-checking phase, which ensures the absence of communication errors and the freedom from deadlocks. After this, the optimization phase is executed to identify opportunities for optimization and improve representation accordingly. The first step combines forwarding and recursion optimizations before moving on to the packing optimization step. Declarations with recursive optimizations are excluded from packing optimization since recursion hinders the identification of possible messages to pack. Finally, the code generation phase generates valid Go code by translating the representatives produced and curated in the previous phases.

The optimization phase comprises three main optimizations, that will be presented in the following sections:

- The **forwarding optimization** simplifies the redirection of messages between channels, reducing synchronization points and the creation of channels;
- **Recursion optimization** converts recursive functions into loops to minimize the overhead of function calls and recursion expenses;
- **Packing messages optimization** combines multiple messages into a single message to decrease the number of messages exchanged and minimize synchronization costs.

5.1 The cost of forwarding

Although creating channels and goroutines in Go is relatively inexpensive, in larger programs these costs are not insignificant so they should be minimized. Due to how the channel forwarding construct `fwd` is compiled, each instance of its use generates an additional channel and spawns another goroutine.

Let us consider the following functions:

```
plus_one : int -> {int ^ @}
fun n -> c <- {
    send c (n+1);
    close c
}
end;
plus_two : int -> {int ^ @}
fun n -> c <- {
    d <- spawn (plus_one (n+1));
    fwd d c
}
end;
```

When `plus_two` is executed, it spawns a process given by `plus_one`, offering its session on channel `d` and then immediately forwards the behavior on channel `d` to its offering channel `c`.

Recalling that processes are compiled to Go closures, the compilation of `plus_two` will result in a closure value (parametrized by the offered channel `c`) which includes the creation of a new channel that will be used to instantiate the closure returned by the call to `plus_one`. Then, the resulting forwarder will forward between the newly created channel and the offered channel `c`.

Drawing inspiration from the classical tail-call optimization we can foresee a better implementation for this scenario: we need not create a new channel or produce the forwarder, we can simply instantiate the closure that results from the execution of `plus_one` directly with the offered c channel. This allows for the reduction of the number of goroutines, channels, and the subsequent synchronization steps required by the forwarding of communication, which in large programs easily builds up and negatively affects performance.

We can informally state that these modifications preserve the programs' semantics considering previous works [42–44] that prove process equivalence in similar cases. Instantiating the closure directly with the offered channel d is semantically equivalent to instantiating the process in channel c and then forwarding the channel c to channel d:

$$P(c) \mid\mid \texttt{fwd}\ c\ d \text{ is equivalent to } P(d)$$

5.1.1 Implementation

To implement this optimization we need a new node to represent these situations in the AST. So we added a new node `TailSpawn` to the process layer:

```
TailSpawn of var *
             exp *
             stype option *
             var list *
             bool *
             var option *
             var list
```

The `TailSpawn` corresponds to a new Ocaml constructor, and its definition needs seven fields, whose types are indicated in the definition above. This new constructor will represent a new node type in the AST and will be used to represent the forwarding optimization. This node has a similar structure as the `Spawn` node but does not have a continuation process and has three additional fields: a boolean, to indicate if recursion has been detected and two other fields to assist the recursion optimization, which will be explained in Section 5.2. The first `var` field holds the channel created by the spawn, the `exp` is the expression to be spawned, followed by its type and the list of additional channels used by the spawned process.

We begin by traversing the AST generated from the type-checking phase and searching for a node that corresponds to a `Spawn`. Upon locating one of these nodes, the goal is to identify the `d <- spawn M; fwd d c` pattern, a `spawn` followed by a `forward` to a channel. This optimization is also possible if there are nodes of type `Print` or `If` between the `Spawn` and the `Fwd` nodes since these are not process operations and therefore don't make any communications and don't interfere with the spawning. So, when a `Spawn` node is located the `optimize_spawn_forward` is called, which is presented in Listing 5.1.

```
 1  rec optimize_spawn_forward decl_name next spawn ctxt arg =
 2      match spawn with
 3      | Spawn (channel, exp, opt, _, args) -> (
 4          match next with
 5          | Fwd (st, dst, src) ->
 6              if src = channel then (
 7                  let is_rec = check_recursion exp decl_name in
 8                  if is_rec then (
 9                      is_rec_decl := true;
10                      TailSpawn (src, exp, opt, args, is_recursive, arg, ctxt))
11                  else
12                      TailSpawn (src, exp, opt, args, is_recursive, None, []))
13              else Spawn (channel,
14                          exp,
15                          opt,
16                          optimize_proc decl_name next ctxt,
17                          args)
18          | Print (exp, proc) ->
19              Print (exp, optimize_spawn_forward decl_name proc spawn ctxt)
20          | If (exp, proc1, proc2) ->
21              If ( exp,
22                  optimize_spawn_forward decl_name proc1 spawn ctxt,
23                  optimize_spawn_forward decl_name proc2 spawn ctxt )
24          | _ -> Spawn (channel,
25                      exp,
26                      opt,
27                      optimize_proc decl_name next ctxt,
28                      args))
29      | _ -> spawn
```

Listing 5.1: Function `optimize_spawn_forward` that performs the forward optimization

This function receives five arguments. The `spawn` is the node that corresponds to the spawn node found and `next` is the continuation process. The `decl_name`, `ctxt`, and `arg` are used for the recursion optimization, which will be explained in Section 5.2. The goal is to identify the pattern `d <- spawn M; fwd d c`, so the function checks if the continuation process is a `Fwd` node, in lines 4 and 5 of Listing 5.1, and if the "source" channel of the `Fwd` node is the same as the channel of the `Spawn` node, in line 6. If it is then the wanted pattern was encountered and we replace it with our `TailSpawn` node, on line 12 of Listing 5.1. Otherwise, we maintain the `Spawn` and `Fwd` nodes, in lines 13 to 17. If `Print` or `If` nodes are found before the `Fwd` node, in lines 18 and 20 respectively, we can still perform the optimization, so we call the `optimize_spawn_forward` recursively on the continuation process until we find a `Fwd` node. If any other node is found, in line 24, we cannot perform the optimization, so we maintain the `Spawn` and continue the search for other opportunities to optimize, calling the `optimize_proc` on the continuation process.

Now that we have identified the pattern to optimize, we move on to the code generation phase. Here, the closure of the spawned process is directly instantiated in the "destination" channel of the `Fwd` node, reducing the number of channels created and goroutines spawned.

To provide a clearer illustration of the outcome resulting from the forward optimization, we showcase the variances in the generated code with and without the optimization in Listing 5.2. This refers to the function `plus_two` discussed earlier in this section. The optimized version solely calls the function `plus_one`, without creating a new goroutine, and avoids some superfluous operations and the creation of unnecessary channels.

```go
func plus_two(n int) func(_x *_state_0) {
  return func(c *_state_0) {
    d := init_state_0(make(chan interface{}))
    go plus_one((n + 1))(d)
    // FWD c d Start
    cd, d0 := d.Recv()
    c0 := c.Send(cd)
    d0.Recv()
    c0.Send(nil)
    return
    // FWD c d End
  }
}
```

(a) Generated code without the forwarding optimization

```go
func plus_two(n int) func(_x *_state_0) {
  return func(c *_state_0) {
    plus_one((n + 1))(c)
  }
}
```

(b) Generated code for optimized forward

Listing 5.2: Differences in the forwarding compilation

5.2 Recursion Optimization

In our language, recursion is accomplished using the `spawn` and `fwd` constructs, following a pattern similar to the one identified in Section 5.1. This explains why these two optimizations are combined in the same optimization phase: they are closely related.

To illustrate the problem let us consider the following example, that produces an infinite stream of integers:

```
stype IntStream rec x. int ^ x;
nats : int -> {IntStream}
fun n -> c <- {
    send c n;
    d <- spawn (nats (n+1));
    fwd d c
} end;
```

The resulting code will spawn the recursive call and forward to the result channel `c`. However, each recursive call will itself result in a new forwarder process, effectively establishing a linked chain of forwarders. These forwarders must synchronize to communicate up the chain, eventually reaching the initially offered result channel.

The intended alternative is to modify the forward recursive pattern to run on the same goroutine instead of creating a new one for each iteration, and use a new communication channel. It is proposed that whenever possible, recursive processes should be turned into loops, thus avoiding the creation of additional processes and the costs associated with recursion.

The reasoning behind the preservation of the original semantics is comparable to the one previously demonstrated in Section 5.1 for the forwarding optimization, as this is a special case of the forwarding optimization:

$$P(c) \mid\mid \texttt{fwd}\ c\ d \text{ is equivalent to } P(d)\ [42\text{–}44]$$

Furthermore, the equivalence of iteration and recursion is a well-established concept in computer science and programming theory. Any problem that can be solved using recursion can also be solved using iteration, and vice versa.

5.2.1 Implementation

As seen, recursion follows the previous pattern of `d <- spawn M; fwd d c`, with an added condition: the spawned process must call the current declaration. Hence, the `optimize_spawn_forward` function in Listing 5.1 requires the `decl_name` argument to verify if the spawned process is a call to the current declaration. The `ctxt` comprises all channels utilized within the function, for updating them during code generation. The `arg` has an `Option` field containing the recursive function's argument, if any.

In line 7 of Listing 5.1, the function `check_recursion` checks whether the process spawn is a recursive call. If it is, then we replace the found pattern by the `TailSpawn` node - as previously introduced - and include the channels used by the function and the argument of the recursive function, as seen in line 10.

Additionally, while traversing the tree if we find recursion we change the `FunDef` node that contains the recursion to a new node `RecFunDef`:

```
RecFunDef of var * ty option * exp * ty option
```

This new Ocaml constructor represents a AST node that contains identical information to a FunDef node and serves only to aid the code generation phase in identifying recursive functions. It includes fields for the argument name, argument type, function body, and return type.

In the code generation phase when a `RecFunDef` node is found the recursion is transformed into a cycle, updating the necessary channels and the arguments in each iteration.

In Listing 5.3, we present a comparison between the generated code with and without the recursion optimization for the `nats` function from the previous example. The optimized version avoids creating new goroutines and channels by integrating the function body into a loop, where arguments and channels are updated at the end of each iteration.

```go
func nats(n int) func(_x *_state_0) {
  return func(c *_state_0) {
    c0 := c.Send(n)
    d := init_state_0(make(chan interface{}))
    go nats((n + 1))(d)
    // FWD c d Start
    for {
      c0d, c0_d := d.Recv()
      d = c0_d
      c0 = c0.Send(c0d)
    }
    // FWD c d End
  }
}
```

(a) Generated code without the recursion optimization

```go
func nats(n int) func(_x *_state_0) {
  return func(c *_state_0) {
    for {
      c0 := c.Send(n)
      // Update arguments
      n = (n + 1)
      // Update channels
      c = c0
    }
  }
}
```

(b) Generated code for optimized recursion

Listing 5.3: Differences in the recursion compilation

5.3 Packing messages

The compiler collects type information during type-checking and from it, we can determine the directionality of a message. Using this information, messages in the same direction could potentially be bundled together, reducing the burden of synchronization.

Consider the following example:

```
1  send_ints : int -> {int ^ int ^ int ^ @}
2  fun n -> c <- {
3      send c n;
4      send c (n+1);
5      send c (n+2);
6      close c
7  } end;
8  ;; m <- {
9      e <- spawn (send_ints 1);
10     a1:int <- recv e;
11     print a1;
12     a2:int <- recv e;
13     print a2;
14     a3:int <- recv e;
15     print a3;
16     wait e;
17     close m
18 };;
```

The function `send_ints` sends three messages along its offering channel `c`. Dually the main, which starts on line 8 process awaits to receive three messages from a channel created by spawning `send_ints`.

Since we explicitly send three messages on channel `c`, the generated code will require three synchronization points, one for each message. We propose to change code generation such that these three messages are sent all together in a single communication bundle.

This optimization preserves the semantics of the original program, as the order of the messages is maintained, and the number of messages exchanged is the same. The only difference is that the messages are sent in a single communication instead of three. The reason why the semantics is preserved relies on session types isomorphisms [42], is analogous to the fact that the type $A \to B \to C$ is isomorphic to $(A \otimes B) \to C$. In most scenarios, the behavior remains unchanged except for non-terminating programs. In those cases, despite preserving the non-termination, the behavior could vary since a non-terminating message in certain situations may prevent the entire bundle from being sent, whereas in the original program, the terminating messages would be sent.

5.3.1 Implementation

This optimization is more intricate than the previous ones and should be approached in three steps: firstly, identifying all opportunities to group sent messages and grouping them;

secondly, locating all corresponding receives and grouping them; and finally, adjusting the original types to match the modifications.

In code generation, we explore two distinct approaches: one that sends the package messages using a `interface{}` type list that requires casting upon receiving, and another that utilizes a structure with a field for each message, thereby avoiding the need for casting.

Grouping sent messages We begin by identifying the `Send` nodes that can be grouped, as chaining send operations is more restrictive than grouping receives. This optimization requires traversing the AST to locate all `Send` nodes deemed suitable for grouping. To enable the packaging of multiple sends, a `Send` node must have another `Send` node as a continuation process and the same destination channel. This identification is made by the `pack_sends` function presented in Listing 5.4.

```
1  let rec pack_sends next proc channel =
2      match proc with
3      | Send (c, to_send, send_type, next) ->
4          if c = channel then
5              let new_next, list_sends, list_types = pack_sends next channel in
6              (new_next, to_send :: list_sends, send_type :: list_types)
7          else (proc, [], []) (*channel is not the same - break chain*)
8      | _ -> (proc, [], [])
```

Listing 5.4: Function `pack_sends` to pack multiple sends

The `pack_sends` function is initially invoked with the first `Send` node of the chain to validate that the associated channel of all subsequent `Send`'s, if present, is the same, this verification occurs in line 4 of Listing 5.4. This function is recursively invoked for the continuation process until the channels don't match, in line 7 or a non-`Send` node is located, in line 8. A tuple is then returned, comprising the relevant data on the `Send` chain, including its continuation process, a list of all expressions to send, and a list of its types. These details are then used in constructing a new node - the `MultiSend`:

```
MultiSend of var * exp list * ty option list * proc
```

The `MultiSend` Ocaml constructor represents an AST node with four fields. These fields are designated for storing the sending channel, a list of expressions and their corresponding types, and the continuation process. The `MultiSend` node replaces the chains of `Send` nodes found while traversing the AST, as shown in Listing 5.5.

The `bundle_send_proc` function of Listing 5.5 takes as arguments a process node (`proc`) for analysis, the offering channel of the process (`off_ch`), the declaration name in which the process is included (`decl_name`), a context of channels available for the packing optimization (`ctxt`, gathered during the tree traversal), and a list of all recursive functions (`rec_fun`). The information regarding recursive functions is important to prevent message packing within them since recursion hinders this optimization, channels created by spawning a recursive function will not be added to the available context.

The challenge of optimizing recursive functions with package massages lies in the fact that although send operations can be chained, we cannot guarantee that the receives will also be in a chain. They may occur in different iterations of the recursion, which makes it unsuitable for bundled messages.

```
1  let rec bundle_send_proc proc off_ch decl_name ctxt rec_fun =
2      match proc with
3      | Send (ch, to_send, s_type, next) ->
4          if Map.mem ch ctxt || ch = off_ch then
5              let next_proc, list_send, types = pack_sends proc ch in
6              if List.length list_send > 1 then
7              (* MultiSend detected *)
8              let multi_type =
9                  STMultiSend
10                 ( List.map (fun t -> Option.get t) types,
11                     get_type_proc off_ch next_proc ) in
12             let found_multisend =
13                 if off_ch = ch then External multi_type
14                 else
15                 let associated_fun =
16                     match Map.find_opt ch ctxt with
17                     | None -> decl_name
18                     | Some v -> v  in
19                 Internal (Some associated_fun, multi_type) in
20             let detected, new_next =
21                 bundle_send_proc next_proc off_ch decl_name ctxt rec_fun in
22             (found_multisend :: detected,
23             MultiSend (ch, list_send, types, new_next))
24             else  (* No multisend detected *)
25             let detected, new_next =
26                 bundle_send_proc next off_ch decl_name ctxt rec_fun in
27             (detected, Send (ch, to_send, s_type, new_next))
28         else (* No multisend detected *)
29             ...
30     | ...
```

Listing 5.5: Function `bundle_send_proc` to search for chains of sends

The `bundle_send_proc` traverses the tree to locate a `Send` node. When one is found it checks if it is eligible for a packing optimization, meaning that the channel through which the message is sent exists in the available context or is the offered channel, this verification occurs in line 4. If deemed for optimization, it invokes the `pack_sends` function, on line 5 to examine if there is a sequence of `Send` nodes, as previously seen.

If a suitable chain of sends is detected, meaning that the list returned by `pack_sends` has more than the initial `Send` node, then it creates a `MultiSend` node. It is necessary to introduce a new type to match the new node, so we added the `STMultiSend` type to the `stype` type:

```
STMultiSend of ty list * stype
```

The STMultiSend has a list of the types of expressions sent and the type of the continuation process. The results of the pack_sends are used to construct the STMultiSend, in line 8 of Listing 5.5, and the MultiSend node, in line 23 of Listing 5.5.

In order to match the MultiSend's to their corresponding receive chain, the details of the MultiSend's constructed are essential. Therefore, in addition to the modified process node, the bundle_send_proc also returns a list of all MultiSend's identified. For storing this data, we have two auxiliary Ocaml structures:

```
type found_multisends =
  | External of stype
  | Internal of var option * stype
```

We use the structure External to store a STMultiSend associated with the original channel. The construct Internal is used to store a STMultiSend associated with a channel created by a spawn and the var option is used to store the name of the function that created the channel, if it exists. This differentiation is helpful in accurately linking send chains with their respective receive chains. We build the necessary auxiliary structures in line 12 of Listing 5.5.

If there is no chain of Send nodes, then it calls the bundle_send_proc recursively on the continuation process, and a Send node is maintained, as seen in line 27 of Listing 5.5.

At the conclusion of this stage, the AST was reconstructed using the MultiSend nodes instead of chains of Send nodes. The initial declarations are kept for use in instances where the optimized variant cannot be applied, such as recursive functions. Furthermore, we create a map that associates each declaration with the found MultiSend's, which is used to find the matching receiver chains.

Grouping received messages The subsequent step involves identifying chains of receives corresponding to the multi-sends detected in the previous phase. The grouping of receives entails no strict limitations provided they are on the same channel, and other nodes may be present in between. The grouping operation begins by traversing the AST utilizing the function bundle_receive_proc presented in Listing 5.6 (a) and Listing 5.6 (b). This function takes as argument a process node (proc to analyze along with a map (send_map) containing all discovered STMultiSends, a channel context (ctxt) containing all available channels for packing receives, and the current declaration name (fun_name).

Context is established by navigating the AST, when a new channel is created by a spawn as seen in Listing 5.6 (a). When a spawn is detected, we begin by checking if it is a function call, on line 5, verifying that the expression of the spawn is of type FunApp. If it is, we consult the send_map, in line 7, to determine if that function has a multi-send optimized form. If the function called has a multi-send optimization, we need to verify if it is available to be used by an external process by checking if it is an External type, in lines 14 to 17. If all these conditions are true, we add the channel to the context of available channels, in lines 19 to 21.

If the spawn is not a function application, but rather the launch of a new anonymous process, we check if the current declaration with the name `fun_name` has an associated multi-send optimization, in line 26 of Listing 5.6 (a) and verify if it is of type `Internal`, on lines 30 to 33. If so, the channel is added to the context.

```
 1 let rec bundle_receive_proc proc send_map ctxt fun_name =
 2  match proc with
 3  | Spawn (ch, exp, st, p, v_list) -> (
 4    match exp with
 5    | FunApp (Var name, _) -> (
 6      let opt_name = name ^ "_optimized" in
 7      match Map.find_opt opt_name send_map with
 8      | None -> (*No matching multisend found*)
 9         let was_opt, new_proc =
10         bundle_receive_proc p send_map channels_ctxt fun_name in
11         (was_opt, Spawn (ch, exp, st, new_proc, v_list))
12      | Some multisends ->
13        (* The multisend of a function applied must be External *)
14        let extern =
15         List.filter
16           (fun x -> match x with External _ -> true | _ -> false)
17           multisends in
18        (* Add channel with matching multisend to the context *)
19        let new_ctxt =
20         if List.length extern > 0 then Map.add ch extern ctxt
21         else ctxt in
22        let was_opt, new_proc =
23         bundle_receive_proc p send_map new_ctxt fun_name in
24        (was_opt, Spawn (ch, exp, st, new_proc, v_list)))
25    | _ -> (
26      match Map.find_opt fun_name send_map with
27      | Some multisends ->
28        (* The multisends found must be Internal *)
29        (* Add channel with matching multisend to the context *)
30        let internal =
31         List.filter
32           (fun x -> match x with Internal _ -> true | _ -> false)
33           multisends in
34        let new_ctxt =
35         if List.length internal > 0 then Map.add ch internal ctxt
36         else ctxt in
37        let was_opt, exp1 =
38         bundle_receive_exp exp send_map new_ctxt fun_name in
39        let was_opt1, new_proc =
40         bundle_receive_proc p send_map new_ctxt fun_name in
41        (was_opt || was_opt1, Spawn (ch, exp1, st, new_proc, v_list))
42      | None ->
43        (*No matching multisend found*)
44        let was_opt1, exp1 = bundle_receive_exp exp send_map ctxt fun_name in
45        let was_opt2, new_proc = bundle_receive_proc p send_map ctxt fun_name in
46        (was_opt1 || was_opt2, Spawn (ch, exp1, st, new_proc, v_list))))
47  | ...
```

Listing 5.6 (a): Function `bundle_receive_proc` - Spawn

When a `Rcv` node is encountered, the code checks for its eligibility for optimization, as depicted in Listing 5.6 (b). Firstly, it confirms if the channel in the receive operation is present in the available context, in line 4. If so, it implies the existence of a matching multi-send, requiring the creation of a multi-receive holding identical types and number of messages. Subsequently, information on the number of messages and their types is extracted from the identified multi-send, as in lines 10 and 13 of Listing 5.6 (b). This information, along with the receive channel and the initial `Rcv` node of the chain, is used to invoke the function `pack_receives`, at line 16 of Listing 5.6 (b), which will be explained later.

```
1  let rec bundle_receive_proc proc send_map ctxt fun_name =
2    match proc with
3    | Recv (v1, ch, ty, p) -> (
4        match Map.find_opt ch ctxt with
5        | None -> (* No matching multisends*)
6            let was_opt, new_next = bundle_receive_proc p send_map ctxt fun_name in
7            (was_opt, Recv (v1, ch, ty, new_next))
8        | Some multisends ->
9            let multisend = List.hd multisends in
10           let n = match multisend with
11               | External s | Internal (_, s) -> (
12                   match s with STMultiSend (l, _) -> List.length l | _ -> 0) in
13           let types = match multisend with
14               | External s | Internal (_, s) -> (
15                   match s with STMultiSend (l, _) -> l | _ -> []) in
16           let vars, types, nexts =  pack_receives proc ch types n in
17           let new_proc =  remove_receives nexts in
18           let new_ctxt = (* Remove the used multisend from context *)
19               if List.length multisends > 1 then
20               let m = Map.remove ch ctxt in
21               Map.add ch (List.tl multisends) m
22               else Map.remove ch ctxt in
23           let _, proc1 = bundle_receive_proc new_proc send_map new_ctxt fun_name in
24           (true, MultiRecv (ch, vars, types, proc1))
25   | ...
```

Listing 5.6 (b): Function `bundle_receive_proc` - Receive

The `pack_receives` function returns details imperative for creating our new node `MultiRcv`, inclusive of the names of the variables that will store the received values, the types of the received values, and a list of the processes identified between the `Rcv` nodes, necessary to reconstruct the tree. To rebuild the tree with the intermediate processes found, the `remove_receives` function is called in line 17 of Listing 5.6 (b). This function takes the list of processes and sets the continuation process of a process as the succeeding process in the list.

Since the multi-send is already matched to the respective multi-receive, it should no longer be available for new matches unless it is added again. Therefore, we have updated the context of the available channels in line 18 of Listing 5.6 (b).

Finally, we build the `MultiRcv` node, as seen in line 24 of Listing 5.6 (b), and return it along with a boolean indicating if the process was optimized. The `MultiRcv` node has the following structure:

```
MultiRecv of var * var list * ty option list * proc
```

and has fields to store the channel from where to receive the messages, a list of variable names where to store the received values, a list of the types of the received values, and the continuation process. To match this node a new type is needed:

```
STMultiRecv of ty list * stype
```

This type has a list of types received and the type of the continuation process.

In Listing 5.7 we present the `pack_receives` function. The function takes as arguments the first receive of the chain (`proc`), the channel in which all the receives must be made (`channel`), the types to receive (`types`), and the number of messages of the corresponding multi-send (`size`).

```
let rec pack_receives proc channel types size =
    match proc with
    | Recv (v, ch, receive_type, next) ->
        if ch = channel && size > 0 then
            let n = List.length types - size in
            if Option.get receive_type = List.nth types n then
            (* matching type *)
            let vars, lst_types, others =
                pack_receives next channel types (size - 1) in
            (v :: vars, receive_type :: lst_types, others)
            else assert false (*types don't match, something is wrong*)
        else
            let vars, lst_types, others = pack_receives next channel types size in
            (vars, lst_types, proc :: others)
    | Print (_, next)
    | Send (_, _, _, next)
    | ...
    | Wait (_, next) ->
        let vars, lst_types, others = pack_receives next channel types size in
        (vars, lst_types, proc :: others)
    | Close _ | Fwd (_, _, _) | TailSpawn (_, _, _, _, _, _, _) -> ([], [], [ proc ])
    | If (e1, p1, p2) ->
        let new_p1, lst_types1 = modify_branch_proc p1 channel types size in
        let new_p2, _ = modify_branch_proc p2 channel types size in
        let vars = List.mapi (fun i _ -> "var_multi_" ^ string_of_int i) lst_types1 in
        (vars, lst_types1, [ If (e1, new_p1, new_p2) ])
    | Choice (v, list) -> ...
```

Listing 5.7: Function `pack_receives` to pack multiple receives

The `pack_receives` function returns a tuple with a list of the variable names where the received values will be stored, the types of the received values, and a list of the processes identified between the `Rcv` nodes. The function is recursively called on the continuation process. When a `Rcv` node is reached with a matching channel and type, the function adds the variable name to the list being returned, the type to the list of types, and continues processing the continuation process while decreasing the number of receives to find, as shown in lines 8 to 10 of Listing 5.7. If other nodes are found, we add them to the list of processes and continue to analyze the continuation process, as seen in lines 15 to 21.

In cases of branching nodes, such as `If` and `Choice` we must consider that the receives may be inside the branches, and to successfully group the receives we must extract them outside the branching node. As the types match, it is guaranteed that each branch will have the same number of `Rcv` nodes, but the variables where values are stored may have different names and purposes. Thus, we must extract the `Rcv` nodes and rename the variables to ensure uniformity of names between both branches. The removal of `Rcv` nodes and the renaming of variables is done by the function `modify_branch_proc`, called for each branch, in line 23 and 24 of Listing 5.7.

After processing each declaration, we perform an optimization check using the boolean value returned by the `bundle_receive_proc` function. If optimized, the original version is retained and the optimized version is added. This boolean also enables us to discern whether a declaration utilizes an optimization, so we save this data to utilize in the subsequent phase of the optimization.

Adjusting types In the previous phases, a considerable amount of changes were made to the sessions that must be carried out by the final program. For instance, three consecutive send operations are replaced by a single `MultiSend` node. The types annotation in the tree should reflect this change for a proper compilation. Furthermore, we have the original and the optimized declarations so we must ensure that the optimized call is used whenever possible.

Since type-checking was performed beforehand, we can guarantee that the program is well-typed. Therefore, to modify the types, we simply traverse the tree and match the corresponding type for each node in the tree. While traversing the tree to change its type, the program checks the spawned functions for an optimized version. If an optimized version is available and applicable, the call is substituted with the optimized version.

This phase ensures that the session types are altered to the correct `STMultiRecv` and `STMultiSend` so that they represent only one state in the session, sending or receiving multiple values at once.

Code generation This optimization involves modifying not only the compilation of declarations but also the compilation of the preamble. During preamble generation, the `STMultiRecv` and `STMultiSend` types are translated into Go types to represent that stage in the session. These types differ from the standard `STSend` and `STRcv` types in that they allow for the sending and receiving of multiple values. So, in the `Send` method of the state type, all values to be sent are included as arguments, and in the `Recv` method of the type, all received values are returned, following the interface:

```go
type _state_0 struct {...}
func init_state_0(c chan interface{}) *_state_0 { ... }
func (x *_state_0) Send(v0 int, v1 int, ... , v_n int) *_state_1 {...}
func (x *_state_0) Recv() (int, ... ,int, int, *_state_1) {...}
```

Two approaches were tested for the send and receive of multiple values. In the first approach, the values are transmitted as a list of type `interface{}`, requiring casting upon reception using the annotations of `STMultiRecv` type. This constant casting may constitute a performance bottleneck. The alternative entails constructing a custom struct with designated fields for each message, with the corresponding types, thus eliminating the need for casting upon value reception.

In Listing 5.8 (a), the original version of the generated code for the previous example is presented, featuring unique states for each transmitted message: `_state_0`, `_state_1`, and `_state_2`. The associated code for each state is identical, so we present the details solely for `_state_2`. Finally, `_state_3` denotes the session's termination. In the function `send_ints`, there are three separate send operations, one for each message, while the function `main` has corresponding individual receives intertwined with print operations.

Listing 5.8 (b) presents an optimized version that uses a list to transmit multiple values. This version simplifies by having only two states: `_state_0` transmits a bundle of three messages, and `_state_1` represents the session's termination. The `Send` method of `_state_0` constructs a list of type `interface{}` with the input parameters and transmits it via the associated channel. In the `Recv` function, a list is received via the associated channel, and the function returns the multiple values that require prior casting. Using this state the function `send_ints_optimized` only has one send operation, and the function `main` has a single receive operation, which returns multiple values. In `main` we can also observe that the receive operations were pulled together, pushing the previously interleaved print operation together for later execution.

The optimized version using a struct is presented in Listing 5.8 (c). This version, similar to the list version, only has two states. The custom type `_multisend_type_state_0` serves to bundle the three messages, and has a field for each message, with the corresponding type. The `Send` operation of `_state_0` builds a struct of type `_multisend_type_state_0` with the input parameters and sends it via the associated channel. The `Recv` function receives a struct of type `_multisend_type_state_0` via the associated channel and returns the struct fields, with no casting required. As the `Send` and `Recv` methods use the same interface as the list version the compilation of the functions `send_ints_optimized` and `main` is identical to the list version and therefore omitted.

The differences in performance of the two versions will be further elaborated in Chapter 6.

```go
type _state_3 struct {...}
func init_state_3(c chan interface{}) *_state_3 { ... }
func (x *_state_3) Send(v interface{})          { x.c <- v }
func (x *_state_3) Recv() interface{}           { return <-x.c }

type _state_2 struct {...}
func init_state_2(c chan interface{}) *_state_2 { ... }
func (x *_state_2) Send(v int) *_state_3 {
    //...
    return x.next
}
func (x *_state_2) Recv() (int, *_state_3) {
    //...
    return (<-x.c).(int), x.next
}

type _state_1 struct {...}
func init_state_1(c chan interface{}) *_state_1 { ... }
func (x *_state_1) Send(v int) *_state_2 {...}
func (x *_state_1) Recv() (int, *_state_2) {...}

type _state_0 struct {...}
func init_state_0(c chan interface{}) *_state_0 {...}
func (x *_state_0) Send(v int) *_state_1 {...}
func (x *_state_0) Recv() (int, *_state_1) {...}

func send_ints(n int) func(_x *_state_0) {
    return func(c *_state_0) {
        c0 := c.Send(n)
        c1 := c0.Send((n + 1))
        c2 := c1.Send((n + 2))
        c2.Send(nil)
    }
}
func main() {
    m := init_state_3(make(chan interface{}))
    go func() {
        m.Recv()
    }()
    func(m *_state_3) {
        e := init_state_0(make(chan interface{}))
        go send_ints(1)(e)
        a1, e0 := e.Recv()
        fmt.Printf("%v\n", a1)
        a2, e1 := e0.Recv()
        fmt.Printf("%v\n", a2)
        a3, e2 := e1.Recv()
        fmt.Printf("%v\n", a3)
        e2.Recv()
        m.Send(nil)
    }(m)
}
```

Listing 5.8 (a): Generated code without the package messages optimization

```go
// Preamble generation
type _state_1 struct {c chan interface{}}
func init_state_1(c chan interface{}) *_state_1 { return &_state_1{c} }
func (x *_state_1) Send(v interface{})          { x.c <- v }
func (x *_state_1) Recv() interface{}           { return <-x.c }

type _state_0 struct {...}
func init_state_0(c chan interface{}) *_state_0 { return &_state_0{c, nil} }
func (x *_state_0) Send(v0 int, v1 int, v2 int) *_state_1 {
    //...
    x.c <- []interface{}{v0, v1, v2}
    return x.next
}
func (x *_state_0) Recv() (int, int, int, *_state_1) {
    //...
    ll := <-x.c
    l := ll.([]interface{})
    return l[0].(int), l[1].(int), l[2].(int), x.next
}
//Declaration list compilation
func send_ints_optimized(n int) func(_x *_state_0) {
    return func(c *_state_0) {
        c0 := c.Send(n, (n + 1), (n + 2))
        c0.Send(nil)
    }
}
// Main compilation
func main() {
    m := init_state_1(make(chan interface{}))
    go func() {
        m.Recv()
    }()
    func(m *_state_1) {
        e := init_state_0(make(chan interface{}))
        go send_ints_optimized(1)(e)
        a1, a2, a3, e0 := e.Recv()
        fmt.Printf("%v\n", a1)
        fmt.Printf("%v\n", a2)
        fmt.Printf("%v\n", a3)
        e0.Recv()
        m.Send(nil)
    }(m)
}
```

Listing 5.8 (b): Generated code with the package messages optimization - Version in list

```go
// Preamble generation
type _state_1 struct {...}
func init_state_1(c chan interface{}) *_state_1 {...}
func (x *_state_1) Send(v interface{})          { ... }
func (x *_state_1) Recv() interface{}           { ... }

type _state_0 struct { ... }
func init_state_0(c chan interface{}) *_state_0 { ... }
type _multisend_type_state_0 struct {
  v0 int
  v1 int
  v2 int
}
func (x *_state_0) Send(v0 int, v1 int, v2 int) *_state_1 {
  //...
  x.c <- _multisend_type_state_0{v0, v1, v2}
  return x.next
}
func (x *_state_0) Recv() (int, int, int, *_state_1) {
  //...
  ll := <-x.c
  l := ll.(_multisend_type_state_0)
  return l.v0, l.v1, l.v2, x.next
}
//Declaration list compilation
...
// Main compilation
...
```

Listing 5.8 (c): Generated code with the package messages optimization - Version in struct

6 Evaluation

The purpose of our work is to improve the performance of the code generated by the compiler developed by Geraldo [11, 12]. After presenting the optimizations and their variations from the original version, our focus now shifts to evaluating their actual influence on the performance of the generated code. Along with analyzing the generated code's efficacy, we also assess the optimizations' impact on the compiler's performance. It's worth noting that minor enhancements in code performance may not justify the effort put into optimization or significant increases in the compilation time.

As the optimizations were implemented, we created multiple examples to test and validate the results obtained. Therefore, we utilized 20 examples[1]. Out of the total number, 3 examples examine the forward optimization, 9 examples evaluate the recursion optimization, and 8 examples analyze the optimization associated with message grouping. By comparing the output of the original version with the output of the optimized version, we confirmed that the result remained consistent in all cases. This brief qualitative assessment and the informal reasoning explained for each optimization in Chapter 5 leads us to believe that our optimizations preserve the behavior of the programs.

Considering our first modification to the compiler was to enhance its performance by shifting the code generation to use a buffer instead of string concatenation, we anticipate a slight enhancement in compilation time with the updated version. After promptly analyzing it with the basic examples, the average improvement rate for the compilation times is about 14% when utilizing the buffer. When we compare the compilation times of larger programs the string concatenation really shows its burden, making the compilation times up to 100 times slower than the buffer version. Therefore, we will compare compilation times between the new version using the buffer with the optimization phase toggled off and on to assess its effect on compilation times.

To accurately compute the execution times of the generated code we use Go's `Test` package with the benchtime functionalities [45]. Additionally, to measure the compilation

[1]All examples can be consulted in the project's GitHub repository [14]

times we use the Ocaml library `Core_bench` [46]. We select a representative example for each optimization and measure the compilation and execution times. Next, we scale up the base example to better evaluate the impact of the optimization on larger programs.

In the following sections, we calculate the percentage of differences between the new optimized version and the original version for both execution and compilation times. The percentage of difference is calculated using the following formula:

$$\frac{Time\ of\ original\ version - Time\ of\ optimized\ version}{Time\ of\ optimized\ version} \times 100$$

6.1 Forward optimization

For the forwarding optimization, the three examples[2] used showed an average improvement of 42% in the execution times but for these simple programs, the compilation times were, on average 6,8% slower. Let us consider the further example to better assess the impact of this optimization:

```
plus_one : int -> {int ^ @}
fun n -> c <- {
    send c (n+1);
    close c
}
end;
plus_two : int -> {int ^ @}
fun n -> c <- {
    d <- spawn (plus_one (n+1));
    fwd d c
}
end;

;; m <- {
    d <- spawn (plus_two 1);
    a:int <- recv d;
    print a;
    wait d;
    close m
};;
```

For this example, execution time improved by 56% with optimizations, although compilation time increased by 14%. To determine the impact of the optimization in larger programs we will scale up the example by increasing the number of processes spawned by the `main` function. The results for both execution and compilation times are presented in Table 6.1 and these results result in Figure 6.1.

We can observe a slight increase in performance gain as the number of calls increases, albeit a small one. Surprisingly, the potential negative impact of the optimization phase appears to be lessened slightly in larger programs.

[2]Examples in project's GitHub repository [14]: `a.prog`, `c.prog`, `d.prog`

Table 6.1: Results for execution and compilation times for the forward optimization

Num. Calls	Compilation			Execution		
	Non-Opt (μs)	Opt (μs)	Diff (%)	Non-Opt (ns)	Opt (ns)	Diff (%)
2	8.7	9.5	-8.1	5,191.2	3,243.4	60.1
4	12.0	12.4	-3.5	9,398.0	5,649.8	66.3
8	18.7	19.5	-4.0	18,289.4	10,781.8	69.6
16	32.3	32.6	-1.0	36,516.6	21,944.2	66.4
32	71.4	74.6	-4.4	75,169.0	44,860.6	67.6
64	177.2	189.8	-6.7	151,902.8	86,422.2	75.8
128	529.2	542.8	-2.5	296,461.4	175,754.0	68.7

The optimized version is faster in all cases, with the gains in execution time outweighing the losses in compilation time. On average, the execution times for all versions of this example improve by 66%, while the compilation times are 6% slower. This results in a global average improvement of 60%, demonstrating the value of this optimization.

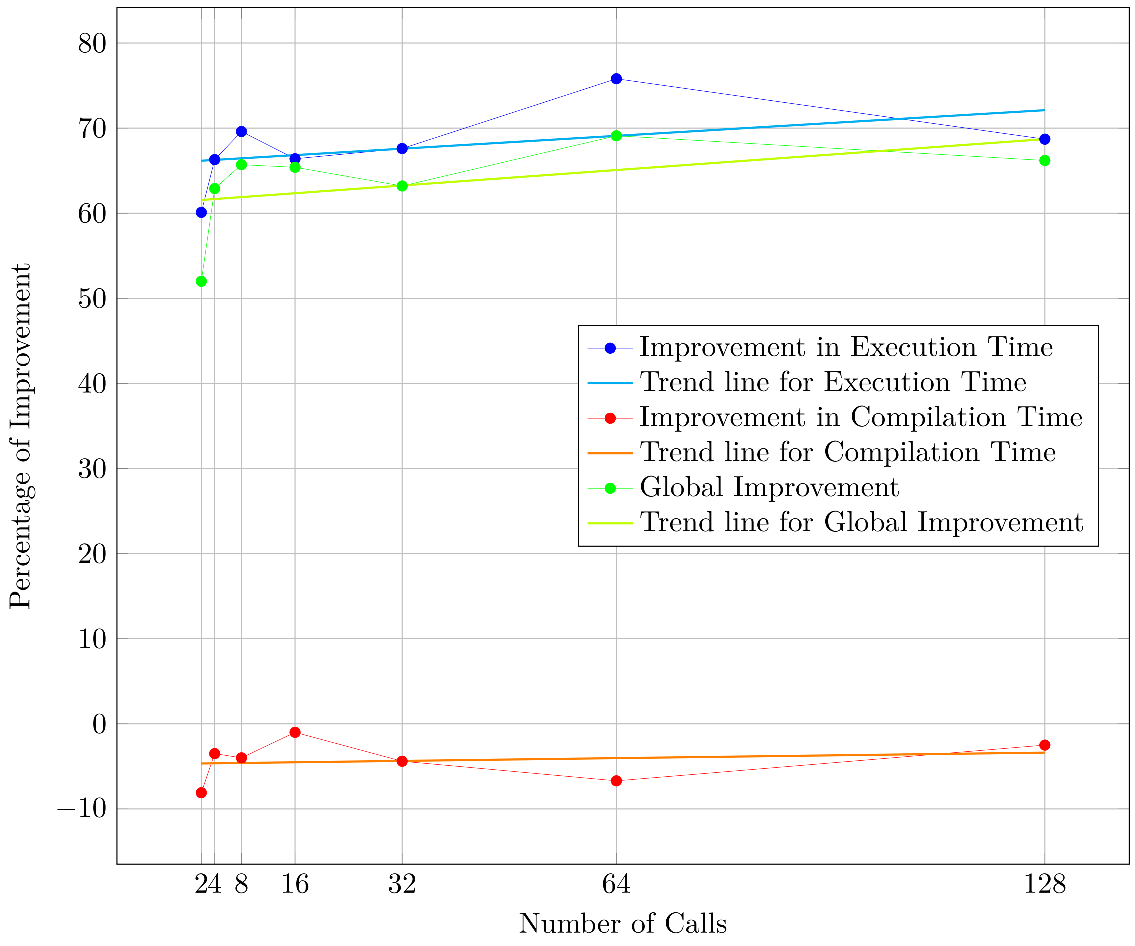

Figure 6.1: Difference in execution and compilation times for the forward optimization, increasing the number of calls

6.2 Recursion optimization

The nine examples[3] employed for recursion optimization evaluation demonstrated an average 149% enhancement in execution times and 22% decrease in compilation times. Although the improvement in execution times is substantial, the noteworthy and surprising decrease in compilation times cannot be ignored. As the declarations that are optimized for recursion are excluded from the most expensive optimization, the packing messages optimization, a smaller impact on compilation times was expected. However, the compilation times are still faster than the version without optimizations, suggesting that the subsequent phases may be easier to execute with the optimized representation.

Consider the following example for a better understanding of the impact of this optimization:

```
stype IntCStream rec x. &{next: int^x, stop: @};
fib : int -> int -> {IntCStream}
fun n acc -> c <- {
    case c of
        next: (
            send c n;
            d <- spawn (fib acc (n + acc));
            fwd d c
        ) stop: (
            close c
        )
} end;
;; m <- {
    f <- spawn (fib 0 1);
    f.next;
    x1:int <- recv f;
    f.stop;
    wait f;
    close m
} ;;
```

In this example, we have only one call to the recursive function `fib` and yet we have already seen a 40% improvement in execution time when comparing the optimized version to the non-optimized version. We enhance this example, by increasing the number of iterations of the recursive function, to better assess the impact of the optimization on larger programs. The results for both execution and compilation times are presented in Table 6.2 and shown in Figure 6.2.

As the number of iterations of the recursive function increases, the benefits of recursion optimization grow significantly, achieving an outstanding 3395% reduction in execution time at 128 iterations, execution is 35 times faster. The compilation times are not adversely affected, and an average improvement in compilation times of 12%, across all versions of this example demonstrates that this optimization is a valuable addition to the compiler.

[3]Examples in project's GitHub repository [14]: `bit_counter.prog`, `calculator.prog`, `CCO_queue.prog`, `filter.prog`, `fibbonacci.prog`, `intList.prog`, `intStack.prog`, `map-reduce.prog`, `nats.prog`

Table 6.2: Results for execution and compilation times for the recursion optimization

Num. iterations	Compilation Non-Opt (μs)	Compilation Opt (μs)	Compilation Diff (%)	Execution Non-Opt (ns)	Execution Opt (ns)	Execution Diff (%)
1	20.7	16.4	26.1	4,308.2	3,064.0	40.6
2	22.7	18.1	25.1	8,017.0	4,613.4	73.8
4	26.1	21.8	19.7	17,204.6	6,921.8	148.6
8	33.8	30.2	11.9	45,088.2	14,460.8	211.8
16	48.0	44.3	8.2	146,314.0	25,422.4	475.5
32	76.1	73.1	4.0	484,144.6	52,208.2	827.3
64	133.1	130.8	1.8	1,821,613.8	102,567.0	1,676.0
128	249.7	249.4	0.1	7,078,160.0	202,500.6	3,395.4

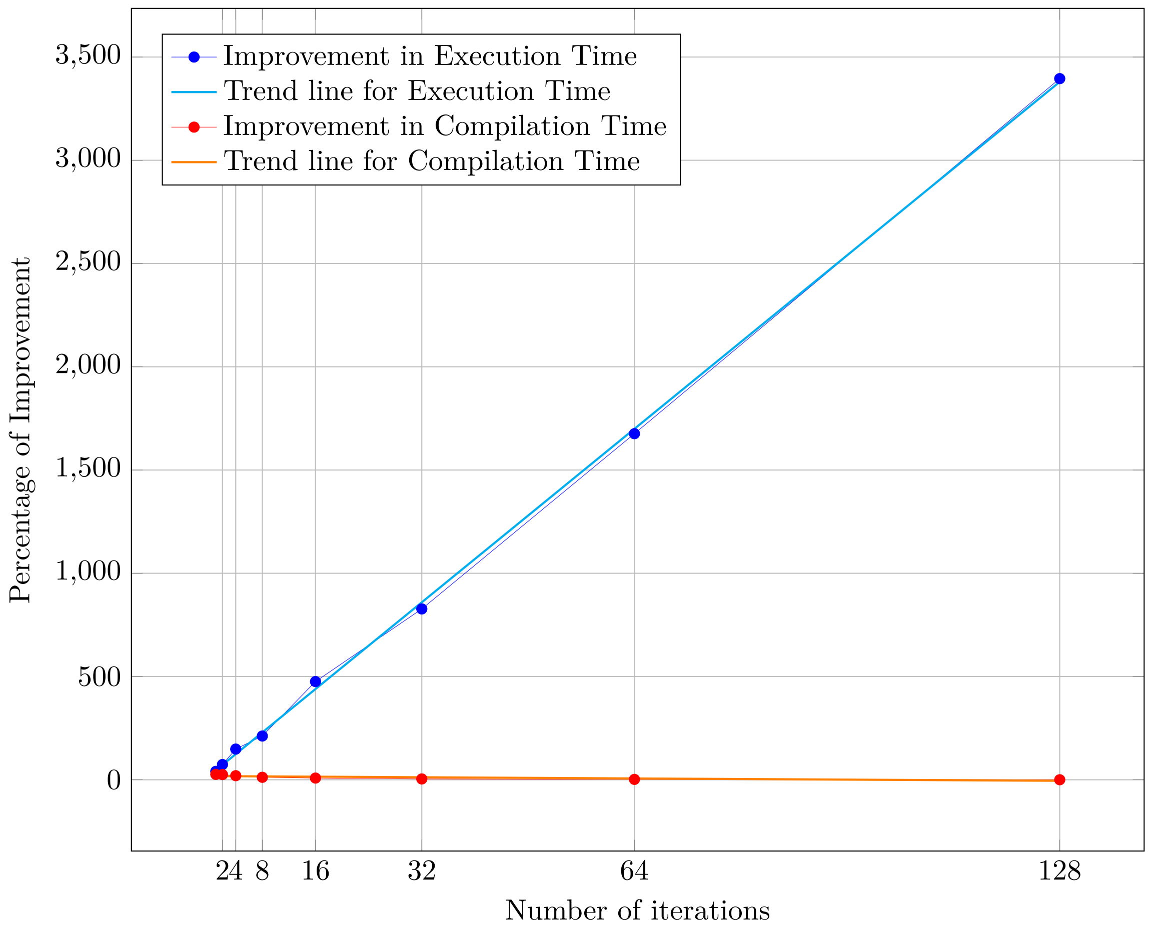

Figure 6.2: Difference in execution and compilation times for the recursion optimization, increasing the number of iterations

6.3 Packing messages optimization

All the examples testing the packing messages optimization[4] account for an average 18% improvement in execution times and an 18,5% decrease in compilation times. While the improvement in execution times may not be as significant as the other optimizations, it still yields a positive result. To better evaluate the effect of this optimization, we will examine the following example:

```
send_ints : int -> {int ^ int ^ int ^ @}
fun n -> c <- {
    send c n;
    send c (n+1);
    send c (n+2);
    close c
} end;
;; m <- {
    e <- spawn (send_ints 1);
    a1:int <- recv e;
    print a1;
    a2:int <- recv e;
    print a2;
    a3:int <- recv e;
    print a3;
    wait e;
    close m
};;
```

This example shows a 14% improvement in execution time and a 4% decrease in compilation time. Although the optimization phase can cause worse compilation times, we observe that they generally decrease with this optimization. This outcome is not surprising considering that chains of `Send` and `Recv` nodes are replaced by a single `MultiSend` or `MultiRcv` node, which lightens the processing load of the AST during the code generation phase. Therefore, the optimization phase's added time is counterbalanced by a more straightforward code generation phase. This example is scaled up by increasing the number of messages sent/received to better assess the impact of the optimization on larger programs.

As stated in Section 5.3, we conducted tests on two optimization approaches, one utilizing a list and the other utilizing a struct, the results are summarized in Table 6.3. The results for both list and struct execution are shown in Figure 6.3. The impact in compilation times is shown in Figure 6.4.

As we observe, there are no noteworthy differences discovered between the list approach and the struct approach. This suggests that the creation of structs bears its own costs, which appear similar to the expenses of casting the values of the list. In programs with a large number of messages bundled together, we can observe a nearly 100% improvement

[4]Examples in project's GitHub repository [14]:, `cripto_miner.prog`, `receive-send.prog`, `send_receive_choice.prog`, `send_receive_separeted.prog`, `send_receive_two_functions.prog`, `send-receive-simple.prog`, `send-receive-spawn-in-function.prog`, `send-receive-spawn.prog`

in execution times, which is a significant improvement. This stems from the reduced need for synchronization points between the processes, which is an expensive operation.

Table 6.3: Results for execution and compilation times for the packing messages optimization

	Compilation			Execution					
					List Version		Struct Version		List vs.
Num Mess.	Non-Opt (μs)	Opt (μs)	Diff (%)	Non-Opt (ns)	Opt (ns)	Diff (%)	Opt (ns)	Diff (%)	Struct (%)
2	7.1	12.3	42.4	2,551.2	2,373.4	7.5	2,355.4	8.3	−0.8
4	12.1	19.0	36.3	3,776.6	3,321.8	13.7	3,196.0	18.2	−3.8
8	26.2	38.0	31.0	6,462.0	5,238.6	23.4	5,241.0	23.3	0.0
16	79.8	101.4	21.4	11,487.4	9,293.6	23.6	9,462.6	21.4	1.8
32	356.6	388.4	8.2	22,610.8	15,753.8	43.5	14,931.4	51.4	−5.2
64	2.1	2.2	2.3	44,977.4	27,100.4	66.0	28,131.8	59.9	3.8
128	15.5	15.5	−0.1	99,783.8	50,990.0	95.7	51,315.0	94.5	0.6

In the results concerning compilation times, it is evident that the improvements decrease with an increase in the number of messages. This may be due to the fact that the optimization phase's burden is no longer counterbalanced by the faster code generation phase. Nevertheless, it is reassuring to note that this optimization does not introduce any relevant strain in compilation and has a positive overall impact on performance.

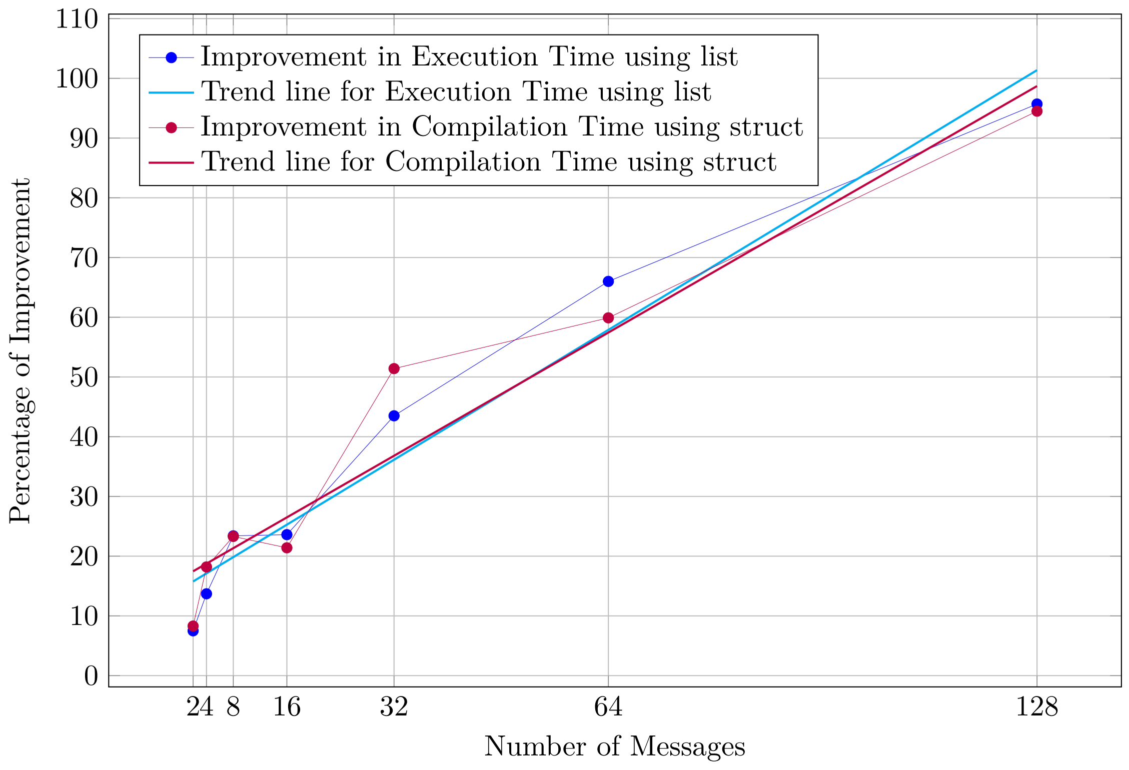

Figure 6.3: Difference in execution times for the packing messages optimization, increasing the number of messages

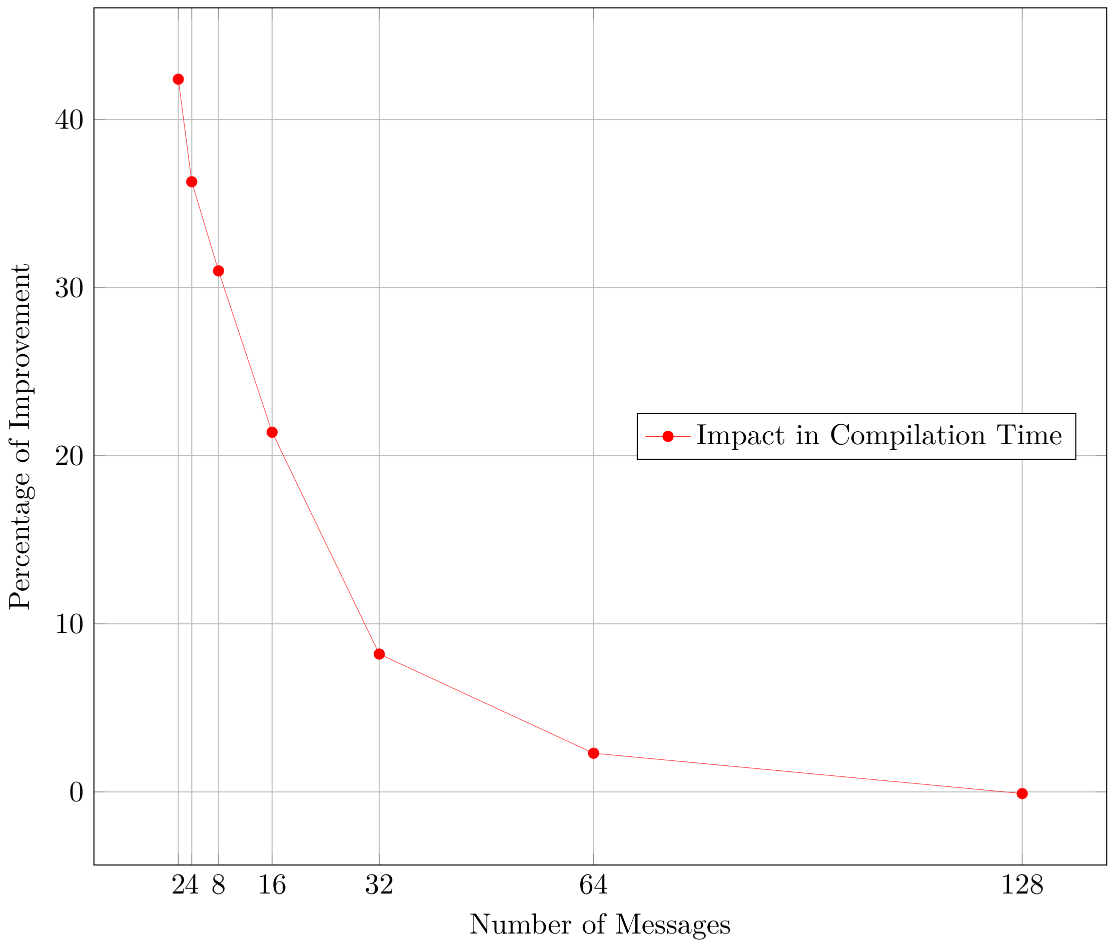

Figure 6.4: Impact of the packing messages optimization in compilation times

7 Conclusion

In this study, an optimization phase was implemented in a compiler for a session-typed language previously developed by Geraldo [11, 12]. The optimization phase consists of three optimizations: forwarding mechanism optimization, recursion optimization, and message packing optimization. A qualitative analysis was conducted to ensure the preservation of the program's behavior after optimization. We also conducted a quantitative analysis to assess the effect of the optimizations on the performance of the generated code and the compilation time burden.

The forwarding was optimized by eliminating unnecessary forward operations. Recursion optimization was achieved by converting it into cycles. Message packing was optimized by bundling sequential messages into a single bundle, resulting in fewer synchronization points.

All of the optimizations proved to be meritorious, particularly the recursion optimization, which greatly improved performance. The message packing optimization also showed a significant impact, especially for lengthy message sequences. None of the optimizations had a perceptible negative effect on compilation time, further proving their value. In fact, in most cases, the optimized version had better compilation times.

Future work can focus on two main concerns: expanding the optimization of packing messages and compiling to LLVM instead of Go. The optimization of packing messages can be expanded to include more complex scenarios, such as recursion, including more operations between `Send` nodes, and optimize with channels provided as parameters to spawned processes. Furthermore, the Go programming language is a high-level language. Compiling to LLVM can result in additional optimizations since LLVM uses a language-independent intermediate representation that serves as a portable, high-level assembly language, and can be optimized with various transformations over multiple passes.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[2] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Redwood City, Calif: Benjamin/Cummings Pub. Co, 1991. 637 pp. ISBN: 978-0-8053-0086-4 (cit. on pp. 1, 4, 5).

[3] K. Honda, V. T. Vasconcelos, and M. Kubo. "Language Primitives and Type Discipline for Structured Communication-Based Programming". In: *Programming Languages and Systems*. Ed. by C. Hankin. Red. by G. Goos, J. Hartmanis, and J. van Leeuwen. Vol. 1381. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 122–138. ISBN: 978-3-540-64302-9 978-3-540-69722-0. DOI: 10.1007/BFb0053567 (cit. on pp. 2, 5–10).

[4] R. Chen, S. Balzer, and B. Toninho. *Ferrite: A Judgmental Embedding of Session Types in Rust*. 2022-05-31. DOI: 10.48550/arXiv.2009.13619. arXiv: 2009.13619. (Visited on 2023-02-06). preprint (cit. on p. 2).

[5] K. Imai, S. Yuen, and K. Agusa. "Session Type Inference in Haskell". In: *Electronic Proceedings in Theoretical Computer Science* 69 (2011-10-18), pp. 74–91. ISSN: 2075-2180. DOI: 10.4204/EPTCS.69.6. arXiv: 1110.4163 [cs]. (Visited on 2023-02-06) (cit. on p. 2).

[6] J. Polakow. "Embedding a Full Linear Lambda Calculus in Haskell". In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell '15. New York, NY, USA: Association for Computing Machinery, 2015-08-30, pp. 177–188. ISBN: 978-1-4503-3808-0. DOI: 10.1145/2804302.2804309. (Visited on 2023-02-06) (cit. on p. 2).

[7] R. Pucella and J. A. Tov. "Haskell Session Types with (Almost) No Class". In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell '08. New York, NY, USA: Association for Computing Machinery, 2008-09-25, pp. 25–36. ISBN: 978-1-60558-064-7. DOI: 10.1145/1411286.1411290. (Visited on 2023-02-06) (cit. on p. 2).

[8] M. Neubauer and P. Thiemann. “An Implementation of Session Types”. In: *Practical Aspects of Declarative Languages*. Ed. by B. Jayaraman. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 56–70. ISBN: 978-3-540-24836-1. DOI: 10.1007/978-3-540-24836-1_5 (cit. on p. 2).

[9] B. Almeida, A. Mordido, and V. T. Vasconcelos. “FreeST: Context-free Session Types in a Functional Language”. In: *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*. Ed. by F. Martins and D. Orchard. Vol. 291. EPTCS. 2019, pp. 12–23. DOI: 10.4204/EPTCS.291.2 (cit. on pp. 2, 10, 12).

[10] M. Willsey, R. Prabhu, and F. Pfenning. “Design and Implementation of Concurrent C0”. In: *Electronic Proceedings in Theoretical Computer Science* 238 (2017-01-17), pp. 73–82. ISSN: 2075-2180. DOI: 10.4204/EPTCS.238.8. (Visited on 2022-12-08) (cit. on pp. 2, 10, 11).

[11] J. Geraldo. “Making Session Types Go”. Master Thesis in Computer Science. NOVA School of Science and Technology, 2022-11 (cit. on pp. 2, 27, 50, 58).

[12] J. Geraldo and B. Toninho. *Making Session Types Go - Compilation of a Session-Typed Functional Language to Go* (cit. on pp. 2, 27, 28, 50, 58).

[13] B. Toninho, L. Caires, and F. Pfenning. “Higher-Order Processes, Functions, and Sessions: A Monadic Integration”. In: *Programming Languages and Systems*. Ed. by M. Felleisen and P. Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 350–369. ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_20 (cit. on pp. 2, 10, 22, 25, 26).

[14] JoanaSoaresF. *Sessint-Compiler*. 2023-09-22. URL: https://github.com/JoanaSoaresF/sessint-compiler (visited on 2023-09-22) (cit. on pp. 3, 50, 51, 53, 55).

[15] R. De Nicola. “Process Algebras”. In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Boston, MA: Springer US, 2011, pp. 1624–1636. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_450 (cit. on p. 5).

[16] R. Milner, ed. *A Calculus of Communicating Systems*. Red. by G. Goos et al. Vol. 92. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980. ISBN: 978-3-540-10235-9 978-3-540-38311-6. DOI: 10.1007/3-540-10235-3 (cit. on p. 5).

[17] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. “A Theory of Communicating Sequential Processes”. In: *Journal of the ACM* 31.3 (1984-06-26), pp. 560–599. ISSN: 0004-5411. DOI: 10.1145/828.833 (cit. on p. 5).

[18] J. Bergstra and J. Klop. “Process Algebra for Synchronous Communication”. In: *Information and Control* 60.1 (1984), pp. 109–137. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(84)80025-X (cit. on p. 5).

[19] R. Milner, J. Parrow, and D. Walker. "A Calculus of Mobile Processes, I". In: *Information and Computation* 100.1 (1992), pp. 1–40. ISSN: 0890-5401. DOI: 10.1016/0890-5401(92)90008-4 (cit. on p. 5).

[20] D. Sangiorgi and D. Walker. *The [Pi]-Calculus: A Theory of Mobile Processes.* 1st paperback edition. Cambridge New York Port Melbourne: Cambridge University Press, 2003. 580 pp. ISBN: 978-0-521-54327-9 978-0-521-78177-0 (cit. on p. 5).

[21] N. Kobayashi, B. C. Pierce, and D. N. Turner. "Linearity and the Pi-Calculus". In: *ACM Transactions on Programming Languages and Systems* 21.5 (1999-09-01), pp. 914–947. ISSN: 0164-0925. DOI: 10.1145/330249.330251 (cit. on p. 5).

[22] F. Pfenning and D. Griffith. "Polarized Substructural Session Types". In: *Foundations of Software Science and Computation Structures.* Ed. by A. Pitts. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2015, pp. 3–22. ISBN: 978-3-662-46678-0. DOI: 10.1007/978-3-662-46678-0_1 (cit. on p. 10).

[23] B. Toninho, L. Caires, and F. Pfenning. "Dependent Session Types via Intuitionistic Linear Type Theory". In: *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming.* PPDP '11. New York, NY, USA: Association for Computing Machinery, 2011-07-20, pp. 161–172. ISBN: 978-1-4503-0776-5. DOI: 10.1145/2003476.2003499 (cit. on p. 10).

[24] P. Rocha and L. Caires. "Safe Session-Based Concurrency with Shared Linear State". In: *European Symposium of Programming* (2023), p. 107 (cit. on p. 10).

[25] P. Rocha and L. Caires. "Propositions-as-Types and Shared State". In: *Proceedings of the ACM on Programming Languages* 5 (ICFP 2021-08-22), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3473584. (Visited on 2022-12-08) (cit. on p. 10).

[26] D. Griffith. "Polarized Substructural Session Types". University of Illinois at Urbana-Champaign, 2016. URL: https://www.cs.cmu.edu/~fp/theses/griffith16.pdf (cit. on p. 11).

[27] P. Thiemann and V. T. Vasconcelos. "Context-Free Session Types". In: *SIGPLAN Not.* 51.9 (2016-09), pp. 462–475. ISSN: 0362-1340. DOI: 10.1145/3022670.2951926 (cit. on p. 12).

[28] B. Almeida, A. Mordido, and V. T. Vasconcelos. "Deciding the Bisimilarity of Context-Free Session Types". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by A. Biere and D. Parker. Cham: Springer International Publishing, 2020, pp. 39–56. ISBN: 978-3-030-45237-7 (cit. on p. 12).

[29] A. V. Aho et al. *Compilers: Principles, Techniques, and Tools.* USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0-321-48681-1 (cit. on pp. 13–19).

[30] J. Dunfield. "Bidirectional Typechecking". In: *McGill University COMP 302* (2012-11-09), p. 6 (cit. on p. 14).

[31] J. Dunfield and N. Krishnaswami. “Bidirectional Typing”. In: *ACM Computing Surveys* 54.5 (2022-06-30), pp. 1–38. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3450952. (Visited on 2022-11-27) (cit. on pp. 14, 15).

[32] S. Lindley, C. McBride, and C. McLaughlin. “Do Be Do Be Do”. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* POPL ’17. New York, NY, USA: Association for Computing Machinery, 2017-01-01, pp. 500–514. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009897. (Visited on 2023-02-06) (cit. on p. 15).

[33] S. P. Jones et al. “Practical Type Inference for Arbitrary-Rank Types”. In: *Journal of Functional Programming* 17.1 (2007), pp. 1–82. ISSN: 1469-7653. DOI: 10.1017/S0956796806006034 (cit. on p. 15).

[34] U. Norell. “Towards a Practical Programming Language Based on Dependent Type Theory”. Thesis for the degree of Doctor of Philosophy. Chalmers University of Technology and Goteborg University, 2007. 166 pp. URL: https://www.cse.chalmers.se/~ulfn/papers/thesis.pdf (cit. on p. 15).

[35] A. Serrano et al. “A Quick Look at Impredicativity”. In: *Proc. ACM Program. Lang.* 4 (ICFP 2020-08). DOI: 10.1145/3408971 (cit. on p. 15).

[36] F. E. Allen and J. Cocke. “A Catalogue of Optimizing Transformations”. In: *IBM Technical Reports* (1971) (cit. on pp. 16–18).

[37] A. W. Appel. *Modern Compiler Implementation in ML.* Revised and expanded ed. of: Modern compiler implementation in ML : basic techniques, repr. with corr., first paperback ed. Cambridge: Cambridge Univ. Press, 2004. 538 pp. ISBN: 978-0-521-60764-3 978-0-521-58274-2 (cit. on pp. 19, 20).

[38] P. Wadler. “Deforestation: Transforming Programs to Eliminate Trees”. In: *Theoretical Computer Science* 73.2 (1990), pp. 231–248. ISSN: 0304-3975. DOI: 10.1016/0304-3975(90)90147-A (cit. on p. 20).

[39] R. Hinze, T. Harper, and D. W. H. James. “Theory and Practice of Fusion”. In: *Implementation and Application of Functional Languages.* Ed. by J. Hage and M. T. Morazán. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 19–37. ISBN: 978-3-642-24276-2. DOI: 10.1007/978-3-642-24276-2_2 (cit. on p. 20).

[40] *OCaml Library : String.* URL: https://v2.ocaml.org/api/String.html (visited on 2023-08-21) (cit. on p. 31).

[41] *OCaml Library : Buffer.* URL: https://v2.ocaml.org/api/Buffer.html (visited on 2023-08-21) (cit. on p. 31).

[42] B. Toninho. “A Logical Foundation for Session-Based Concurrent Computation”. Doctoral Thesis. NOVA School of Science and Technology, 2015-05. URL: https://run.unl.pt/handle/10362/15296 (visited on 2023-09-24) (cit. on pp. 33, 36, 38).

[43] L. Caires et al. “Behavioral Polymorphism and Parametricity in Session-Based Communication”. In: *Programming Languages and Systems*. Ed. by M. Felleisen and P. Gardner. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 330–349. ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_19 (cit. on pp. 33, 36).

[44] J. A. Pérez et al. “Linear Logical Relations and Observational Equivalences for Session-Based Concurrency”. In: *Information and Computation* 239 (2014), pp. 254–302. ISSN: 0890-5401. DOI: 10.1016/j.ic.2014.08.001 (cit. on pp. 33, 36).

[45] *Go Packages Documentation - Testing Flags*. URL: https://pkg.go.dev/cmd/go#hdr-Testing_flags (visited on 2023-09-05) (cit. on p. 50).

[46] *Getting Started with Core_bench*. GitHub. URL: https://github.com/janestreet/core_bench/wiki/Getting-Started-with-Core_bench (visited on 2023-08-31) (cit. on p. 51).

A Typing rules of our language

A.1 Communication values

$$\frac{\Psi \Vdash M : \tau \quad \Psi ; \Gamma ; \Delta \vdash P :: c : A}{\Psi ; \Gamma ; \Delta \vdash \mathsf{send}\ c\ M\ ;\ P :: c : \tau \wedge A} \wedge R$$

$$\frac{\Psi , x{:}\tau ; \Gamma ; \Delta , c{:}A \vdash Q_x :: d : D}{\Psi ; \Gamma ; \Delta , c{:}\tau \wedge A \vdash x \leftarrow \mathtt{recv}\, c\ ;\ Q_x :: d : D} \wedge L$$

$$\frac{\Psi , x{:}\tau\ ; \Gamma ; \Delta \vdash Q_x :: c : A}{\Psi\ ; \Gamma ; \Delta \vdash x \leftarrow \mathtt{recv}\, c\ ;\ Q_x :: c : \tau \supset A} \supset R$$

$$\frac{\Psi \Vdash M : \tau \quad \Psi\ ; \Delta , c{:}A \vdash P :: d : D}{\Psi ; \Gamma ; \Delta , c{:}\tau \supset A \vdash \mathsf{send}\ c\ M\ ;\ P :: d : D} \supset L$$

A.2 Forwarding and Termination

$$\frac{}{\Psi ; \Gamma ; d{:}A \vdash \mathtt{fwd}\, c\ d :: c{:}A}\ id$$

$$\frac{}{\Psi ; \Gamma ; \cdot \vdash \mathsf{close}\, c :: c : \mathbf{1}}\ \mathbf{1}R$$

$$\frac{\Psi ; \Gamma ; \Delta \vdash P :: d : D}{\Psi ; \Gamma ; \Delta , c{:}\mathbf{1} \vdash \_ \leftarrow \mathtt{wait}\ c\ ;\ P :: d : D}\ \mathbf{1}L$$

A.3 Linear Channel Communication

$$\frac{\Psi ; \Gamma ; \Delta , d{:}A \vdash R_d :: c : B}{\Psi ; \Gamma ; \Delta \vdash d \leftarrow \mathtt{recv}\ c\ ;\ R_d :: c : A \multimap B} \multimap \mathsf{R}$$

$$\frac{\Psi;\Gamma;\Delta \vdash P_d :: d : A \quad \Psi;\Gamma;\Delta', c{:}B \vdash Q :: e : E}{\Psi;\Gamma;\Delta,\Delta', c{:}A \multimap B \vdash \ \texttt{send}\ c\ (d \leftarrow P_d)\ ;\ Q :: e : E}\ \multimap\mathsf{L}$$

$$\frac{\Psi;\Gamma;\Delta \vdash P_d :: d : A \quad \Psi;\Gamma;\Delta' \vdash Q :: c : B}{\Psi;\Gamma;\Delta,\Delta' \vdash \texttt{send}\ c\ (d \leftarrow P_d)\ ;\ Q :: c : A \otimes B}\ \otimes\mathsf{R}$$

$$\frac{\Psi;\Gamma;\Delta, d{:}A, c{:}B \vdash R_d :: e : E}{\Psi;\Gamma;\Delta, c{:}A \otimes B \vdash d \leftarrow \texttt{recv}\ c\ ;\ R_d :: e : E}\ \otimes\mathsf{L}$$

A.4 Choice and Branching

$$\frac{\Psi;\Gamma;\Delta \vdash P_1 :: c : A_1 \quad \ldots \quad \Psi;\Gamma;\Delta \vdash P_k :: c : A_k}{\Psi;\Gamma;\Delta \vdash \texttt{case}\ c\ \texttt{of}\ \overline{l_j \Rightarrow P_j} :: c : \&\{\overline{l_j : A_j}\}}\ \&\mathsf{R}$$

$$\frac{\Psi;\Gamma;\Delta, c{:}A_j \vdash P :: d : D}{\Psi;\Gamma;\Delta, c{:}\ \&\ \{\overline{l_j : A_j}\} \vdash \texttt{\_} \leftarrow c.l_j\ ;\ P :: d : D}\ \&\mathsf{L}$$

$$\frac{\Psi;\Gamma;\Delta \vdash P :: c : A_j}{\Psi;\Gamma;\Delta \vdash \texttt{\_} \leftarrow c.l_j\ ;\ P :: c : \oplus\{\overline{l_j : A_j}\}}\ \oplus\mathsf{R}$$

$$\frac{\Psi;\Gamma;\Delta, c{:}A_1 \vdash P_1 :: d : D \quad \ldots \quad \Psi;\Gamma;\Delta, c{:}A_k \vdash P_k :: d : D}{\Psi;\Gamma;\Delta, c{:} \oplus \{\overline{l_j : A_j}\} \vdash \texttt{case}\ c\ \texttt{of}\ \overline{l_j \Rightarrow P_j} :: d : D}\ \oplus\mathsf{L}$$